Tactus: Toolkit-Level Support for Synchronized Interactive Multimedia

Roger B. Dannenberg, Tom Neuendorffer, Joseph M. Newcomer, Dean Rubine

Information Technology Center, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213 USA

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.

1 Introduction

Recently, many proposals have emerged for extending graphics systems to support multimedia applications with sound, animation, and video [17, 18, 16, 7]. Other research has been directed toward real-time transmission of multimedia data over networks [13, 2] and standards for the representation and exchange of multimedia data [14]. New capabilities for real-time interactive multimedia interfaces [5] 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

In Proceedings from: Third International Workshop on Network and Operating System Support for Digital Audio and Video, pages 264-275. IEEE Computer and Communication Societies, San Diego, November 12-13, 1992.

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*, whereby precomputed media can be selected 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, teleconferencing, 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 [12], although this assumption is not critical for most applications.

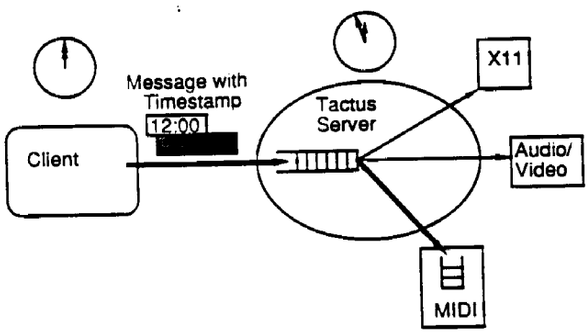

Fig. 1. The Tactus System. Clients send timestamped data (heavy lines) to the server ahead of real time. Data is buffered and then delivered to various presentations devices. Some presentation devices (e.g. MIDI as shown here) may accept data early and provide further buffering and more accurate timing 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.

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 4).

Input is handled in mirror image to output. There is latency between real input at the device level and the arrival of the input data at the application process, so all input must be timestamped. Input events can then be related back to the output that was taking place at the time of the input. It is up to the application to deal with the delay between input and output, for example, by "rewinding" to an indicated stop point or reflecting the input in future output.

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 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 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 [3, 4]. This work is in turn related to discrete-event simulation. Later, Anderson applied these ideas to distributed multimedia [1], but not to interface toolkits. Active objects have long been used for animation [11] and music [6] systems, but have only recently gained attention in multimedia circles [10]. 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 [17], Microsoft's MPC [18], and Dec's XMedia [7]. These systems emphasize storage, playback, and scalability. HyTime [14] 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.

2 The Tactus Toolkit Extensions

The Tactus Server could be used without the Tactus Toolkit, but this would require the user to compute data in advance of real time, implement various protocols (described in Section 3), and interleave computation for various streams. The toolkit simplifies these programming tasks. Our toolkit extensions 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).

2.1 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 to ATK [15]. 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 (according to its clock), and the `Kick` method is called by the system 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`.

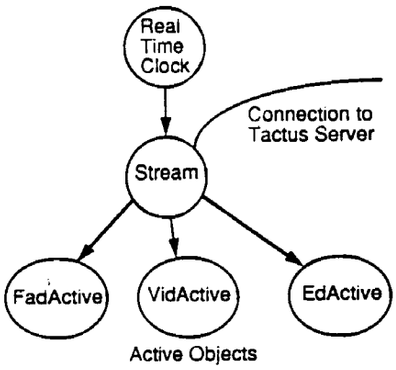

Fig. 2. A Clock Tree. Objects, including clocks, request a wake-up message from their parent in the clock tree. `RealTime` is at the root of the tree and interfaces with the operating system timing facilities. `Stream` is a subclass of `Clock` and manages connections to the Tactus Server. The leaves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

2.2 Clock Objects

Clocks are a subclass of `Active`. Each clock object keeps track of all the active objects (users) 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 users. When a change in the mapping of time occurs, users of the clock are notified (whether or not they are waiting for a `Kick`). We call the time seen by users 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.

2.3 The RealTime Object

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 [8].

2.4 Stream Objects

Stream objects are a subclass of `Clock`. In addition to scheduling and kicking users, stream objects communicate with the Tactus Server and establish timestamps for Tactus messages. Stream objects also schedule their children ahead

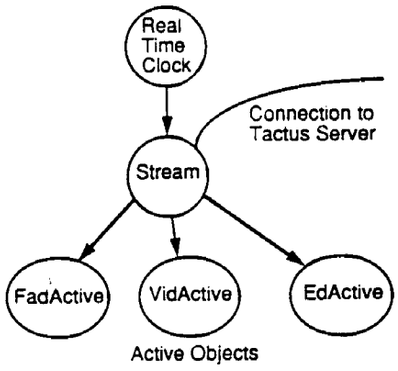

Fig. 2. A Clock Tree. Objects, including clocks, request a wake-up message from their parent in the clock tree. `RealTime` is at the root of the tree and interfaces with the operating system timing facilities. `Stream` is a subclass of `Clock` and manages connections to the Tactus Server. The leaves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

2.2 Clock Objects

Clocks are a subclass of `Active`. Each clock object keeps track of all the active objects (users) 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 users. When a change in the mapping of time occurs, users of the clock are notified (whether or not they are waiting for a `Kick`). We call the time seen by users 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.

2.3 The RealTime Object

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 [8].

2.4 Stream Objects

Stream objects are a subclass of `Clock`. In addition to scheduling and kicking users, 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, a 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.

3 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.

3.1 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.

In a slight variation of the above, Tactus may choose to 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.)

3.2 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.

3.3 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 should be supported as well [1]).

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 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 3). 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.

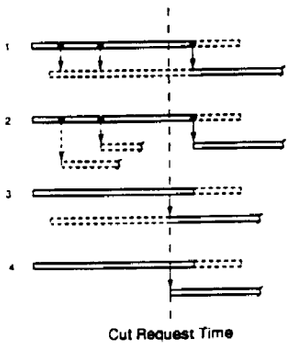

Fig. 3. There are four (4) types of cuts. The top two cuts are restricted to discrete time points, whereas the lower two can take place at any time. The first and third cuts here cut to a stream already in progress while the second and fourth types cut to the beginning of a stream.

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, the application will generate graphics or video for an intersection before knowing whether the user will say "turn left" or not. At this point, 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[1] 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. Clients can also specify an initial set of (initialization) commands to be issued when a cut takes place.

5 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.

5.1 The Editor

As far as this discussion is concerned, the function of the editor (see Figure 4) 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 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. 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.

5.2 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

[1] 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.