IMPLEMENTATION ISSUES FOR A NETWORK CONTINUOUS-MEDIA I/O SERVER

George Homsy
Ramesh Govindan
David P. Anderson

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

International Computer Science Institute
1947 Center St., Berkeley, CA 94704

September 19, 1990

ABSTRACT

ACME (Abstractions for Continuous Media) is a set of abstractions for input/output of digital audio and video, intended as an addition to a network window system. This report discusses the design issues in implementing an ACME server, including 1) the software structure of the server; 2) the process structure of the ACME component of the server; 3) the interface to CM I/O devices, and 4) synchronization of CM data streams.

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

1. INTRODUCTION

Abstractions for Continuous MEdia (ACME) is a set of abstractions that allow multiple concurrent clients to output or input digital audio and video data on a workstation. ACME uses the approach of *integrated digital continuous media* (IDCM) in which CM data is handled in the same hardware and software framework as other data.

This report discusses the design and implementation of an ACME server. The remainder of this section summarizes ACME and proposes a software structure for an ACME server, involving an ACME kernel, a base window system, and a window system extension. Section 2 discusses the implementation of an ACME server. Sections 3 and 4 discuss details of CM device interfaces and network communication. Section 5 discusses the internal details of the ACME kernel. An Appendix lists the interface between the ACME kernel and other components.

1.1. Summary of ACME

These ACME abstractions are briefly summarized below; a more detailed description is given in [1].

Ropes, Strands, and CM Connections

A *strand* is a stream of audio or video data encoded in a byte stream. Each strand has a *strand type* representing the encoding scheme. Multiple strands (say, an audio and a video stream) may be interleaved in a byte-stream *rope*; the interleaving scheme is called a *rope type*. A *CM connection* is network connection used to convey a strand or rope.

Logical Devices

A *logical device* (LDev) is an abstract CM I/O device. There are four types of LDevs: *VWins* (video output), *VCams* (video input), *listeners* (audio input), and *players* (audio output). LDevs have various *attributes* according to their type; for example, a VWin has the attributes of a graphics window: position, size, and stacking order. Clients can *map* LDevs to physical I/O devices. Multiple LDevs may be mapped to a single physical device; in the case of players, the server is responsible for "mixing" the respective outputs. The LDevs associated with the strands of a rope are grouped into a *compound logical device* (CLDev).

Logical Time Systems

LDevs and CLDevs can be associated with a *logical time system* (LTS). All strands in an LTS are played (or generated) in synchrony, even if they come from different sources. The ACME server ensures that the strands start playing at the same time and remain in lockstep. The client may also start, stop, or alter the speed of an LTS, affecting the component strands uniformly.

Resource Reservation

It may be desirable to guarantee the performance of each CM data stream (*i.e.*, to ensure that concurrent activity in the server or its host, the client or its host, or the intervening network, does not interfere with the timing of the input or output). ACME does not itself define a mechanism for doing this, but the ACME protocol makes a provision for attaching a real-time "session" to a CM connection. ACME can then use the real-time constraints of the session to determine the scheduling of its own processes.

1.2. Software Structure of an ACME Server

ACME is intended to be implemented as an extension of an existing network window system such as X11 or NeWS. The proposed software structure of the resulting "ACME server" includes the following components (see Figure 1):

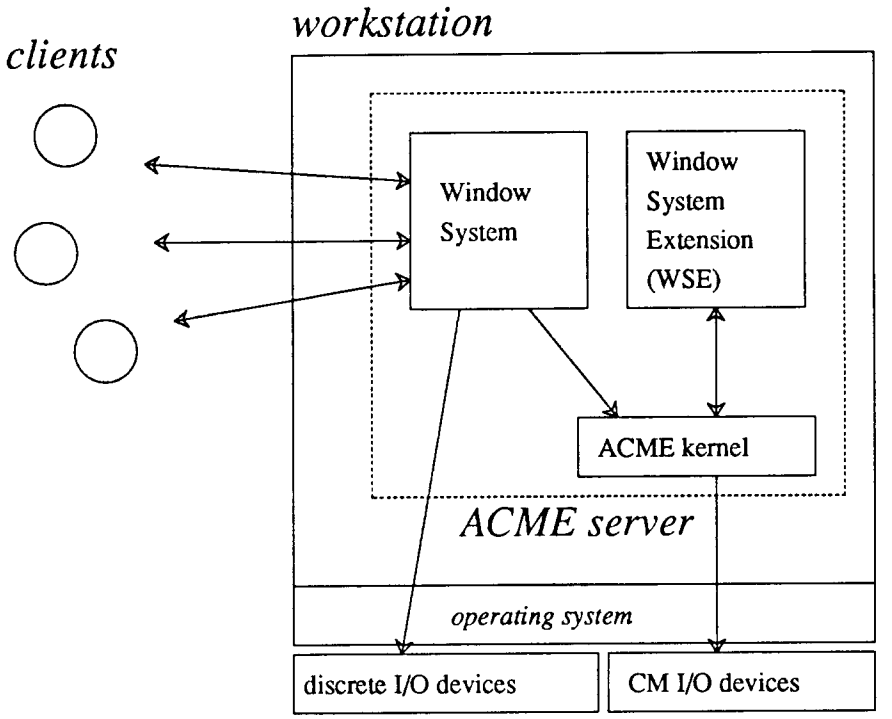

Figure 1: An ACME server consists of a base window system such X11 or NeWS, a window system extension (WSE), and the ACME kernel.

- The *window system* is a modified version of the existing window system. It is assumed to communicate with clients using a *window system protocol.* This protocol must be extended to include ACME requests, replies, and events. The additional duties of the window system include 1) conveying ACME-related protocol requests to the window system extension (see below); 2) making calls to the ACME kernel to inform it of changes in video window visibility; 3) exporting some utility procedures to the ACME kernel; 4) handling and distribution of events which are generated by ACME.
- The *window system extension* (WSE) is in charge of decoding extended protocol messages, and making the appropriate calls to the ACME kernel.
- The ACME kernel exports a procedural interface for creating and controlling instances of ACME abstractions. The kernel is largely device-independent.

2. IMPLEMENTING AN ACME SERVER

A typical ACME server implementation will consist of the base window system, the ACME kernel, and a window system extension, as shown in Figure 2. The window system extension has several functions:

- It implements a set of protocol requests that is an extension of the base window system protocol. These requests allow clients to gain access to ACME functions. Access to ACME requests from the WSE is via a procedural interface.

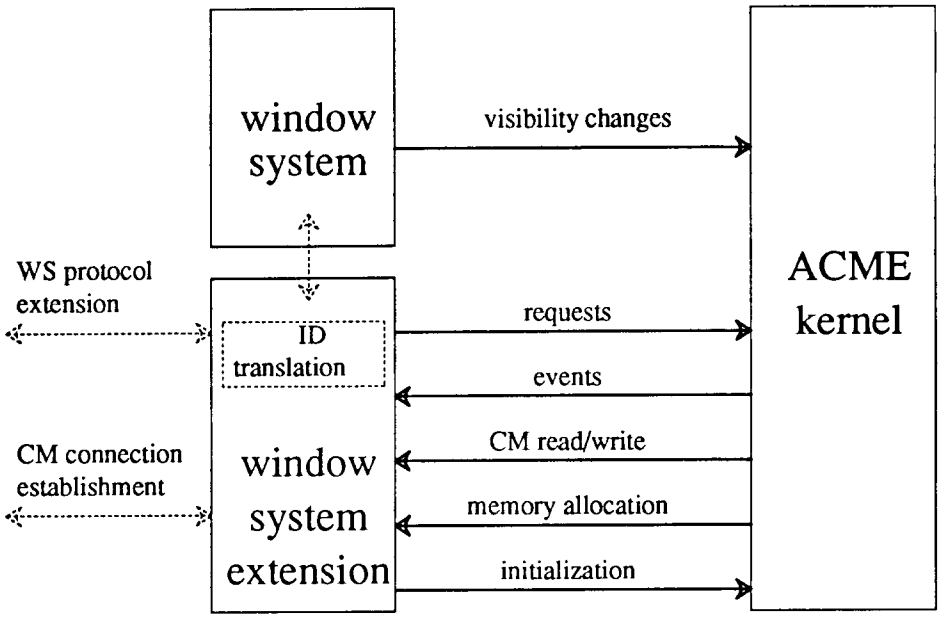

Figure 2: The interface between the window system extension (WSE), the base window system, and the ACME kernel. Bold arrows represent the ACME procedural interface. Dotted lines indicate window system dependent components.

- It establishes CM connections, either actively or passively. This function is in the WSE, rather than the ACME kernel, because the method of connection establishment and resource identifier assignment is window system dependent. The WSE also supplies functions to read and write CM data.
- It supplies event procedures callable by ACME for reporting events generated internally to ACME. These procedures may take action to handle the event, ignore the event, format the event and queue it for distribution, *etc.*
- It makes calls to initialize and terminate ACME, supplying an OS-independent memory allocation interface to ACME, and supplying a routine whereby ACME can queue a request for processing by the base window system.

The base window system itself must be modified or extended in two ways: 1) the internal window structure must be generalized to include VWins, and some provision must be made to distinguish a VWin from a window; 2) the window system must notify ACME of all changes to VWin visibility, using calls in the ACME procedural interface.

2.1. Implementation of the Window System Extension

2.1.1. Add ACME Requests to Interpreter

The protocol interpreter is the portion of the window system software which interprets the client byte stream as a sequence of requests, and causes the window system to perform the

appropriate actions. In NeWS, it is a modified PostScript interpreter. In the X11 sample server, it is the DIX layer of the server code.

The WSE defines an extension to the base window system protocol providing access to the ACME requests to clients. It must translate extension protocol requests into the appropriate procedure calls to the ACME kernel. These interface include calls to create, destroy, and manipulate ACME resources.

In the case of NeWS, postscript-callable primitives corresponding to the ACME requests would be made available to PostScript programs running on the server. In the case of X11, a protocol extension would be added which defines new requests.

2.1.1.1. Resource Creation Requests

Calls to `create_player`, `create_listener`, `create_vwin`, `create_vcam`, `create_cldev`, and `create_lts`. are made when the client requests that an ACME resource be created. The identifier is a pointer to a block of storage which will be initialized to contain the data for the resource. The WSE must preallocate this storage before calling the resource creation procedure. The required size of the storage is `sizeof(type)`.

The reason the WSE allocates storage is that VWins are associated with WS private data (configuration, stacking order, *etc.*) in a WS dependent manner. The window struct might be extended to contain a VWin struct, or a pointer to a VWin might be added to the window struct, or a pairwise association between VWins and windows could be kept in a hash table or other data structure. Clearly the WS is best equipped to handle the problems of storage allocation for VWins. We extend the preallocation requirement to all ACME resources for consistency.

ACME knows nothing about the WS private data (*e.g.*, stacking order) associated with a VWin. The only information available to ACME about a VWin is the region of the screen into which the video pixels should be transferred, and even this information must be maintained by the WSE. This is done using the requests in Section 2.2.2 below.

Within ACME, all resource identifiers are pointers to structs. If the identifiers used by the WS are of a different format than pointers, or if there is a risk of collision of identifiers, the WSE must maintain a map between client-supplied identifiers and pointers to ACME structs. This map can then be used to translate the resource IDs received in client requests into their corresponding pointer values. When a resource is destroyed, the WSE should destroy the entry in the map.

In pure NeWS, no ID translation is needed, since pointers serve as resource identifiers. In X11, however, the client is responsible for assigning IDs to resources it creates, so ID mapping is needed. The X11 sample server already has such a mapping facility, and a method for adding new resource types to it. OpenWindows provides a facility whereby X11 resource IDs can be associated with NeWS objects. In this case, the X11 interpreter must of course map client supplied resource IDs to pointers to ACME objects.

2.1.1.2. Resource Manipulation Requests

The calls `map_ldev`, `unmap_ldev`, `map_cldev`, `unmap_cldev`, `destroy_ldev`, `destroy_cldev`, `get_ldev_attributes`, `set_ldev_attributes`, `get_cldev_attributes`, `start_lts`, and `destroy_lts` manipulate existing resources. The functionality of these procedures is detailed in the Appendix. In general, the WSE should call these procedures whenever the client makes a corresponding request.

2.1.2. Management of CM Connections.

The WSE must provide a mechanism for establishment of CM connections, both actively

(server-initiated) and passively (client-initiated)[1]. If the base window system already supports active establishment of connections, as does NeWS, the CM connection establishment protocol may be an extension. Otherwise, a new request must be implemented, allowing clients to actively create CM connections. When a new CM connection has been established, the WSE registers it with ACME so that it can be identified by its ACME ID. If a CM connection exists between two ACME servers, each endpoint has its own ID. Client-level IDs are conveyed by two mechanisms:

(1) When a connection is created actively (in reponse to a client request), the ID is assigned in the normal manner for the particular window system: In X11, the ID is contained in the connection establishment request; and in NeWS, the connection establishment request returns the ID.

(2) Whan a CM connection is accepted (passively) by the ACME server, the connection ID must be conveyed by some other mechanism. In X11, the client sends the ID to the ACME server in a "registration" request containing enough information to identify the connection (*e.g.*, its port numbers, Internet addresses, and protocol number). In NeWS, the client queries the server for the connection ID.

In some applications, a client *C* may request that a server *A* establish a CM connection to a second server *B*. *C* learns the connection ID on server A using the mechanism (1). If *C* is also a client of server B, it learns the connection ID on server B using mechanism (2). If the CM connection is for use by another client *C'* of server B, *C* must convey to *C'* enough information to identify the connection (using a RPC protocol external to ACME), and *C'* then learns its ID using mechanism (2).

The WSE must also provide `read_cm_data()` and `write_cm_data()` procedures for CM connections, to be used by ACME. It is possible that some WSEs may wish to support more than one transport protocol. In this case, the WSE must be able to discern between connections using different protocols, so that the `read_cm_data()` and `write_cm_data()` procedures can perform the correct actions. The ACME CM connection descriptor has a WSE private data index, to be used for OS and WS dependent connection information, which can be used to make this determination.

2.1.3. Event Handling

Event reporting procedures are provided by the WSE for each event type generated by ACME. Whenever an ACME event is generated, the ACME kernel calls the appropriate procedure for that event type. These procedures may 1) take action to handle the event, 2) ignore the event, or 3) format and queue the event for delivery to interested client(s) by the window system. Lightweight processes are preemptively scheduled within the ACME kernel, but in order to simplify WSE implementation, ACME guarantees that calls to event reporting procedures will be non-reentrant.

2.1.4. Miscellaneous Modifications

`acme_init()` should be called once by the WSE before using any ACME global data structures or procedure calls. `acme_done()` should be called once by the WSE after it is done using all ACME global data structures and procedure calls.

Some storage management procedures must be provided by the WSE. These include `acme_alloc()`, `acme_free()`, and `acme_realloc()`. ACME uses these to manage storage for ACME internal data. Descriptions of these are provided in the Appendix.

---

[1] CM connections are simplex, and their data direction is independent of which end is active.

2.2. Modifications to the Base Window System

2.2.1. Modifications to the Window Descriptor Structure

The base window system's internal data structure must be able to contain VWins. More storage is required for a VWin descriptor than for a window descriptor because of VWin private data. Therefore the window descriptor must either be increased in size to hold the extra data, or a method must be devised for associating a VWin private data block with a VWin.

Some window systems, such as the X11 sample server, provide a means whereby extensions can extend the size of a window descriptor. In other cases, the window system will have a pointer field in the window descriptor which can be used as a pointer to an VWin private data block. In still other cases, the window system will have no provision for extending the window descriptor. In these cases, the WSE will have to maintain a mapping from window descriptor to VWin data block.

Also, there must be some method of distinguishing VWins from windows. An appropriate method for doing this will be obvious once it is decided how to extend the window descriptor.

2.2.2. Notifying ACME of VWin Visibility Changes

The base window system must be modified to notify ACME of the visible area ("clipping region") of each VWin. Any visibility changes on VWins must be reported to ACME so that it can update its private visibility structures. This is accomplished by means of the screen area arbitration requests. These are `vwin_move_notify`, `vwin_resize_notify`, `vwin_configure_notify`, and `vwin_clip_mask_notify`. These procedures allow the WS to inform ACME of a VWin's size, position, and visible region.

A window/VWin reconfiguration operation may involve several steps by the WS: 1) ACME operations as listed above; 2) damage events sent to clients; 3) copies between different parts of the frame buffer; 4) copies to backing store; 5) copies from backing store; 6) fills from background patterns. The WS must do these steps in a "safe" order, or else it is possible that video will incorrectly overwrite graphics. For example, suppose that a graphics window is partially overlapped by a sibling VWin, and that the graphics window is raised. If the clipmask of the VWin were modified after redrawing the graphics window, it is possible that the shared area will be overwritten with video data, thus leaving it damaged but with the WS unaware of the damage.

In general, there are multiple safe orders for any given reconfiguration problem. A simple scheme that guarantees a safe order is the "shrink-graphics-grow" scheme: Whenever any reconfiguration affects the visible regions of one or more VWins, let A and B be the regions of the screen on which the VWins in question are visible after and before the reconfiguration, respectively. Perform the reconfiguration as follows: First, reduce the VWins area to the intersection of A and B. Next, perform all needed graphics operations, such as drawing window borders, moving graphics image data, or filling with background patterns. Finally, modify the VWins clipmasks to their final shapes. This scheme has the property that the portion of a VWin not affected in the reconfiguration will continue to display video throughout the reconfiguration.

If a WS supports backing store, this means that it already has a mechanism for detecting when damage to a window is about to occur, since this is precisely when the contents of the about-to-be-damaged regions are saved into backing store. Such a WS could easily be modified to check whether the about-to-be-damaged window is a VWin and, if so, subtract the area about to be obscured from the VWin's clipmask instead of storing its contents in backing store. Since the damage detection must be done before the damage actually occurs in order for backing store to work properly, the WS automatically provides the correct ordering for shrink-graphics-grow.

Also, since events which notify clients of damage to their windows are normally sent after any graphics operations on the framebuffer are completed, a similar modification to the damage

event generation mechanism can be made to support growing of the VWin's clipmask: When damage is detected on a window, check if the window is a VWin. If so, then instead of generating a damage event for a given area and sending it to a client, simply add the area to the VWin's clipmask.

Another scheme for avoiding damage to graphics windows by VWins is to temporarily stop writing video data into the VWins which are affected by the reconfiguration. After the reconfiguration is complete, the video output can be restored. ACME provides the `freeze_vwin()` procedure for this purpose. Simpler still, but even worse functionally, is to freeze all video output during any reconfiguration using `freeze_video()`.

The WS may use any or all of these techniques to avoid damage to graphics. It may freely intermix the techniques. The guiding principle is that the WS has final responsibility for controlling ACME properly so as to avoid damage.

3. ACME PROCESS STRUCTURE

In this section, we discuss the process structure of an ACME server. We define a simple abstract model of CM I/O hardware, and discuss the operating system support necessary for implementing the server. We then describe the partitioning of server functionality into *real-time threads*, and how these threads communicate. Finally, we explain how logical time systems may be implemented.

3.1. Hardware Platform

In order to discuss video and audio devices uniformly, we need an abstraction that applies to both. For this purpose we define a *CM device group* as CM I/O hardware having the following properties.

- It has a memory (the *CM device memory*) that can be accessed by the CPU. This access may be direct or by DMA depending on the device group.
- It has one or more *transducers* that convert from device memory to analog output, or from analog input to device memory.
- It has zero or more *device coprocessors* (DCPs) that operate on the device memory. In some cases, the transducers may be connected to the device coprocessors (through, for instance, their serial ports). Input or output to the transducers may only be through the DCPs. In other cases, input and output may be performed directly to the transducers.

As an example, a DVI board [6] would be a CM device group. Its device memory is VRAM, its transducers include VDP2 and frame capture, and its DCP is VDP1.

3.2. Operating System Support

ACME uses "CM threads" to handle CM data. Primitives for thread creation/deletion, sleep locks, and spin locks are needed (these are typical features of thread interfaces such as C-threads and POSIX Pthreads [5, 11]).

Ideally, CM threads should be preemptively scheduled according to real-time requirements, both among themselves and with other system processes. This implies an OS-level (rather than a user-level) implementation of threads. CM Threads must be able to change their priorities to reflect when the next message should be completed. These can be achieved using the following set of system calls:

```
get_current_time(time); /* read high-resolution clock */
sleep_until(time);      /* put thread to sleep until given time */
change_deadline(x);     /* change deadline (scheduling priority) */
```

Ideally, these functions should be implemented in such a way that they do not make a system call

in the average case. For example, `get_current_time()` can be implemented using a hardware clock mapped into the server address space. We are currently investigating low-overhead methods for implementing the other system calls.

To achieve the needed real-time behavior, CM threads need to be able to read a high-resolution clock with low overhead. Threads need to be able to sleep until precisely a given instant of time (when a particular continuous media message might be expected to arrive) and have a particular priority on wakeup.

Because an ACME server needs performance guarantees, it must eliminate page faults by locking some pages (*e.g.*, the code and data used by CM threads) into physical memory. It will need a system call like

```
lock_pages(low, high);  /* lock pages in physical memory */
```

Such an interface, for example, has been proposed for Mach and for the POSIX Pthreads interface [5].

A second memory-related issue is how to move data between the address spaces of the server and the kernel. In a typical scenario, CM data (say, part of a rope) arrives from a network interface. It is passed to the ACME server, which divides it into its strands. Each strand is then passed back to the kernel, to be output to a DAC or video DCP. Software copying is an inefficient mechanism for data movement because it uses CPU power and bus cycles [12]. One alternative to copying is VM remapping. This technique is used in Mach [9] and in DASH [12].

3.3. Partitioning Server Functionality

In some window servers, processing of client requests is first-come-first-served (FCFS) and it suffices to implement the server as a single process. In cases where operations are time-consuming, it may be necessary to use one thread per client [2]. This prevents one client from starving another.

The real-time nature of ACME necessitates a process structure that is more complex than either of these. The CPU processing done by the ACME server can be divided into three categories:

- Non-real-time functions. This includes all standard window system operations, and ACME operations not directly manipulating CM data.
- Functions that have real-time constraints, but that can do "work-ahead" (*e.g.*, preliminary processing of data from CM connections).
- Functions that have real-time constraints, and that cannot work ahead (*e.g.*, copying data between main memory and framebuffer memory).

We propose a process structure for ACME in which the above functions are handled by separate processes (threads). This makes it possible for any activity to preempt a lower-priority activity (necessary for assuring tight deadlines) and for an activity to asychronously "work ahead" when possible (desirable for amortizing context switch overhead over more messages).

Threads that have real-time constraints (*real-time threads*, see below) communicate data using the *strand buffers* in main-memory. Strand buffers contain data corresponding to a particular LDev, in the strand format of that LDev. When two input LDevs are mapped to the same PDev and have the same strand format, they share a single strand buffer.

Our design uses the following threads (see Figure 3):

- The non-real time tasks are handled by the *window system thread.* This thread is responsible for processing all protocol requests and making appropriate calls to the ACME procedural interface (see Section 2).

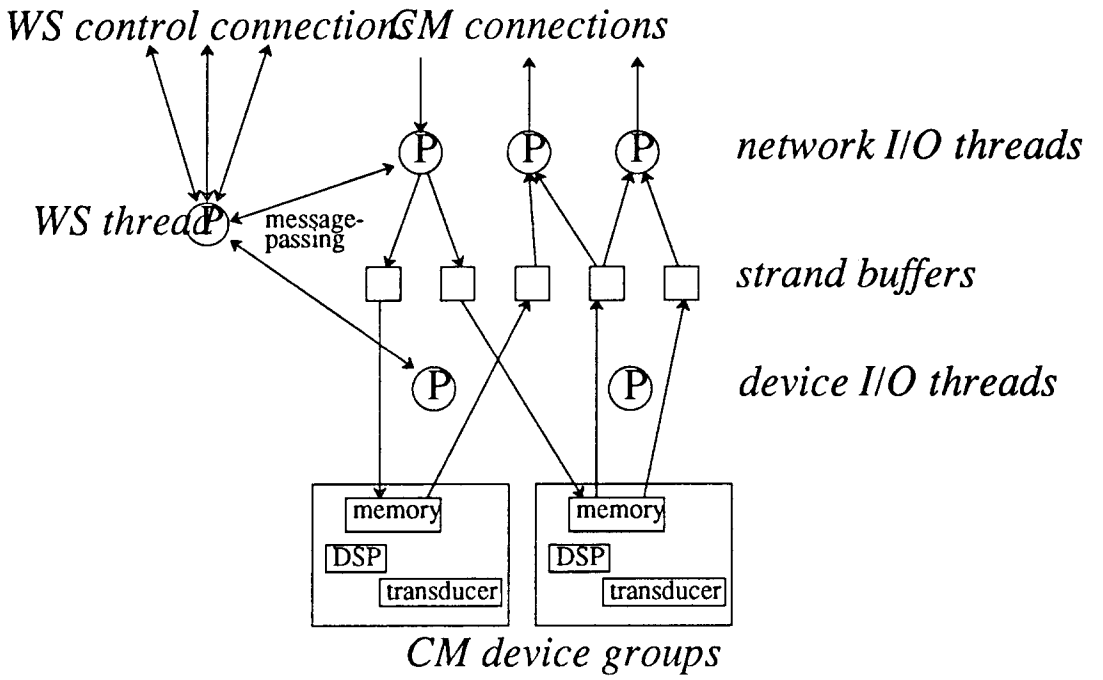

Figure 3: In addition to a *window system thread* for discrete I/O operations, the ACME server uses two types of real-time threads: *network I/O threads* handle data on CM connections, and *device I/O threads* transfer data to and from CM I/O devices. The threads communicate through data FIFOs and with message-passing.

- For each output CM connection, there is a single *network output thread* that handles data arriving on that connection. This thread separates the data into strands, does software format conversion if needed, and leaves the data in strand buffers.
- For each input CM connection, there is a single *network input thread* that gathers data from strand buffers, combines it into a rope, performs output conversion if needed, and writes the result to the CM connection.
- For each CM device group, there is a single *device I/O thread.* This thread supervises the operation of the device group: It transfers data between strand buffers and the device memory, and handles messages requesting CM device group functionality. If a coprocessor is present, the device I/O thread controls it.

The real-time threads (the network and device I/O threads) communicate with the window system thread using messages. The window system thread sends thread startup information and thread control signals (such as kill, suspend and resume) as messages. Each thread has a FIFO "mailbox" to which these messages may be sent. The `send()` operation is non-blocking. A message `receive()` may be blocking (if the thread needs to receive some information before proceeding) or non-blocking. Since the messages do not cross address space boundaries, the message passing system may be implemented using shared memory and the synchronization primitives.

3.4. Structure of Real-Time Threads

A network output thread may work ahead if data arrives ahead of schedule on the CM connection. The thread changes its deadline as it processes data. It sleeps only when waiting for more data or if it has worked too far ahead in advance. It receives blocks of rope data from the network, separates the rope into its constituent strands and performs strand format conversions. Each network input thread executes a loop of the following form:

```
for (;;) {
      check for control messages;
      read a block of rope data from connection;
      unravel rope block into its constituent strands;
      if (we have worked too far ahead or there is no more data)
            sleep until when next data block is expected;
      else
            change deadline to when the next message should be processed;
}
```

A network input thread can fall "behind schedule" up to the maximum CPU delay associated with the CM connection. It executes a loop of the following form:

```
for (;;) {
      check for control messages;
      combine strands into rope block, performing conversions as necessary;
      transmit rope block on connection;
      if (ahead of schedule)
            sleep until the time the next block will be generated;
      else
            change deadline to when the next message should be sent;
}
```

Each I/O device thread periodically outputs a block of data to the I/O device. The loop for output from a strand buffer is structured as follows:

```
for (;;) {
      check for control messages;
      move the data from strand buffers to device memory;
      start device coprocessors;
      change deadline to when the next block should be sent to the device;
      sleep until the time the next block should be output;
}
```

3.5. Message Passing

The main thread communicates with the real-time threads using message-passing. We now list the messages communicated between the main thread and the network and the device I/O threads.

The network output thread is started when the CLDev is created. When data is received over a connection, the network output thread increments a counter to update the amount of data that has arrived but not yet been displayed. Rope blocks are divided into their constituent strands and the data is placed in the strand buffer. When the counter reaches *buffer size* (see [1]) the network output thread sends a `START_STRAND` message to the device threads (see below). When the amount exceeds the *near-full point* value set by the client, the network thread sends a `STOP_DATA` message to the window system thread. The window system thread sends the corresponding event to the client. The network output thread receives a `CLEAR_BUFFER` message when the client sends a request to clear the CLDev buffer. Additionally, when the CLDev is destroyed by the client, a `KILL` message is sent to the network output thread.

The network input thread is sent a START message when the CLDev is mapped. It combines the data from the different strands and sends the resulting rope block on the CM connection. When the CLDev is unmapped, a STOP message is sent to the thread. When the CLDev is deleted, the thread received a KILL message.

The window system thread communicates with the device output thread also through messages. The messages required for this purpose are listed below.

SETUP_STRAND: This message instructs the device thread to prepare the device for I/O from a specified strand buffer. The device thread may perform some device-dependent initializations but does not start reading data from the strand buffer. In particular, the device thread may download some microcode needed to handle the strand.

PREPARE_STRAND: This message informs the device thread to prime the device pipeline (if any). The device fills the pipeline from the strand buffer but does not start displaying the data. Once the pipeline is full, no more data is read from the strand buffer.

START_STRAND: This message is issued whenever the CLDev transits to the Ready state. The effect of this message is to commence display immediately, if the CLDev or LDev is mapped. If the CLDev or LDev is not mapped, data is discarded from the strand buffer into the "bit bucket" at a rate determined by the characteristics of the physical device.

MAP_STRAND: This message is sent to the device whenever the client sends a request to map the CLDev or LDev. If the device has been sent a START_STRAND message, the effect of this message is to display data from the strand buffer. If not, this message just sets the map state of the physical device. A subsequent START_STRAND message will enable display from the strand buffer onto the physical device.

STOP_STRAND: This message prevents the device from reading the strand buffer until a subsequent start_strand message is received. This coincides with the transition of a CLDev from a Ready to a NotReady state.

UNMAP_STRAND: This message is sent to the device whenever the client sends a request to unmap the CLDev or LDev. If the device thread is currently displaying data, it switches to discarding the data into the "bit bucket". If the CLDev is in the NotReady state, this message simply records the state of the strand as being unmapped.

END_STRAND: This message terminates I/O from a strand buffer.

CHANGE_PROPERTIES: Changes in the specific attributes of an LDev are conveyed to the device thread using this message. For instance, a change in the gain level at which the particular strand is output is indicated by sending a change_properties message to the device thread in charge of the speaker.

The device input thread needs only four messages: SETUP_STRAND, MAP_STRAND, UNMAP_STRAND and CHANGE_PROPERTIES. These messages have the same functions as for the device output thread. For input LDevs, local echo is implemented as follows: If an input LDev has a non-null local echo attribute (see [1]), the LDev's strand buffer is shared by the device input and device output threads. When the input LDev is Ready, it sends a PREPARE_STRAND followed by a START_STRAND message to the device output thread. After this instant, the device output thread either discards input data (if the local echo LDev is not mapped) or echoes input (it the local echo LDev is mapped). When the input LDev is mapped, the main thread sends a MAP_STRAND message (as usual) to the input thread. When the local echo LDev is mapped, the device output thread is sent a MAP_STRAND message. Thus, local echo does not depend on the map state of the input LDev.

3.6. Implementation of LTSs

When a number of CLDevs are mapped to the same LTS and the LTS is started by the client, the ACME server must ensure that the CLDevs start displaying simultaneously. Two criteria must be met before the CLDevs which are mapped to an LTS are started: all CLDevs must have received enough data so that they do not starve[2], and all PDevs must be ready to begin display. Consider a PDev that is not ready to display immediately after it has received its first data item (an example of this is a DVI board decoding a keyframe before being able to start displaying video). If this PDev is mapped to the same LTS as a PDev that offers instant response, then simply waiting for the first criterion to be met does not guarantee synchronization.

Each LTS descriptor has, in addition to pointers to its CLDevs, a "ready count". When the client issues a StartLTS request, the ready count is set to zero, and `SETUP_STRAND` messages are sent to the device threads. Each network I/O thread keeps track of how much data has arrived on its associated connection, and increments the ready count (the `ready()` operation on the LTS) when a sufficient amount of data has been received. When an LDev is ready to display data, the device I/O thread performs a `ready()` operation on the LTS.

Whenever the LTS's ready count reaches the total number of CLDevs and associated LDevs mapped to it, the start criteria have been met. Each time a network I/O thread or a device I/O thread increments the ready count, it checks the value against this target. If the target has been reached, `START_STRAND` messages are sent to all appropriate device I/O threads, and display commences. The last thread to issue the ready operation also records the (real) start time of the LTS.

After an LTS bas started, its value increases monotonically, but not necessarily at the same rate as real time. Each of the constituent I/O devices (video system, audio converters) may have a timing base that differs slightly from the real-time clock. It may be possible to "vary the rate" of an I/O device in software, *e.g.*, by repeating or skipping video frames or audio samples, or by interpolating audio samples. In the normal case, the rate of progress of an LTS is determined by the I/O device that is hardest to vary (typically audio). The rates of the other devices are varied as needed to maintain synchronization. An LTS may also pause (*i.e.*, stop advancing) because one or more of its CLDevs starves. In addition, client requests may stop, resume, and vary the rate of an LTS (these operations and their semantics have not yet been defined).

The current value of an LTS and a list timer requests for the LTS are maintained in the LTS structure. The device I/O thread for the time-generating device as defined above periodically updates the value and handles timer requests (by making the appropriate call to the WSE).

3.7. Networks and Protocols

The ACME design requires that CM data be carried on network connections. As with CPU scheduling, this raises the issue of how to guarantee performance. This problem has two related components: 1) the performance of the underlying network; 2) the performance and characteristics of transport protocols.

Networks with nondeterministic media access protocols, such as Ethernet, cannot provide performance guarantees in general. They can be used for CM data as long as the competing load is bounded or nonexistent. Token-ring networks such as FDDI [10] can provide performance guarantees in a more flexible way. Future networks such as BISDN [3] may have a fixed set of performance levels. With either of these latter networks, the question remains of how to

[2] "Jitter" (variable transmission delay) can cause starvation even if the minimum average data rate is maintained by the CM sender.

coordinate the network with other system resources; the CM-resource model addresses this problem.

Finally, transport protocols such as TCP [8] are not ideally suited to continuous media. Their mechanisms for error recovery and flow control may be unnecessary, and when exercised may interfere with real-time performance guarantees. Researchers have developed other transport protocols the may be better-suited to CM [4, 7]. However, we believe that TCP may be sufficient in many cases, especially if other mechanisms (such as the workload limits imposed by the CM-resource model) prevent its error and flow-control mechanisms from being exercised.

4. CONCLUSION

We have sketched the implementation of an ACME server, and its integration into a network transparent window system. ACME can be implemented largely separately from the window system, but the window system has to be modified slightly to provide support for VWins. Operating system support will have to be expanded to provide page locking, multiple threads per address space, fast memory transfer between address spaces, real time deadline scheduling, and guaranteed performance network sessions. We have introduced the notion of CM device group, and have sketched a set of primitives allowing the ACME server to access such device groups in a unified way. We have proposed a process structure for the ACME server consisting of the group of processes forming the window server, a network I/O thread for each CM connection, and a device I/O thread for each CM device group; and we have shown how these processes are to communicate. And finally, we have sketched a scheme whereby output from a number of different CM sources can be temporally synchronized.

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. J. Brezak, I. Elliott and N. Meyers, "A Multi-threaded Server for X and PEX", *Proc. X Technical Conference, 1990*, Boston, MA, Jan 1990.
3. W. R. Byrne, T. A. Kilm, B. L. Nelson and M. D. Soneru, "Broadband ISDN Technology and Architecture", *IEEE Network*, Jan. 1989, 23-28.
4. G. Chesson, "XTP Design", *Proc. IFIP International Workshop on Protocols for High-Speed Networks*, Zurich, May 9-11, 1989.
5. W. Corwin, "Realtime Extension for Portable Operating Systems", *Technical Committee on Operating Systems of the IEEE Computer Society, P1003.4/D1*, Dec 1, 1989.
6. A. C. Luther, *Digital Video in the PC Environment*, McGraw-Hill, 1989.
7. G. M. Parulkar and J. S. Turner, "Towards a Framework for High Speed Communication in a Heterogeneous Networking Environment", *IEEE INFOCOM 89*, , 655-667.
8. J. Postel, "Transmission Control Protocol", *DARPA Internet RFC 793*, Sep. 1981.
9. R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky and J. Chew, "Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor Architectures", *IEEE Trans. on Computers*, Aug. 1988, 896-908.
10. F. E. Ross, "FDDI - A Tutorial", *IEEE Communications Magazine*, May 1986, 10-15.
11. A. Tevanian, R. Rashid, D. Golub, D. L. Black, E. Cooper and M. Young, "Mach Threads and the Unix Kernel: The Battle for Control", *Proceedings of the 1987 Summer USENIX Conference*, Phoenix, Arizona, June 8-12, 1987, 185-197.
12. S. Tzou and D. P. Anderson, "A Performance Evaluation of the DASH Message-Passing System", Technical Report No. UCB/CSD 88/452, Computer Science Div., EECS Dept., Univ. of Calif. at Berkeley, Nov. 1988.

APPENDIX: THE ACME PROCEDURAL INTERFACE

This Appendix describes the C procedural interface of the ACME kernel. The following primitive types are used: `PLAYER`, `LISTENER`, `VWin`, `VCAM`, `CLDEV` are ACME-defined structs. `STRAND` and `ROPE` are structs representing strand and rope types. `PDEV` is an integer representing a PDEV ID. `FRAC` is a rational number. `LDEV_ATTRS` is a struct containing the generic LDev attributes. `BITMAP` is a bitmap represented as an origin, size and 2-D byte array.

In each operation that creates an object of type *X*, the first argument is a pointer to `sizeof(X)` bytes of memory, allocated by the WSE.

1. ACME Requests

1.1. LDev Requests

```
/* create an audio output LDev */
int create_player(
      PLAYER      *player,
      PDEV        pdev,       /* must be audio output */
      STRAND      strand,     /* must be audio */
      int         volume[4]); /* volume control attribute values */

/* creates an audio input LDev */
int create_listener(
      LISTENER    *listener,
      PDEV        pdev,       /* must be audio input */
      STRAND      strand,     /* must be audio */
      int         volume[4]);

/* create a video output LDev */
int create_vwin(
      VWin        *vwin,
      PDEV        pdev,       /* must be video output */
      STRAND      strand,     /* must be video */
      int         xoffset,    /* offset in native image */
      int         yoffset,
      FRAC        xscale,     /* magnification factors */
      FRAC        yscale);
```

The WSE must call `vwin_configure_notify()` and `vwin_clipmask_notify()` on a newly created VWin before it is mapped (see below).

```
/* create a video input LDev */
int create_vcam(
      VCAM        *vcam,
      PDEV        pdev,       /* must be video input */
      STRAND      strand);    /* must be video */

/* destroy an LDEV */
int destroy_ldev(
      LDEV        *ldev);
```

If the LDev is mapped, it is first unmapped. This procedure fails if this LDev is part of a CLDev.

```
int map_ldev(
      LDEV        *ldev);
```

If LDev is of output type, starts displaying output on the associated PDev. If LDev is of input type, starts sending input on the associated connection.

```
int unmap_ldev(
```

```
        LDEV        *ldev);
```

If LDev is of output type, stops displaying output on the associated PDev. If LDev is of input type, stops sending input on the associated connection.

```
/* get current attribute values */
void get_ldev_attributes(
        LDEV        *ldev,
        LDEV_ATTRS  *attrs);

/* set attribute values */
void set_ldev_attributes(
        LDEV        *ldev,
        LDEV_ATTRS  *attrs);
```

The only attributes that can be set are the specific attributes. The generic attributes in the struct are ignored. To change only a subset of the specific attributes, call `get_ldev_attributes()`, change the values of concern, then call `set_ldev_attributes()`.

1.2. CLDev Requests

```
int create_cldev(
        CLDEV       *cldev,
        ROPE        rope,
        CONNECTION  *connection,
        LTS         *lts,
        LDEV        *ldev_list);      /* ptr to array of pointers to ldevs */
```

All LDevs must have the same input/output type, and this type must match that of the connection. This becomes the input/output type of the CLDev. If the CLDev is of output type, processing of data arriving on the connection starts immediately. Display of data does not start until the start criteria for the LTS are met. If the CLDev is of input type, data is collected from input PDevs and processed into strand buffers immediately. Data is not written to the connection until the start criteria for the LTS are met.

```
/* destroy a CLDev */
void destroy_cldev(
        CLDEV       *cldev);
```

This does not destroy the constituent LDevs.

```
/* map the LDevs for which bits are set */
int map_cldev(
        BITMAP      which);

/* unmap the LDevs for which bits are set */
void unmap_cldev(
        BITMAP      which);

void get_cldev_attributes(
        CLDEV       *cldev,
        CLDEV_ATTRS *attr);

/* the buffer of an output LDev */
void cldev_clear_buffer(
        CLDEV       *cldev);
```

Performs a clear buffer operation on the indicated CLDev. This consists of clearing the rope buffer and all private queues, stopping display or capture on all PDevs, and resetting the CLDev to the not ready state.

1.3. LTS Requests

```
/* create an LTS */
int create_lts(
      LTS          *lts);
```

All CLDevs created after this call, with their *lts* field pointing to this LTS, will be associated with this LTS. All such CLDevs must have the same input/output type, and that input/output type becomes the type of the LTS.

```
void start_lts(
      LTS          *lts);
```

Notifies ACME that it can start the LTS as soon as the start criteria are met. In the case of an output LTS, these criteria are: 1) enough data has arrived on each connection to avoid starvation, and 2) each constituent LDev must be ready to display. As soon as these criteria are met, ACME will start display to all mapped LDevs associated with the LTS. The first chunk of data displayed from each rope is guaranteed to correspond to the same value of real time.

In the case of an input LTS, the start criterion is that all LDevs must be ready to send data. The first chunk of data sent on each rope is guaranteed to correspond to the same real time value.

```
/* destroy an LTS */
void destroy_lts(
      LTS          *lts);
```

All associated CLDevs must have been destroyed previously.

1.4. CM Connection Requests

```
int register_cm_connection(
      CONNECTION   *conn,
      BOOLEAN      in_or_out,
      CONNECTION_ID handle);
```

Registers a CM connection with the ACME server, so that CM data can be sent or received on it. *In_or_out* specifies the direction of the connection. *handle* is OS-specific information which allows ACME to read or write the connection via the `read_cm_data()` and `write_cm_data()` calls to the WSE.

CM connection establishment and maintainance is the responsibility of the WSE. This request does not cause ACME to initiate a connection; it simply tells ACME that a certain transport level connection is to be used for CM.

2. Clipping Region Change Requests

ACME need not know about the window hierarchy. Instead, the WSE must notify ACME of the clipping region of each VWin. These procedures are called by the WSE to implement this protocol. The following terms are used: The *full region* of a VWin is the portion of a VWin that would be viewable if it were unobscured. The *clip mask* of a VWin is the portion of the VWin that is viewable on the screen, expressed in coordinates relative to the VWin's origin (upper-left corner). The *viewable region* of a VWin is the portion of the screen upon which the VWin is currently viewable (this is the clip mask, but expressed in screen coordinates). Let A and B denote the VWin's viewable regions before and after the change. All of the calls are synchronous in the sense that no pixels will be written to (A - B) after call returns.

```
int vwin_move_notify(
      VWin         *vwin,
      int          newx,
      int          newy);
```

Notifies ACME that a VWin has moved from its former position. In response, ACME starts

copying the video image data into the new position, as defined by *newx* and *newy*. This call does not alter the VWin's clip mask.

```
int vwin_resize_notify(
      VWin        *vwin,
      int         newwidth,
      int         newheight);
```

Notifies ACME that a VWin has changed size. In response, ACME starts copying video image data into a new region, as specified by *newwidth* and *newheight*. If *newwidth* and *newheight* imply a full region that excludes any portion of the VWin's current clip mask, the excluded portion is subtracted from the clip mask.

```
int vwin_configure_notify(
      VWin        *id,
      int         newx,
      int         newy,
      int         newwidth,
      int         newheight);
```

Notifies ACME that a VWin has changed size and/or position. In response, ACME starts copying video image data into the new viewable region defined by *newx*, *newy*, *newwidth*, *newheight*, and possibly a modified clip mask defined as follows. Let B' and A' denote the VWin's clip masks before and after an imaginary `vwin_resize_notify()` on the VWin with the given *newwidth* and *newheight*. If (B' - A') is nonempty, it is subtracted from the VWin's clip mask.

This call must be made to initialize the configuration of a VWin after it is created, and before it is mapped.

```
int vwin_clip_mask_notify(
      VWin        *vwin,
      BITMAP_OPERATION op1,
      BITMAP      *bm1,
      BITMAP_OPERATION op2,
      BITMAP      *bm2,
      BITMAP_OPERATION op3,
      BITMAP      *bm3,
      ...);
```

This procedure allows the WSE to alter a VWin's clip mask. The specified bitmaps are added to or subtracted from the VWin's clip mask, as specified by the bitmap operations. The resulting clip mask is clipped to the VWin's full region, defined by the size of the VWin. This call must be made to initialize the clip mask of a VWin after it is created, and before it is mapped. The initial clip mask for a VWin is null.

An alternative version of this call could take regions represented as collections of run length encoded "spans". The choice of version is best determined by the structure which ACME uses internally to specify clip masks. This, in turn, depends on the video processor architecture: If the video processor can access the clipping bitmap and the video image data in one memory cycle (*e.g.*, if the alpha channel is used to store the bitmap and the VSP accessed R, G, B, and alpha all in one cycle), then the performance of the bitmap method will be good. On the other hand, if extra memory cycles are required to access the bitmap, and if some internal processor registers are available, it might be better to simply read one span at a time into the processor registers, and copy the whole span with no further access to clipping information.

The following are utility calls that can be used by the WSE to simplify the task of reconfiguring VWins without overwriting graphics data on the screen.

```
freeze_vwin(
      VWin        *vwin,
```

```
        BOOLEAN     state);
```

This call with a state of FALSE suspends copying of video image data into the VWin's viewable region. No pixels are copied into the viewable region after this call returns. With a state of true, resumes copying. This call can take up to one frame time to return, and should be avoided if possible.

```
freeze_video(
        BOOLEAN     state);
```

This call with a state of FALSE suspends copying of all video image data for all VWins. No pixels are copied into the framebuffer after this call returns. With a state of true, resumes copying. This call can take up to one frame time to return. and should be avoided if possible.

3. Initialization

The following are calls for initializing and terminating ACME.

```
int acme_init();
```

This should be called once by the WSE after WS startup and before calling any other ACME procedures, or referencing any global ACME data structures.

```
void acme_done();
```

This should be called once by the WSE before WS shutdown. No ACME procedures should be called or any ACME global data structures referenced after this call.

4. Procedures Exported by the WSE

4.1. Memory Allocation Procedures

The following procedures are used by ACME for storage allocation and deallocation. They must be provided by the WSE, and made available to ACME:

```
void *acme_alloc(unsigned int n);
```

This procedure is a wrapper around whatever storage allocation mechanism is used by the WS. It returns a pointer to n bytes of newly allocated storage.

```
void acme_free(void *p);
```

A wrapper around the WS deallocation mechanism, this procedure frees the block of storage pointed to by p.

```
void *acme_realloc(void *p, unsigned int n);
```

This procedure takes a previously allocated block of storage pointed to by p, and returns a pointer to a block of storage of size n, the first k bytes of which are initialized to the value of the old block (k is the size of the old block).

4.2. Miscellaneous WSE Procedures

```
void acme_insert_ws_request(char *p, unsigned int len);
```

This procedure inserts a "fake" window system request into the request stream. The window system should process this request at some later time. ACME uses this call to implement deferred requests.

4.3. CM Connection Read/Write Procedures

```
int read_cm_data(
        CONNECTION  *cm_conn,
        void        *buf,
        unsigned    count);
```

This reads the next count bytes of CM data from the indicated connection into the buffer.

```
int write_cm_data(
      CONNECTION *cm_conn,
      void *buf,
      unsigned count);
```

This writes count bytes of CM data from the buffer to the indicated connection.

5. Event Reporting Interface

The WSE must provide a procedure for ACME to call when events are generated.

```
void report_event(
      EVENT_TYPE  type,
      void        *data);      /* type-specific additional data */

enum EVENT_TYPE {
      CLDevStopData,           /* rope buffer nearly full */
      CLDevBufferOverrun,      /* rope buffer full, data lost */
      CLDevStartData,          /* rope buffer nearly empty */
      CLDevBufferEmpty,        /* rope buffer empty, device starved */
      CLDevStartNotify,        /* report CLDev LTS start time */
      CLDevEndData,            /* end-of-data marker reached in rope */
                               /* includes LTS stop time */
      CLDevDataStamp,          /* data mark reached */
                               /* includes LTS time, opaque data */
      LTSAlarmClock,           /* LTS alarm clock reached; includes time */
      LTSStartNotify           /* LTS has started; includes real time */
}
```