Tactus: Toolkit-Level Support for Synchronized Interactive Multimedia[1]

Roger B. Dannenberg
Tom Neuendorffer
Joseph M. Newcomer
Dean Rubine
David B. Anderson
Information Technology Center
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA

Email: dannenberg@cs.cmu.edu, tpn+@andrew.cmu.edu,
newcomer@cs.cmu.edu, dandb+@andrew.cmu.edu, dba+@cs.cmu.edu

ABSTRACT

Tactus addresses problems of synchronizing and controlling various interactive continuous-time media. The Tactus system consists of two main parts. The first is a server that synchronizes the presentation of multiple media, including audio, video, graphics, and MIDI, at a workstation. The second is a set of extensions to a graphical user interface toolkit to help compute and/or control temporal streams of information and deliver them to the Tactus Server. Temporal toolkit objects schedule computation events that generate media. Computation is scheduled in advance of real time to overcome system latency, and timestamps are used to allow accurate synchronization by the server in spite of computation and transmission delays. Tactus supports precomputing branches of media streams to minimize latency in interactive applications.

KEYWORDS: interface, toolkit, multimedia, synchronization, interactive, real-time

---

[1]Published as: Dannenberg, Neuendorffer, Newcomer, and Rubine, ''Tactus: Toolkit-Level Support for Synchronized Interactive Multimedia,'' ***Multimedia Systems*** 1(2), 1993, pp 77-86.

1. Introduction

Recently, many proposals have emerged for extending graphics systems to support multimedia applications with sound, animation, and video [Wayner 91, Yager 92, Ripley 89, Xmedia 92]. Other research has been directed toward real-time transmission of multimedia data over networks [Little 91, Anderson 91, Rowe 92] and standards for the representation and exchange of multimedia data [Newcomb 91]. New capabilities for real-time interactive multimedia interfaces [Blattner 92] create new demands upon application programmers. In particular, programmers must manage concurrent processes that output continuous media. Timing, synchronization, and concurrency are among the new implementation problems.

Traditionally, object-oriented graphical interface toolkits have presented a high-level programming interface to the application programmer, hiding many details of underlying graphics systems such as X or Display Postscript. However, timing is usually overlooked in these systems. Programmers usually add animation effects by ad-hoc extensions, and synchronization at the level of milliseconds needed for lip-sync, smooth animation, and sound effects is not generally possible.

We have extended an existing toolkit with new objects, abstractions, and programming techniques for interactive multimedia. We also implemented a synchronization server that supports our toolkit extensions. Intuitively, our synchronization server does for time what a graphics server does for (image) space. In our terminology, the application program is the *client*, which calls upon the *server* to synchronize and present data. We call the combined toolkit and server the Tactus System.

The Tactus System has a number of novel and interesting features. It works over networks with unpredictable latency, and it can maintain synchronization even when data underflows occur. The techniques are largely toolkit-independent, and the Tactus Server is entirely toolkit-independent. Data is computed ahead of real time to overcome latency problems, but the initial latency of a presentation is due only to computation and bandwidth limitations. Tactus is organized so that pre-existing graphical objects acquire real-time synchronizing behavior without changes to the existing code. The Tactus System also offers a new mechanism called *cuts*, which allows for user interaction by selecting among precomputed media with very low latency.

1.1. Assumptions

Before describing Tactus, we will present some assumptions and ideas upon which it is based. First, we are interested in distributed systems, and thus we assume that there will be significant transmission delays between servers and clients. Second, we assume that multimedia output will require the merging of multiple data streams; we want more than just ‘‘canned’’ video in a window. By *data stream*, we mean any set of timed updates to an output device. Data streams include video, audio, animation, text, images, and MIDI.

The assumption that delays will be present imposes limitations on the level of interaction we can expect. Network media servers may take seconds to begin presenting video even though the presentation, once started, is continuous. We intend to support applications where media start-up delays and latency due to computation of 10 to 1000 ms are tolerable. This includes such things as multimedia documents, presentations, video mail, and visualizations. It also includes more interactive systems such as hypermedia, browsers, and instructional systems where user actions determine what to view next. Although we rule out continuous feedback systems such as video games and artificial reality, we want to support rapidly altering the presentation at discrete choice points.

1.2. Principles

To deal with transmission and computation delays, it is necessary to start sending a data stream before it is required at the presentation site. Because of variance in computation, access, and transmission delays, it is also necessary to have a certain amount of buffering in the Tactus Server at the presentation site. When multiple streams are buffered, it is necessary to synchronize their output. With Tactus (see Figure 1), all data streams are timestamped, either explicitly or implicitly, so that Tactus can determine when each component of a stream should be forwarded to a device for presentation. We assume a distributed time service that can provide client software with an accurate absolute time, with very little skew between machines [Kolstad 90], although this assumption is not critical for most applications.

Multiple presentations may be buffered at the Tactus Server. At any time, one is being presented while the others are potential responses to user choices. This avoids the latency of transmitting a presentation over the network after the user makes a choice. Transition points are marked so that smooth cuts are possible (see Section 5).

Input is handled in mirror image to output. Input events are timestamped so that they can be related back to the output that was taking place at the time of the input. Since applications compute ahead of real time, it is up to the application to deal with the skew between input and output. For example, output can be ‘‘rewound’’ to an indicated stop point, or input might be applied to future output only.

The task of synchronizing output in a distributed environment is simplified by pre-computing or pre-transmitting data streams and timestamping them. Without additional support, however, this would complicate the work of the application, which then must compute data streams in advance of real time. One way to reduce this problem is to schedule application activity by a

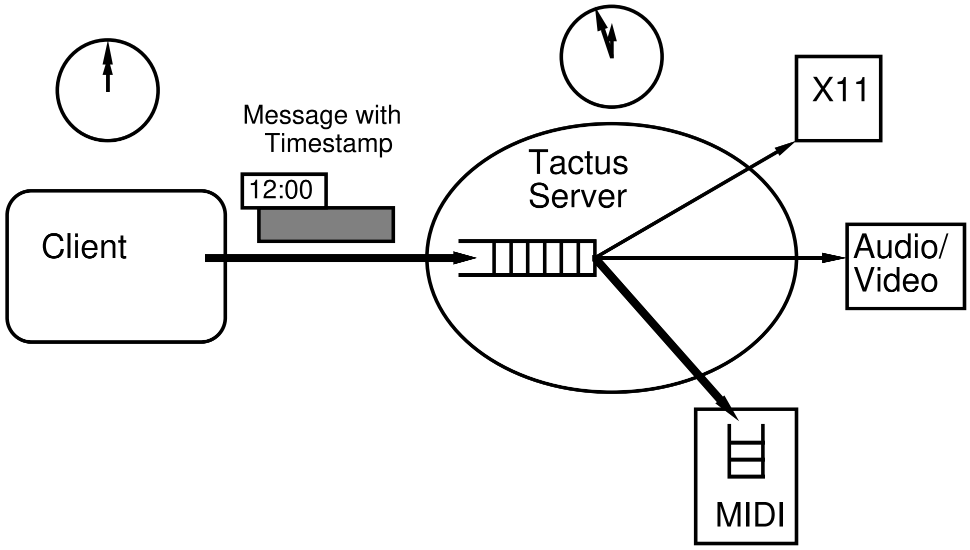

Figure 1: The Tactus System. Clients send timestamped data (heavy lines) to the server ahead of real time. Data is buffered and then delivered to various presentations devices. Some presentation devices (e.g. MIDI as shown here) may accept data early and provide further buffering and more accurate timing than can be provided by the Tactus Server. The clock on the left shows logical time as seen by the client, while the clock on the right shows real time as seen by the Tactus Server.

clock that is ahead of real time. A good analogy is that if you set your watch ahead by 5 minutes, you are more likely to show up on time for meetings.

In summary, the three most important principles of Tactus are (1) compute data streams in advance of real (presentation) time, (2) use a server at the presentation site to buffer and synchronize data streams, and (3) buffer all possible responses to user choices to minimize response times. Buffering data at the presentation site can greatly increase the timing accuracy with which data is presented.

1.3. Previous Work

Few of these *principles* are original, but their integration and application are new. Tactus was inspired by David Anderson and Ron Kuivila's work on event buffering for computer music systems [Anderson 86, Anderson 90a]. This work is in turn related to discrete-event simulation. Later, Anderson applied these ideas to distributed multimedia [Anderson 90b], but not to interface toolkits. Rowe and Smith [Rowe 92] developed a system similar to Tactus. Their system implemented sophisticated network protocols for continuous media, but their client toolkit does not offer the features implemented in Tactus. Active objects (see Section 3.1) have long been used for animation [Kahn 79] and music [Cointe 84] systems, but have only recently gained attention in multimedia circles [Gibbs 91]. The Cognitive Coprocessor [Robertson 89] manages computational latency by adapting animation rate and detail according to principles of human perception. This approach could be used in conjunction with the Tactus architecture.

To our knowledge, we are the first to extend an object-oriented application toolkit with support for managing latency through precomputation and event buffering. CD-ROM based video systems have used buffering of images at choice points to allow for seek time. Our work focuses more on the implications of all these techniques for application toolkits.

Recently, many commercial multimedia systems have been introduced, including Apple's Quicktime [Wayner 91], IBM's MMPM/2 [IBM 92], Microsoft's MPC [Yager 92], and Dec's XMedia [Xmedia 92]. These systems emphasize storage, playback, and scalability. HyTime [Newcomb 91] provides a standard representation for hypermedia but no implementation is specified. These systems could benefit from the synchronization and latency management techniques we propose, and our work suggests how a graphical interface toolkit might be extended to take advantage of commercial multimedia software.

Although this area is relatively new, efforts to establish standards are already underway. The Interactive Multimedia Association has published an RFT [IMA 93] for multimedia system services which sketches out an architecture for interactive, distributed multimedia and gives some indication of how synchronization and timing services like those in Tactus may fit into future industry standards.

2. The Tactus Server

The Tactus Server provides buffering, resource management, timing, and synchronization to its clients. It can be thought of as analogous to a display server, such as X11, with two important differences. First, the Tactus Server is a real-time server; all data is buffered until its appropriate output time. This supports the smooth uninterrupted flow of continuous-time media to hardware presentation devices. Input data is also timestamped. Second, the Tactus Server manages multiple media, including audio, video, and MIDI in addition to graphics. It is the job of the Tactus Server to present each stream according to a specified time offset.

An important point here is that all the latencies in the system are handled by the Tactus Server in a fashion that is largely transparent to the application. The server handles this adaptively, being able to compensate for transient worst case delays and to take advantage of any transient improvements. A drawback is that the server must be extended to handle new media and devices, but in practice, only a small amount of code is needed to support a new device. For the most part, Tactus merely forwards data to indicated devices at the correct time.

The server is responsible for the timely delivery of media to devices, so it places certain real-time requirements on the operating system. First, the server should have an accurate source of time. Not only should the server clock be synchronized with the client(s), but corrections for drift should not impact multimedia presentations. For example, if the local clock is set ahead by 100ms, a noticeable jump may occur in the presentation. Therefore, the local clock should advance smoothly with respect to allowable media jitter. It should be synchronized to the client within a small fraction of the time that media is buffered in the server. In the future, low-cost atomic clocks [Creedy 93] may eliminate clock drift and synchronization problems.

A second requirement for the operating system is that the server must wake up at times indicated by timestamps and deliver output to devices with low latency. An interesting property of the server is that wakeup times are aperiodic, but known in advance, offering the possibility of online pre-scheduling. The upper bound on latency for some media, such as MIDI, is in the millisecond range, which may not be possible in some systems. An alternative used by the Tactus Server is to forward data to devices ahead of time (just as clients forward data to the server ahead of time), and let the device driver provide accurately timed output. Device drivers can often provide more accurate timing by processing at the interrupt level, by using a co-processor, by using DMA, or by running within the operating system kernel. However, this approach adds complexity and reduces the fine-grain control of devices, so it should be used only when necessary. A hybrid approach is sometimes possible where Tactus sends control information only. Devices then access precomputed media data directly from storage.

If the final timing is left to device drivers, there must be a feedback path to notify the Tactus Server when timing problems arise. For example, if a graphics pipeline falls out of real time, the Tactus Server must be informed so that other media output can be delayed as well. More details on device control will follow in Section 4.4.

3. The Tactus Toolkit Extensions

The Tactus Toolkit Extensions augment a graphical user interface toolkit such as ATK [Palay 88] or Interviews [Linton 89]. The Tactus Server could be used without the toolkit extensions, but this would require the clients to compute data in advance of real time, implement various protocols (described in Section 4), and interleave computation for various streams. The toolkit simplifies these programming tasks. Our toolkit extensions to ATK include clock objects for scheduling and dispatching messages, active objects that receive wake-up messages and compute media, and stream objects that manage Tactus Server connections and timestamping (see Figure

2). The class hierarchy is illustrated in Figure 3.

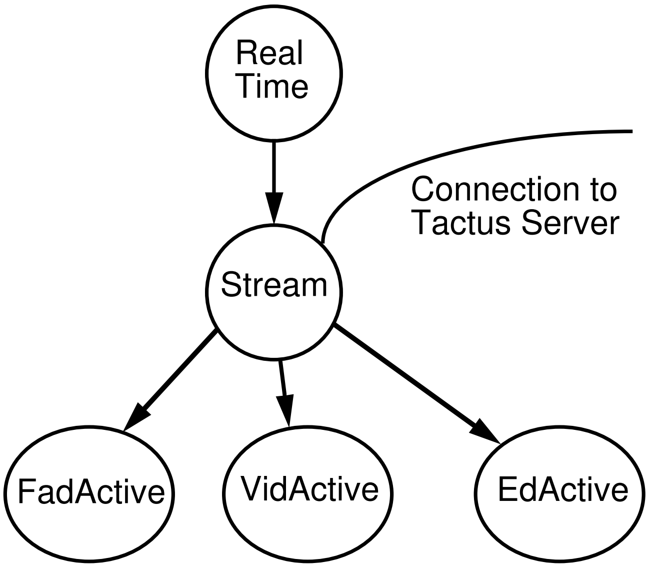

Figure 2: A Clock Tree. Objects, including clocks, request a wake-up message from their parent in the clock tree. **`RealTime`** is at the root of the tree and interfaces with the operating system timing facilities. **`Stream`** is a subclass of **`Clock`** and manages connections to the Tactus Server. The leaves of the tree are subclasses of **`Active`** which produce and control multimedia data (see Section 6). **`Kick`** messages flow in the direction of the arrows, while **`RequestKickAt`** messages are sent in the opposite direction.

3.1. Active Objects

Active objects form the base class for all objects that handle real-time events and manage continuous time media in the Tactus extensions. Each active object uses a clock object (set via the **`UseClock`** method) to tell time and to request wake-up calls. The **`RequestKick`** method schedules the active object to be awakened at some future time by sending **`RequestKickAt`** to the object's clock. The object's **`Kick`** method will then be called by the object's clock at the requested time.

Active objects are intended to take the place of light-weight processes, and often perform tasks over extended periods of time. This is accomplished by having each execution of the **`Kick`** method request a future **`Kick`**. For example, we have a software video active object whose **`Kick`** method is roughly as follows:

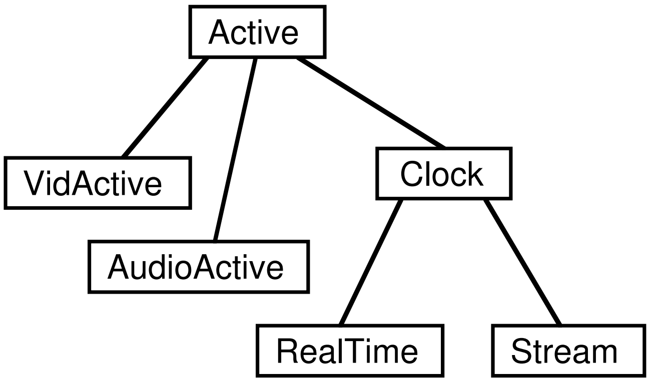

Figure 3: Active Object Class Hierarchy. **`Active`** objects, which answer **`Kick`** (to perform the action) are specialized to create media presenters such as **`VidActive`** and **`AudioActive`**. **`Clock`** objects answer **`RequestKickAt`** (to schedule a kick) and send **`Kick`** messages to their children at the appropriate times. **`RealTime`** and **`Stream`** are specializations of **`Clock`**.

```
Kick(self, time) {
    if (read(InputFile, Buffer, ImageSize)) {
        XPutImage(Buffer, ...);
        RequestKick(self, time + InterFrameTime);
    }
}
```

In this example, **`XPutImage`** is the standard X11 call to display a raster image, but it is implemented by a special library that adds a timestamp and sends the message to the Tactus Server. The Tactus Server is then responsible for forwarding the message to the real X server at the proper time[2]. Note how **`RequestKick`** is used after each video frame to schedule the next one. During a video presentation, the computations of many other active objects may be interleaved with this one.

Rather than develop a special timed graphics library, we support the ordinary **`xlib`** calls with a substitute library **`txlib`**, which makes a connection to the Tactus Server rather than the X11 server. The **`txlib`** sends the timestamp along with each X11 message. The advantage of this

[2]Since we haven't modified the X server to process timestamped events, we have found it helpful to run X at a higher priority in order to help ensure that the display is updated soon after the Tactus Server sends the graphics commands.

approach is that pre-existing graphics code can be used without modification. This point will be further explained in Sections 3.3 and 6.3. While **`txlib`** handles output for X11, similar libraries are used for audio and other media.

3.2. Clock Objects

Clocks are a subclass of **`Active`**. Each clock object keeps track of all the active objects (children) that have attached themselves via the **`UseClock`** method. Since clocks are active objects, they too can be attached to other clocks. Clocks are useful not only for their wake-up service but also because they manage mappings from one time system to another. Mappings are linear transformations, meaning that a clock can shift and stretch time as seen by its children. When a change in the mapping of time occurs, children of the clock are notified (whether or not they are waiting for a **`Kick`**). We call the time seen by children of Clocks *logical time*, as opposed to the *real time*. Logical time allows active objects to compute in ‘‘natural’’ time coordinates. Meanwhile, clocks can be adjusted to achieve ‘‘fast forward’’, ‘‘rewind’’, ‘‘pause’’, and ‘‘continue’’ effects.

The basic duty of a clock is to keep track of the times at which to kick child objects and to issue **`Kick`** messages at the appropriate times. A clock keeps a list of children sorted by their local requested kick times. The clock maps the earliest local wakeup time into parent time and issues its own **`RequestKick`** so that it will be kicked by its parent. When the clock is kicked, it maps from parent time to local time and kicks the appropriate child.

Clocks form a ‘‘clock tree’’ whose leaves are active objects, whose internal nodes are clocks, and whose root is a special subclass of clock called **`RealTime`**. A **`RealTime`** object serves as the true source of time for the entire clock tree. It should be noted that the clock tree is entirely independent of the graphical view tree typically found in graphical user interfaces [Cox 87].

3.3. Stream Objects

Stream objects are a subclass of **`Clock`**. In addition to scheduling and kicking child objects, stream objects communicate with the Tactus Server and establish timestamps for Tactus messages. Stream objects also schedule their children ahead of real time by the worst-case system delay called *Latency*, a number which is presently determined empirically.

Recall that the Tactus Server expects all messages to have timestamps which serve as the basis for synchronization of multiple media. It might seem logical to use the kick times of active objects, but because kick times are the composition of perhaps several mappings at different levels for the clock tree, the active object kick time may have no simple relationship to real time or to the kick times of other active objects.

Rather than use active object kick times, timestamps are based on the idealized real time of the kick, that is, the requested kick time mapped to real time. Before a kick, the clock tree is inactive. When the kick time arrives, a **`Kick`** message is propagated from the **`RealTime`** object through the tree to an active object at a leaf of the tree. On the path from root to leaf, exactly one stream object is kicked. The stream sets a globally accessible timestamp and stream identifier before propagating the **`Kick`** message. If the active object performs an output action, the output function called accesses the timestamp and stream identifier in order to compose a message for the Tactus Server. In this way, timestamps are implicitly added to client output.

Together, these classes and their specializations serve to insulate the application programmer from the detailed protocols necessary to send streams of data to the Tactus Server. The extensions do such a fine job of hiding details that existing programs can use Tactus without modification. (Tactus libraries are linked dynamically.) Although this provides no benefits to existing applications, it means that *existing application components can be given real-time synchronization capabilities*. For example, objects that formerly displayed text or images can now be called upon to deliver output synchronously with other media.

4. The Tactus System

As described in the introduction, the Tactus System consists of a Tactus Server as well as a set of extensions to an object oriented toolkit. In this section, we will describe how the two work together.

4.1. Steady State Media Delivery

Steady-state on the client side consists of active objects waiting for wake-up messages. At each wake-up, an object computes data such as a packet of audio or a frame of animation and sends it to the Tactus Server. The wake-up message is scheduled by a stream object that forces the computation to happen ahead of real-time. When no more computation is pending, the stream object computes when the next wake-up will occur and sends a null message to the Tactus Server with that timestamp. This tells the Server not to expect more messages until that time.

As discussed earlier, Tactus may deliver the data to the presentation device slightly ahead of time, relying on the hardware or device driver to delay the presentation until a given timestamp. For example, our MIDI driver maintains buffers of timestamped packets of MIDI data and outputs data at the designated time. In contrast, X11 (our ‘‘graphics device driver’’) has no buffering or timestamping capability yet, so Tactus provides all timing control for graphics. These differences are invisible to clients.

Time is used to regulate the flow of data from clients to Tactus, thus alleviating the need for explicit flow-control messages. The client simply produces ‘‘one second of data per second’’ and sends it to Tactus. When the client is behind, it computes as fast as possible in order to catch up. If the client falls too far behind, the Tactus Server buffers will underflow and a recovery mechanism must be invoked. (See below.)

4.2. Stream Start-Up

We anticipate that the worst-case delay from client to device will be quite large (perhaps seconds). This is too large to be acceptable for the normal stream start-up time. Tactus clients typically will start streams with the goal of delivering media to the user as soon as possible. Therefore, the stream object advances logical time, causing the client to run compute-bound until it catches up. To further facilitate rapid start-up, each device has a *minimum* amount of buffering (measured in seconds) required before it can start, and the Tactus Server rather than the client determines when to start a presentation.

Presentations start with the stream in the *starting* phase. The Tactus Server waits for data to be buffered. During the low-water phase, the data is generated and sent as fast as possible from the client to the Server. Packets are accumulated until a *low-water mark* is met *for each device opened in the starting stream* (the low-water mark may differ between devices). When the low-water criteria are met, the Server starts delivering data to devices, and we are in the *running* phase. At the phase transition, a message is sent to the client(s) indicating how much time elapsed in the starting phase. This time is called *Glitches*.

4.3. Synchronization

Synchronization is achieved by attaching timestamps to data. Temporal data are thus synchronized to a clock (as opposed to synchronizing directly to other data, such as video frames). Multiple streams may be grouped for synchronized presentations. It is up to device drivers to either present data at specified times or to report timing failures back to the Tactus Server. Timing failures in devices are treated just like underflow, which is described next.

4.4. Underflow

An underflow is caused by the stream buffer running out of data. More precisely, underflow occurs when it is time to dispatch a data packet at time T, but there is no packet containing data at a time greater than T. Since data arrives in time order, a timestamp greater than T is desired because it indicates that all data for time T has arrived. It is the current policy of Tactus to halt all media presentation at time T until all media for time T can be updated, but we believe other policies could and should be supported as well [Anderson 90b].

If data has been sent ahead to devices, Tactus can reset the logical time in the device (setting it back) so that the device immediately pauses. If this is not possible, the stop and restart may be somewhat staggered in time across devices. Eventually, all device output is stopped and a low-water phase is entered. This phase resembles the start-up protocol described above, and the same steps are taken.

No immediate feedback to the client is necessary upon underflow (presumably, the client is already compute-bound trying to catch up). When Tactus resumes data output, it sends a message to the client indicating the amount by which the presentation was delayed. This information can be used to control the total delay between computation and presentation. The default behavior is for the client to keep a constant presentation latency; if the Tactus Server stops the presentation for 2 seconds, then the client holds off on computation for 2 seconds as well.

Generally, this protocol takes place only at the stream level, and the clocks and active objects beneath the stream remain oblivious to the time shifts. On the other hand, active objects can attempt to avoid underflow by noticing or predicting when computation falls too far behind real time. For example, our animation object drops frames, maintaining a constant number of frames per second, when the logical time rate (playback speed) is increased.

4.5. Device Drivers

The Tactus has some interesting implications for presentation subsystems and device drivers. Devices should start continuous media output at precisely stated times and should synchronize data to the system clock. Devices should also monitor the delivery of output and report exceptions back to Tactus.

A condition similar to underflow arises when a device fails to meet its expected timing requirements. For example, a CD-ROM player might fail to seek within an allocated time. In this case, the device should recognize that timely delivery of media has (or will) fail, and the Tactus Server should be notified. The server treats this condition as a form of underflow and enters the a low-water phase. The running phase is reentered when the failing device reports that it is ready to continue, e.g. the seek completes. Another example would be a graphics accelerator that is unable to complete a frame of animation in time.

5. Cuts

Because of various latencies, multimedia systems are often unable to respond to input without obvious ‘‘glitches’’ where, for example, the video image is lost, digital audio pops, and graphics are partially redrawn. This usually happens because there is a time delay between taking down one stream and starting up another. These annoying artifacts could be hidden if the new stream could be started before the the old one is stopped. Tactus supports this model, and a switch from one stream to another is called a *cut.*

In Tactus terminology, a *cut* is made from a primary stream to a secondary stream. To minimize latency, cuts are performed by the Tactus Server on behalf of its client. The client requests a cut, but the request may or may not be honored, depending upon whether the secondary stream is ready to run.

There are two attributes that describe a cut (see Figure 4). The first determines whether a cut may be taken at any point in time or only at certain time points, and the second describes whether the cut is made to the beginning of a secondary stream or to the current time.

Cuts must be anticipated by the application. Since the application runs ahead of the real presentation time, it will naturally come to choice points before the user has a chance to make a choice. For example, a driving simulator application will generate graphics or video for an intersection before knowing whether the user will say ‘‘turn left’’ or not. While computing ahead, the application will create a cut object to arbitrate between the current (primary) stream and a new ‘‘turn left’’ (secondary) stream.

Within the Tactus Server, the secondary stream will perform a normal stream start-up. If the user requests ‘‘turn left’’, the application[3] sends a cut message with a timestamp to the Server. If the message arrives before the time indicated by its timestamp, and if the secondary stream is ready to run, then Tactus switches to the secondary stream at the designated time. A message is returned to the application indicating success or failure. If the cut was a success, then the objects generating the no-longer useful primary stream will be freed.

6. An Example Application

It is now time to see how clocks, streams, active objects, and the Tactus server work together to produce a synchronized multimedia presentation. We will describe an application we have actually built: a time-line editor for sequencing video and animation.

---

[3]User cut requests are processed by the application, not by the Tactus Server; this requires a round-trip message to the application, but keeps the input-processing model uniform.

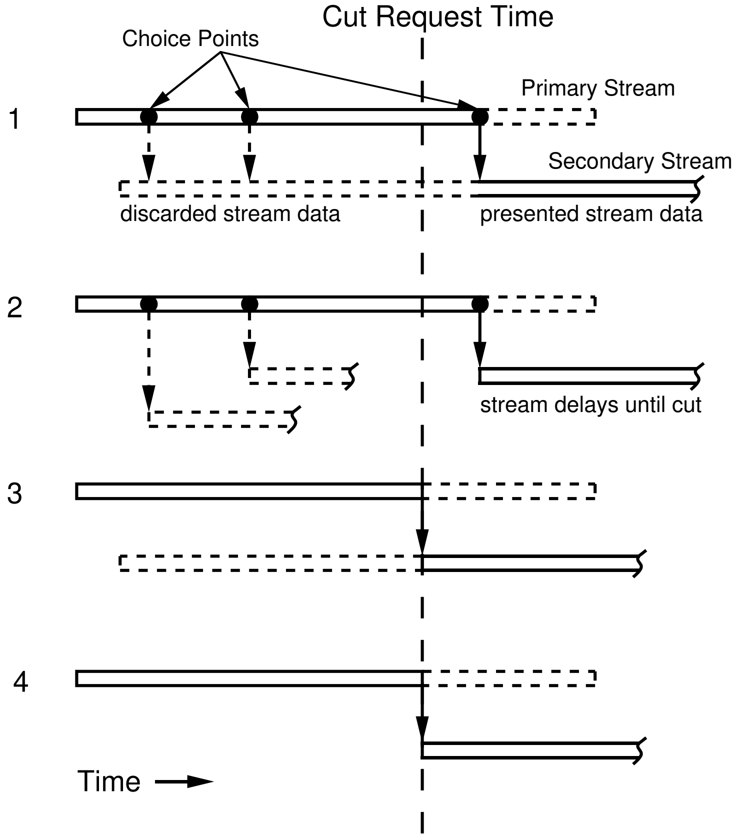

Figure 4: There are four (4) types of cuts. The top two cuts are restricted to discrete time points, whereas the lower two can take place at any time. The first and third cuts here cut to a stream already in progress while the second and fourth types cut to the beginning of a stream.

6.1. The Editor

As far as this discussion is concerned, the function of the editor (see Figure 5) is merely to produce a data structure consisting of a list of animations to run and video segments to show. We will call this data structure the *cue sheet*. An animation sequence represented by the editor consists of a file name, a starting frame, an ending frame, and a duration. An active object of class **`FadActive`** takes these parameters and generates a sequence of display updates showing the sequence of frames and some number of interpolated frames, depending upon the duration.

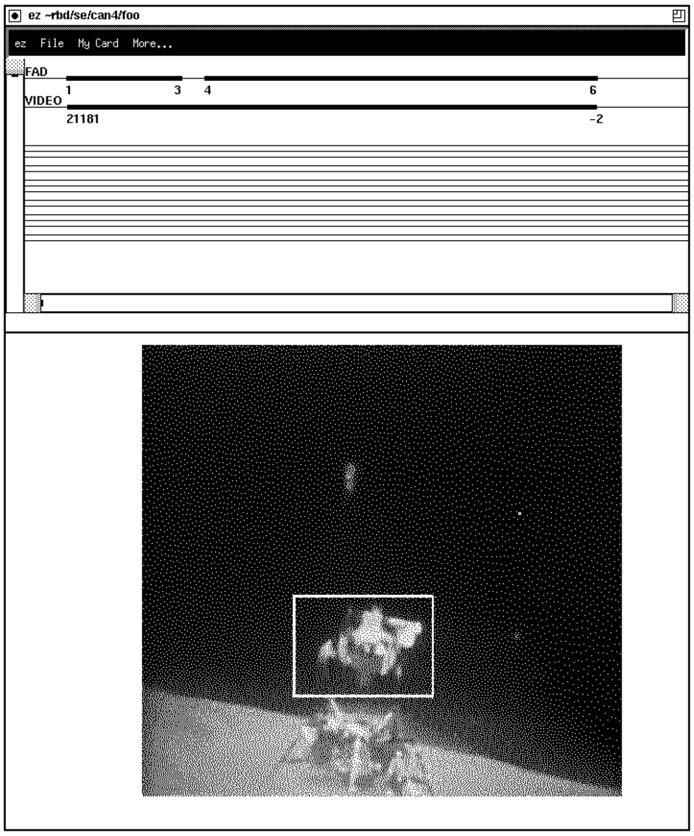

Figure 5: The Editor Application. The top window is a time-line for animation, video, and music. The presentation below shows a video image and an animated graphical overlay, in this case highlighting the Lunar Excursion Module as it lifts off from the moon.

Similarly, a video segment is represented by a starting frame, an ending frame, and a duration. An object of class **`VidActive`** generates control commands for a laser videodisc player (an all-digital video object has also been implemented) to generate the appropriate sequence of video frames.

6.2. Active Objects and the Clock Tree

The structure of the application was shown in Figure 2. An editor creates three active objects: **`FadActive`** for graphical animation control, **`VidActive`** for video control, and **`EditActive`** for sequencing through the editor data structure. All three of these use a stream object as their clock, and the stream object uses **`RealTime`** for its clock.

6.3. Generating a Timestamped Message

Let us trace how the first message of a presentation is generated. When the user selects the Start menu item, the editor sets the stream to logical time 0. This happens at real time T_{start}. The editor also sends the **`EditActive`** object a message which causes it to request a kick at logical time T_{cue}, the time of the first cue on the editor's cue sheet. For now, let us also assume that the first item is well beyond the *Latency*. Thus, a kick request will be registered with the **`RealTime`** object at some time in the future, after which the Start command completes, and the user interface becomes idle.

To summarize so far, the editor has requested that the stream kick it to produce the first action at logical time T_{cue}. The stream will actually kick the editor's active object ahead of time by *Latency*, a programmer-specified time advance[4]. Also, the kick is delayed by *Glitches*, the total time by which presentation has been delayed in the low-water phase. This variable still has its initial value of 0.

Figure 6 illustrates timing in Tactus. The middle line of the figure shows the real time of kicks. At $T_{start} + T_{cue} - Latency + Glitches$, the realtime clock will send a **`Kick`** message to the stream. Figure 7 shows the sequence of calls that lead from the realtime clock to a timestamped message. The stream object constructs a timestamp of $T_{start} + T_{cue}$ and kicks **`EditActive`**, which in turn sends a start message to the **`FadActive`** object. **`EditActive`** also schedules itself for the next cue in the cue sheet.

The **`FadActive`** object initializes its state and prepares to generate a sequence of animated drawings and calls upon a **`View`** object to update the screen with the first image of the animation[5] and returns. The kick to **`EditActive`** returns, and a new kick to **`FadActive`** immediately follows.). The **`View`** object calls upon its associated **`Graphic`** object to perform graphical operations, and the **`Graphic`** object makes standard calls to **`xlib`**, the X11 client interface library.

When the update is finished, the **`FadActive`** object schedules itself for its next update and returns. At this point, the **`FadActive`** object is scheduled to wake up on the next animation frame, and **`EditActive`** will wake up to send the next cue from the cue sheet. Whenever the stream reaches a state where it will actually suspend until there is another kick from **`RealTime`**, the stream computes the stream time at which it will wake up and sends that time to the Tactus

---

[4]In the future, time advance may be estimated automatically on the basis of system and network loads.

[5]This is a slightly simplified description. In reality, the **`FadActive`** does not know who sent the start message and cannot assume that the stream has properly computed a timestamp. As a precaution, it schedules itself for a kick at the current logical time (according to its clock).

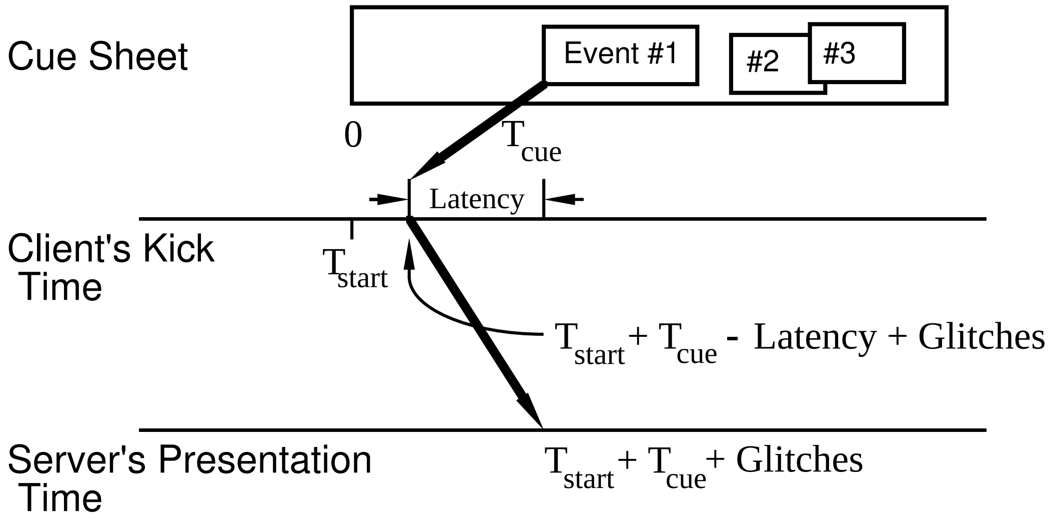

Figure 6: Timing. The top line shows times as specified in a cue sheet. The middle shows the timing of **`kick`** messages, and the bottom shows presentation times. The cue sheet is effectively shifted forward in time from zero to T_{start}. All kicks are advanced in time by the amount *Latency* so that output can be delivered to the server and buffered. The presentation then corresponds exactly to the time-shifted cue sheet. If underflow occurs, Glitches will become non-zero, and both the client computation and the server presentation will move later in time.

Server. This tells the server how far ahead of realtime the computation has progressed. This information (or rather, the lack of it) is in turn used to detect underflow.

On the server side, messages arrive with timestamps and commands for X11 and a videodisc player. The messages are placed in a buffer until the dispatch time arrives. The dispatch time is:

$$T_{dispatch} = Timestamp + Glitches$$

where *Glitches* measures the total elapsed time in the low-water phase. The bottom line in Figure 6 shows the server presentation time. Notice from this example, that the active objects can all operate in the relative time of the cue sheet; for example, presentations can start at logical time zero.

Also notice that because timestamps are passed directly from the stream to txlib, no modification to complex graphical objects (the view object in Figure 7) is needed. Even the full ATK text editor runs under Tactus without modifications. If text were to be scrolled by an active object, the scrolling would be synchronized with other stream elements. Any X application,

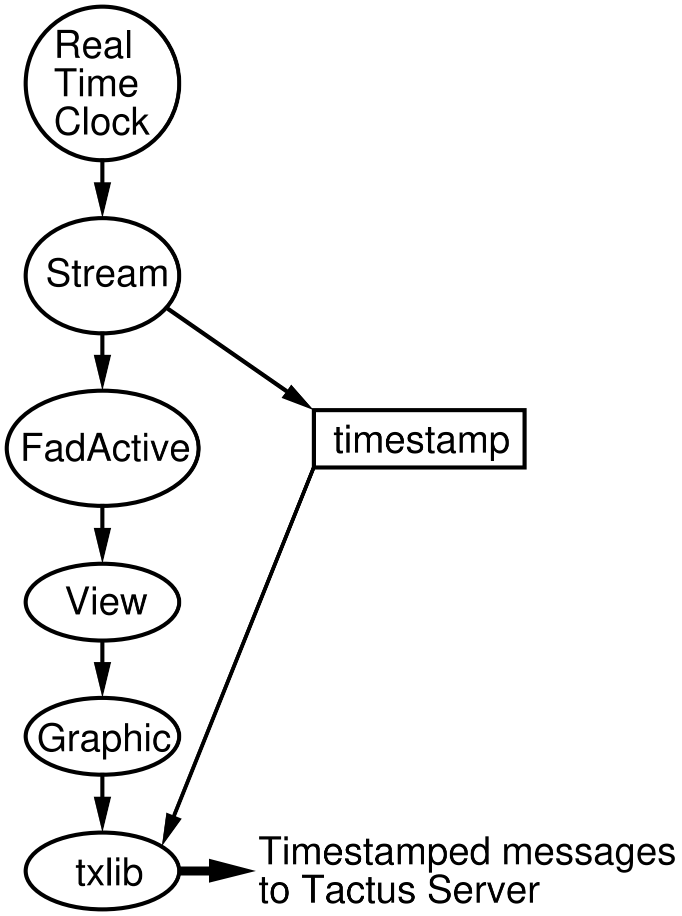

Figure 7: Creating timestamped messages. Scheduled computations are initiated by **`RealTime`**. The **`Stream`** computes a timestamp for later use by **`txlib`**. Control then passes to an **`Active`** object which produces output via a **`View`** object which in turn calls a **`Graphics`** object. X-window calls from **`Graphic`** objects are handled by **`txlib`**, which adds timestamps and sends messages to the Tactus Server.

when linked with the Tactus library, should also run under Tactus, and it is possible to modify other toolkits to use Tactus synchronization.

6.4. Adding an Animated Cursor

Suppose we want to modify the timeline editor to display a vertical cursor that travels across the timeline during playback. The change is trivial since during playback, any graphics commands are automatically synchronized with other media, and the active object architecture supports interleaving the cursor computation with other output. The changes required are: Subclass **`EditActive`** to create a **`CursorActive`** object, and add it to the clock tree as a child of the stream. The **`CursorActive`** object will automatically be notified when the stream starts and stops. When the stream starts, use the **`RequestKick`** and **`Kick`** messages to wake up every 50 ms or so and redraw the cursor according to the current logical time.

7. Current Status

Tactus currently exists as a working prototype on Unix workstations; it supports synchronized digital video, digital audio, animation, and the full range of ATK graphics and text objects. The Tactus Server runs directly on the Mach 3.0 Microkernel [Accetta 86], and another version runs as an ordinary user process under AIX. Typically, both the client and server run on one machine using a choice of shared memory, Mach IPC, or Unix socket interfaces. We are studying performance issues and porting Tactus to Real-Time Mach [Tokuda 90] should provide even better predictability. Even though applications remain as Unix processes, the latency management of Tactus allows them to generate accurately timed output.

8. Summary and Conclusions

Tactus is a system for synchronizing multiple digital media in a distributed environment where latency is an important factor. In Tactus,

- Data is computed ahead of real time,
- An object-oriented system is used to schedule computation and compute timestamps,
- The data is delivered to a synchronization server and buffered, and
- The data is dispatched to presentation devices according to timestamps.
- Latency management is carefully hidden from application programs, which benefit from a simple-to-use interface.

Important considerations in the design of Tactus have been:

- The system should support interactivity by helping clients compute multimedia information,
- The server should handle synchronization at the point of presentation,
- Both discrete events and continuous media should be supported, and
- Existing graphics support libraries should be usable with a minimal amount of

change.

Consequently, a large effort has gone into designing the client side of Tactus as well as the server. This has paid off by simplifying the task of writing multimedia applications.

There are problems Tactus does not address. In particular, we have attempted to mask rather than solve network problems. We assume that enough network bandwidth is available on average for multimedia presentations. The amount of precomputation and buffering can be tailored to actual network parameters. Even if communication is not a problem, Tactus can mask computational and storage latency.

Tactus also does not address issues of resource allocation. We assume that adequate bandwidth and processor resources are available on average. If this is not the case, no other approach will be successful. Ideally, Tactus should be used in conjunction with network protocols and operating systems that can make some resource guarantees. It is up to the application programmer to specify requirements or reserve resources. One area for future research is to examine how secondary streams for cuts should be scheduled when resources are limited.

Tactus has important implications for the design of multimedia systems. Few device drivers or devices support timestamped data, yet this seems to be a useful technique for solving current synchronization problems. For example, MIDI data should be transmitted within about 1 ms of the specified time. This may be difficult even for the Tactus Server. Thus, we adopt a multi-level approach. The Tactus Server buffers the high-latency (100-1000 ms) connection from the client to produce synchronized low-latency (e.g. 10 ms) data for devices. In the case of MIDI, Tactus dispatches data slightly ahead of real-time to enable the very low-latency (less than 1 ms) device driver to provide the final timing.

One could argue that if all device drivers supported timing, the Tactus Server would not be necessary. However, the Tactus Server performs other important functions. It performs local synchronization failure detection and recovery across multiple media and devices. (This could also be performed by the client, but only with many round-trip messages to a potentially overloaded client.) In the case of a remote client, the Tactus Server provides the logic and buffers for cuts. The Tactus Server also supports the synchronization of multiple distributed clients. Finally, the server is a logical place to provide for resource management.

Designers of network time protocols should keep in mind that abruptly setting clocks ahead or behind can cause problems for multimedia and other real-time systems. It seems that network time systems are designed to minimize clock skew at the expense of jitter caused by local clock adjustment. For multimedia synchronization, this jitter should also be minimized.

Tactus uses global clocks for flow control between client and server. Clients know when data will be output, so they send data just before the data is needed. No explicit flow control messages are necessary.

Media delivery subsystems should be designed with facilities for external control and synchronization, whereas most current designs are closed and self-regulating. For example, a video playback sub-system can be integrated with Tactus if it accepts absolute start times, has the potential to pause when other media underflow, and can report to Tactus when a pause due to underflow is about to occur. In contrast, the practice of synchronizing digital video to a local audio device makes it impossible to synchronize to external systems or other channels of audio.

Finally, Tactus raises some interesting and unaddressed scheduling problems [Dannenberg 89] not solved by traditional periodic scheduling models. Tactus events are aperiodic, but the precise time and content of a Tactus event dispatch is known significantly in advance.

9. Acknowledgments

A shorter version of this paper appeared in the Third International Workshop on Network and Operating System Support for Digital Audio and Video, IEEE Computer and Communication Societies, 1992. The work was entirely sponsored by the IBM Corporation. We would also like to thank Jim Zelenka, and Kevin Goldsmith for implementation assistance, and Carol Krowitz and Joy Banks for work on this manuscript.

References

[Accetta 86] Accetta, M., R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Proc. of Summer Usenix*. Usenix, July, 1986.

[Anderson 86] Anderson, D. P. and R. Kuivila. Accurately Timed Generation of Discrete Musical Events. *Computer Music Journal* 10(3):48-56, Fall, 1986.

[Anderson 90a] D. P. Anderson and R. Kuivila. A System for Computer Music Performance. *ACM Transactions on Computer Systems* 8(1):56-82, February, 1990.

[Anderson 90b] D. P. Anderson and R. Govindan and G. Homsy. *Abstractions for Continuous Media in a Network Window System*. Technical Report UCB/CSD 90/596, Computer Science Division (EECS), U.C. at Berkeley, 1990.

[Anderson 91] D. P. Anderson and G. Homsy. A Continuous Media I/O Server and Its Synchronization Mechanism. *Computer* :51-57, October, 1991.

[Blattner 92] M. M. Blattner and R. B. Dannenberg (editor). *Multimedia Interface Design.* ACM Press, 1992.

[Cointe 84] P. Cointe and X. Rodet. Formes: an Object and Time Oriented System for Music Composition and Synthesis. In *1984 ACM Symposium on LISP and Functional Programming*, pages 85-95. ACM, New York, 1984.

[Cox 87] B. J. Cox. *Object-Oriented Programming: an evolutionary approach.* Addison-Wesley, Reading, Mass., 1987.

[Creedy 93] S. Creedy. Time is right for atomic ticker by Westinghouse. *Pittsburgh Post-Gazette* :1-2, April 24, 1993.

[Dannenberg 89] Dannenberg, R. B. Real-Time Scheduling and Computer Accompaniment. *System Development Foundation Benchmark Series. Current Directions in Computer Music Research.* In Mathews, M. V. and J. R. Pierce, MIT Press, 1989, pages 225-262.

[Gibbs 91] S. Gibbs. Composite Multimedia and Active Objects. In A. Paepcke (editor), *OOPSLA '91 Conference Proceedings*, pages 97-112. ACM/SIGPLAN, ACM Press, New York, 1991.

[IBM 92] IBM. *The OS/2 Multimedia Advantage.* IBM Corp., 1992.

[IMA 93] IMA. RFT: Multimedia System Services Version 2.0. 1993

[Kahn 79] K. Kahn. *DIRECTOR Guide*. Technical Report MIT AI Laboratory Memo 482B, MIT, December, 1979.

[Kolstad 90] R. Kolstad. The Network Time Protocol. *UNIX Review* 8:58-61, December, 1990.

[Linton 89] M. A. Linton, J. M. Vlissides, P. R. Calder. Composing user interfaces with Interviews. *Computer* 22:8 - 22, February, 1989.

[Little 91] T. D. C. Little and A. Ghafoor. Spatio-Temporal Composition of Distributed Multimedia Objects for Value-Added Networks. *Computer* :42-50, October, 1991.

[Newcomb 91] S. R. Newcomb and N. A. Kipp and V. T. Newcomb. The HYTIME Multimedia/time-based document structuring language. *Communications of the ACM* 34:67-83, November, 1991.

[Palay 88] A. J. F. Palay, M. Hansen, M. Kazar, M. Sherman, M. Wadlow, T. Neuendorffer, Z. Stern, M. Bader, and T. Peters. The Andrew Toolkit - An Overview. In *Proceedings USENIX Technical Conference*, pages 9-21. USENIX, Winter, 1988.

[Ripley 89] G. D. Ripley. DVI - A Digital Multimedia Technology. *CACM* 32(7):811-822, July, 1989.

[Robertson 89] G. G. Robertson and S. K. Card and J. D. Mackinlay. The Cognitive Coprocessor Architecture. In *Proceedings of ACM Symposium on User Interface Software and Technology*, pages 10-18. ACM Press, New York, 1989.

[Rowe 92] L. A. Rowe and B. C. Smith. A Continuous Media Player. In *Third International Workshop on Network and Operating System Support For Digital Audio And Video*, pages 334-344. IEEE Computer and Communication Societies, November, 1992.

[Tokuda 90] Tokuda, H., T. Nakajima, and P. Rao. Real-Time Mach: Toward a Predictable Real-Time System. In *Proceedings of the USENIX Mach Workshop*. USENIX, October, 1990.

[Wayner 91] P. Wayner. Inside QUICKTIME. *Byte* 16:189, December, 1991.

[Xmedia 92] Digital Equipment Corporation. XMedia Tools, Version 1.1A. Software Product Description SPD 36.55.02. 1992

[Yager 92] T. Yager. The Multimedia PC: High-powered Sight and Sound on Your Desk. *Byte* 17:217, February, 1992.