A Comparison of Streams and Time Advance As Paradigms for Multimedia Systems

Roger B. Dannenberg and Dean Rubine

March 1994
CMU-CS-94-124

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

A common model for multimedia systems is the stream, an abstraction representing the flow of continuous time-dependent data such as audio samples and video frames. The primary feature of streams is the ability to compose processes by making stream connections between them. An alternative time-advance model is related to discrete-event simulations. Data is computed in presentation order, but in advance of the actual presentation time. Timestamped, buffered data is subsequently output with low latency. The primary feature of time-advance systems is accurate output timing. Stream-based and time-advance systems are compared in terms of the programming model, flow control, buffering, support for interaction, synchronization, modularity issues, and real-time requirements.

This research was performed by the Carnegie Mellon Information Technology Center and supported by the IBM Corporation.

Keywords: Multimedia, Streams, Time Advance, Synchronization, Audio, Video, Real Time, Operating Systems

1. Introduction

The stream or dataflow model is often accepted as the standard paradigm for processing continuous-time data such as audio and video. Streams accomplish several purposes: they decouple the producers and consumers of data, they support modular interconnection of stream processing elements, and they support incremental real-time processing of long-running presentations.

Another model for multimedia processing is what we will call the time-advance model. Here, the emphasis is on pre-computing and timestamping data for accurate delivery by a separate presentation process or server. Time-advance systems offer some interesting support for synchronization and also present an interesting model to the programmer.

The purpose of this report is to examine these two approaches to gain a deeper understanding of what each has to offer. We will begin with a description of each model. Then we will compare streams and time-advance techniques along dimensions of: the programming model, flow control, buffering, support for interaction, synchronization, and modularity issues. Finally, we will present our insight into these issues and some recommendations and considerations for future systems.

2. Streams

A common model for multimedia systems is based on the notion of *streams*. A stream is a time-ordered flow of information from a *source* to (usually) one *target*. Streams imply incremental processing of data, which is appropriate for large data sets and continuous real-time data. Streams support modularity by allowing interconnections among software and hardware devices to be determined dynamically. For example, audio data can be routed from a file system directly to an audio output device or to a software mixer where the data is combined with other audio.

Streams resemble ‘‘patch cables’’ in that they connect outputs to inputs, but unlike the hardware counterpart, software streams do not carry data at the speed of light. Typically, streams are implemented as a queue containing a set of data buffers. In distributed systems, a stream may cross several process and machine boundaries, with data buffers all along the way. The total delay from source to target may be substantial. Furthermore, the delay time is often variable: data that is synchronized at the source may not be synchronized when it reaches the target.

3. Time Advance

Another model found in multimedia systems is *time advance*. The idea is to compute media ahead of the presentation time to compensate for variations in computation time and other sources of delay. In a typical system, a media producer (such as a video file server) will try to maintain a fixed time advance. For example, a video file server will read video 1 second ahead of real time, sending the data to a video consumer such as a video display. There, the data is buffered until its presentation time. If the video server falls behind due to disk, network, or processor contention, the presentation can proceed without delays for up to one second. Thus, the source can have up to 1 second of jitter without imposing any jitter upon the target.

Time advance can also be used to regulate the computation of multimedia presentations. Imagine a system in which various processes compute media according to a clock. A video process reads and displays a video image 30 times per second. A MIDI process waits until the time of the next MIDI message and sends it. The policy here is to compute data only at the time you need it. The problem is that by the time you finish computing the data, it might be too late, especially if there are many processes and several of them try to compute at the same time. The time-advance solution is as follows: schedule processes according to a clock that is fast by some amount (the time advance). When a process produces output, attach a timestamp to the data and place the data in a first-in, first-out (FIFO) queue. A high-priority, low-latency process then copies data from FIFOs to output devices. Data is output at (real) times indicated by the timestamps. Assuming the final output can be done with minimal computation time and at high priority, the output timing can be more accurate than the simpler case where processes output data directly to devices. This is because the time advance allows data to be pre-computed.

4. An Incomplete History

The term ‘‘streams’’ was introduced in the dataflow computer literature by Weng [Weng 75], but even then, the idea was not a new one. Streams are undoubtedly related to signals in the circuit theory of electrical engineering. These streams exist as abstract entities in circuit diagrams and also as tangible wires and connectors in every audio and video system. Wires and patch bays predate computers, and indeed, computers are built by wiring together logic modules. Similar modularity exists in analog circuit descriptions, and in particular, communication systems. Typical sources and targets are transformers, filters, and transducers. Inspired by these system-level descriptions, Max Mathews developed the notion of *unit generators* at Bell Laboratories in the early 1960’s. A unit generator is an operator applied to streams of digital audio. Examples of unit generators include adders, multipliers, digital oscillators, and digital filters. Unit generators in Mathews’ ‘‘Music’’ programs [Mathews 69] were perhaps the first implementation of a modular software system based on streams.

The concept of time advance has also been around for quite some time. Discrete-event simulation introduced the notions of events, timestamping, and controlling the system with a virtual clock. In simulation, it is usually the goal to run the virtual time as fast as possible to reach a final outcome. David Anderson and Ron Kuivila, working on computer music systems, modified the discrete-event simulation idea to run in real time [Anderson 86]. In their approach, the virtual clock runs only slightly ahead of real time. The object is to compute output events ahead of real time, buffer them, and then dispatch them quickly and accurately. Anderson and his students went on to use time-advance concepts in the ACME multimedia system [Anderson 91]. This work did not retain the ideas of discrete event simulation.

Tactus [Dannenberg 93] is similar to to ACME in many ways, but it focuses on making application programs easy to write when computational and interactive media are produced. Tactus clients use a graphical user interface toolkit that includes support for active objects. Active objects perform much like objects in a discrete event simulation, and their output is timestamped and buffered much like Anderson’s and Kuivila’s computer music systems.

5. The Programming Model

Multimedia and real-time software is difficult to write in part because it can be hard to reason about parallel processes, synchronization, and time dependencies. Ideally, the programmer should have a clear model of the system he or she is programming, and the model should make it easy to reason about program behavior.

5.1. The Time-Advance Model

One of the big advantages of the time-advance approach is the programming model it offers. The approach assumes that computation takes place instantaneously at discrete time points. Either a process waits for a time, at which point the computation proceeds infinitely fast to the next *wait* statement, or subroutines are scheduled to execute to completion at discrete time points. In either case, time does not *logically* progress during a computation, so all output produced by a computation is synchronous. As explained earlier, real computers take finite time to execute instructions, but by advancing the execution time and timestamping the outputs, a good simulation of the programming model can be obtained.

Schematically, a typical process in the time-advance model looks like the following:

```
outputTime = 0
loop
        WaitUntil(outputTime);
        ComputeOutput();
        outputTime = outputTime + period;
end loop
```

This code is scheduled according to an advanced clock, and output must be timestamped and buffered. The timestamp is implied by the process wakeup time, in this case, the *outputTime* variable passed to *WaitUntil*. Any output instructions invoked by *ComputeOutput* must attach this timestamp and place the output into a FIFO, and another process must take care of delivering media from the FIFO to output devices.

In Tactus, for example, active objects request a message at a designated time. When an advanced clock reaches that time, the time is stored in a global variable and the active object is sent a *kick* message. When the active object calls an output routine, the routine reads the global timestamp, combines it with the output data, and sends the result to the Tactus server. The Tactus server queues output media until the time indicated by the timestamp. It then delivers the data to the appropriate output.

The advantage of using a time-advance scheme is that the programmer can write code as if it runs in real-time on an infinitely fast processor. Timestamping and buffering are hidden in the scheduling mechanism and in the output routines. This hides a considerable amount of detail and makes programs easier to reason about. In particular, programs that manage multiple media or multiple channels of coordinated output are much easier to write if time advances in a coherent fashion. For example, suppose you want to cross-fade both audio and video from one source to another. A simple implementation would look like the following:

```
outputTime = 0;
loop
        WaitUntil(outputTime);
        xFade = ComputeXFadePercent(outputTime) / 100.0;
        -- compute the video
        frameA = ReadVideoFrame(FileA);
        frameB = ReadVideoFrame(FileB);
        frameC = (frameA * xFade) + (frameB * (1 - xFade));
        OutputVideo(frameC);
        -- compute the audio
        audioA = ReadAudioFrame(FileA);
        audioB = ReadAudioFrame(FileB);
        audioC = (audioA * xFade) + (audioB * (1 - xFade));
        OutputAudio(audioC);
        outputTime = outputTime + framePeriod;
end loop
```

This example assumes that data comes from two files, *FileA* and *FileB*, and that frames of video are interleaved with audio. The arithmetic computations in this example are only intended to illustrate where image and signal processing take place. Here, we assume the image and audio are represented in a linear, uncompressed format so that crossfading is a feasible operation. However, crossfade is just one example among many, and these remarks are valid regardless of the medium and representation.

Consider the problems that are solved in this example. First, there is the problem of audio and video synchronization. Audio and video data are typically subjected to very different kinds of buffering and processing before the data reaches the output device. Audio devices are often subsystems with several levels of buffers and a DSP. Compressed video also requires buffers, processing, and delay (although in this example, decompression must take place before the image processing step). The problem for the programmer is how and when to apply the crossfade operation so that both video and audio crossfade together. There are three possibilities:

- Apply crossfade to streams without paying attention to timing and synchronization issues. This approach exhibits timing errors in the output comparable to the timing skew of the audio and video data at the point where the crossfade is applied. Depending upon the details of the implementation, one can imagine that in a quick audio crossfade, the gain of stream A might be raised before the gain of stream B is lowered. This results in extra loud output or even digital overflow which is an extremely undesirable form of distortion.

- Apply crossfade independently to separate audio and video streams. By counting frames, a ‘‘local stream time’’ can be obtained, and *ComputeXFadePercent(localStreamTime*) gives the amount of crossfade. Problems include:
    - Interleaved calls from the audio and video streams would not access the crossfade function at increasing time ordinates. Therefore, *ComputeXFadePercent* must provide essentially random access to a function of time because the *localStreamTime* in the audio stream may be ahead of *localStreamTime* in the video stream. In contrast, in the example code above, *ComputeXFadePercent* is called with strictly increasing time values, so the cross-fade function could itself be a stream that is accessed sequentially.

  - In other situations, there may be an intermedia dependency. For example, audio panning might track the position of an object in an animation stream, or a VU meter might be computed from audio samples. In these cases, it is very convenient to compute audio and image data in synchrony.

- Apply crossfade as in the example. Outputs are pre-computed and timestamped, and it is the system's responsibility to regulate buffer lengths so that data is presented synchronously. This seems to be the simplest and most effective solution.

5.2. The Stream Model

The programming model for streams is quite different from that for time-advance systems. With streams, processing is performed on data as it becomes available. Since buffers exist between streams, each source or target for a stream has a different local time. This makes communication between separate processes problematic. We have already discussed how difficult it would be to achieve simultaneous cross-fades if audio and video were running in separate streams according to different local times. Any computation that is to be coordinated across multiple streams will have this problem, and this will be discussed further in Section 10.

Compared to the time-advance model in which all data is computed synchronously in time order, the stream model poses some difficult problems for the programmer. However, a characteristic of the streams approach is the resulting modularity of the system. This is a benefit to the programmer.

Specifically, streams provide for intuitive dynamic connections between sources and targets of streamed data. In the words of Gibbs [Gibbs 91]:

> Our hope is that such a framework will make configuring multimedia applications as easy as plugging together modular stereo components.

Streams also solve the problem of determining the locus of control, which is a common difficulty in distributed systems. If data is to pass from process A to process B, does A actively ‘‘write’’ the data to B, or does A passively await a ‘‘read’’ request from B? In the first case, the control is with A, whereas in the second case, the control resides with B. If the protocols assumed by A and B are incompatible, then communication cannot take place without some form of intermediary, which adds complexity and overhead.

The streams solution is to dictate a standard protocol followed by all components. For example, in the case of OS/2 MMPM, a stream source repeatedly allocates a data buffer, fills the buffer, and sends the buffer via the Sync/Stream Manager. A target repeatedly reads a buffer, empties and frees it through interaction with the Sync/Stream Manager. In this case, the Sync/Stream Manager is an intermediary ‘‘which adds complexity and overhead,’’ but at least sources and targets can be interconnected in any configuration (as long as the source and target have a data type in common.)

In general, the stream model provides the ‘‘right’’ abstraction when the programming task is to process a single stream of data asynchronously. When the stream should be processed in lock-step or even in approximate coordination with other streams or events, then the model creates programming difficulty.

6. Flow Control and Buffering

Buffers are necessary at the interfaces between asynchronous processes. Where buffers exist, flow control must be used to prevent unwanted overflow or underflow.

6.1. Streams

In the OS/2 MMPM stream model, processes are the normal source and/or target for streams, although streams may also originate and terminate at the interrupt level. Considering interrupts to be a special case of a thread, there is always a separate thread of control on both the sending and receiving end of a stream. This implies the existence of buffers between processes, and processes can block when reading an empty buffer or writing a full one.

Assigning a thread to each source or target is not the only possibility, however. Streams can be implemented as a single thread shared among objects that are either ‘‘eager’’ or ‘‘lazy.’’

In the ‘‘eager’’ case, the send operation invokes a handler for data arriving at the target. Here the protocol is that stream sources always call an output routine and handlers or call-back routines are provided for input processing.

In the ‘‘lazy’’ case, when a stream target wants input, it calls an input routine, which invokes a ‘‘fetch’’ routine in the stream source. Here the protocol is that stream targets call a routine for input and provide a ‘‘fetch’’ routine for output.

Although this is not usually stated explicitly, one goal of the streams approach is to allow precomputation, pre-delivery, and buffering of continuous media. One would expect ‘‘downstream’’ processes to run at increasingly higher priority and decreasing latency to insure final stream delivery with the greatest timing accuracy. The single-threaded stream approaches do not have this property and therefore are not seen as often as multi-threaded streams systems. However, when very high performance and very low latency are required, a more synchronous, single-threaded approach may be appropriate. An example is the IRCAM Signal Processing Workstation [Puckette 91], in which signals are passed along streams through many operators. If every processing stage involved buffering, the total system input-to-output latency would be many times the duration represented by one block of samples. Therefore, a single thread passes the output of one operator synchronously to the next, buffers are only needed at hardware input and output stages, and the total buffering is held to a small number of milliseconds. In this particular system, operator execution is neither eager nor lazy, but statically scheduled by a compiler.

6.2. Time Advance

In the time-advance model, buffers exist only to reduce the jitter of output. Low jitter depends upon the following assumptions:

- Media ‘‘frames’’ can be output with low latency.
- The source of media frames can compute ahead of real time.

As long as the source can compute ahead and buffer data, output will have low jitter (accurate timing), independent of the timing variation in the source. Buffers are only allocated as needed to smooth out this source variation.

Flow control is straight-forward in time-advance systems. Media is generated at the rate of one second of media per second, using a real-time clock to maintain a fixed time advance. No other flow control is necessary unless the source falls behind its real-time deadline. This causes an underflow of the buffer.

When underflow occurs in the Tactus system, the Tactus server halts all synchronized output until the buffer is filled to a low-water mark. This effectively stops the scheduler on the output side. If the source is not informed, the time-advance will grow. In Tactus, we avoid this by sending a message from the server to the client. The client slips its schedule by an equal amount so that the previous time advance is restored.

In general, it seems that buffers should be used whenever there is a mismatch in jitter across an interface. This is usually the case where input or output hardware interfaces with software. In process-to-process interfaces, buffers may add latency and achieve no benefit. This comment is especially applicable to stream-based systems where asynchrony is added for reasons of modularity rather than performance.

7. Interaction

Interactive media requires that presentations change according to real-time input. The degree of interactivity can vary from simply starting and stopping a stream to continuously computing 3D graphics according to head position and orientation in a virtual reality system. An important parameter of an interactive system is the allowable latency or response time. In general, the longer the allowable latency, the more data can be precomputed and buffered. Since both the time-advance and the stream model precompute and buffer data, it is interesting to look at how interactive systems are built within these models.

7.1. Time Advance

The time-advance model is used by many interactive music performance systems. In these systems, it is frequently the case that sophisticated music generation algorithms generate output. The advantage of time advance is that musical output can be accurately timed. For example, all the notes in a chord will start together, and rhythmic passages will be played with regular and precise timing. We are particularly sensitive to timing in music, so it is not surprising that time advance has been employed often in the music domain.

As long as music is generated by reading scores or by algorithms that compute ‘‘into the future,’’ the time-advance approach works well. However, problems arise when there must be a low latency between input and output. For example, imagine a system in which pressing a piano key causes a musical chord to be output. In a pure time-advance system, the output of the chord will be delayed by the duration of the time advance. In a highly interactive music performance system, this can be a serious problem.

Time-advance systems usually buffer enough data so that even in the worst case, data will be computed in advance of its presentation time. Some time-advance systems provide a special interactive or startup mode that takes advantage of average-case performance. For example, in the Tactus system when a data stream is started, the client initially computes data as fast as possible. As soon as data is buffered in advance of real-time by a fixed low-water mark, output can begin. This provides quick response in certain cases.

Consider the case where a user starts a digital video presentation by clicking on a start button. In the pure time-advance case, there will be a delay exactly equal to the time advance between clicking on the button and the first frame of the video. In the Tactus case, the delay will be whatever time is needed to buffer a small amount of video. If the processing were infinitely fast, there would be no delay at all. Similar optimizations can be made to stop presentations by implementing a flush command that removes pre-computed data from output buffers.

Time-advance systems are sometimes useful even in highly interactive applications, provided the amount of time advance is low. For example, in his video telephone application, Jeffay [Jeffay 93] used time advance to keep audio and video in synchronization. The total latency was a few hundred milliseconds — enough to make explicit synchronization important, but small enough for a usable system.

Time-advance systems can also support interactivity by ‘‘taking shortcuts’’ within the time-critical regions of the system. Input data is processed immediately, and results are output rather than being timestamped and buffered. A good example is mouse tracking. If mouse locations are passed to a time-advance process, there will be a lag in cursor movement. Since cursor movement is usually simple to compute, systems typically avoid the lag by updating the graphics display directly rather than routing mouse events through the application. This ‘‘shortcut’’ is useful whether or not the application uses time advance. Other examples of shortcut processing include audio muting, MIDI THRU, and graphics output clipping.

In a slightly more sophisticated approach, timestamped output can leave some information unbound until presentation time. Consider an application where audio data is read over the network from a remote disk and output to a local audio interface, and the user is presented with a slider for volume control. The file system presents a long latency, so the time advance is large, but that leads to an annoying lag when volume is adjusted. A solution is to output commands of the following form: ‘‘at time *t*, scale this block of audio samples by the current volume factor.’’ The volume factor is set asynchronously whenever the slider moves, and the new volume level is propagated to the output almost immediately. This scheme is illustrated in Figure 1.

A feature of the Tactus server is support for *cuts*, which are synchronous, low-latency transitions between media streams. Cuts support interaction by enabling low-latency responses when a choice is made by the user. For example, the user might select a turn in a surrogate travel application. To perform a cut, the application first precomputes and buffers the alternative media stream. This stream is buffered along with the primary stream in the Tactus server. If the user makes a selection (such as ‘‘turn right’’), the application tells the server to ‘‘cut’’ to the alternative stream. Since the alternative stream is already buffered at the display site, a clean transition can be made with minimal delay.

The stream approach could also be used to support cuts, but since a separate ‘‘cut’’ object would be used for each medium (audio, video, graphics, etc.), it might be hard to make the cut synchronous across all media. Furthermore, Tactus can reject the cut request if the alternative stream has not been precomputed to a given low-water mark. This might also be hard to do using streams because each stream independently buffers data.

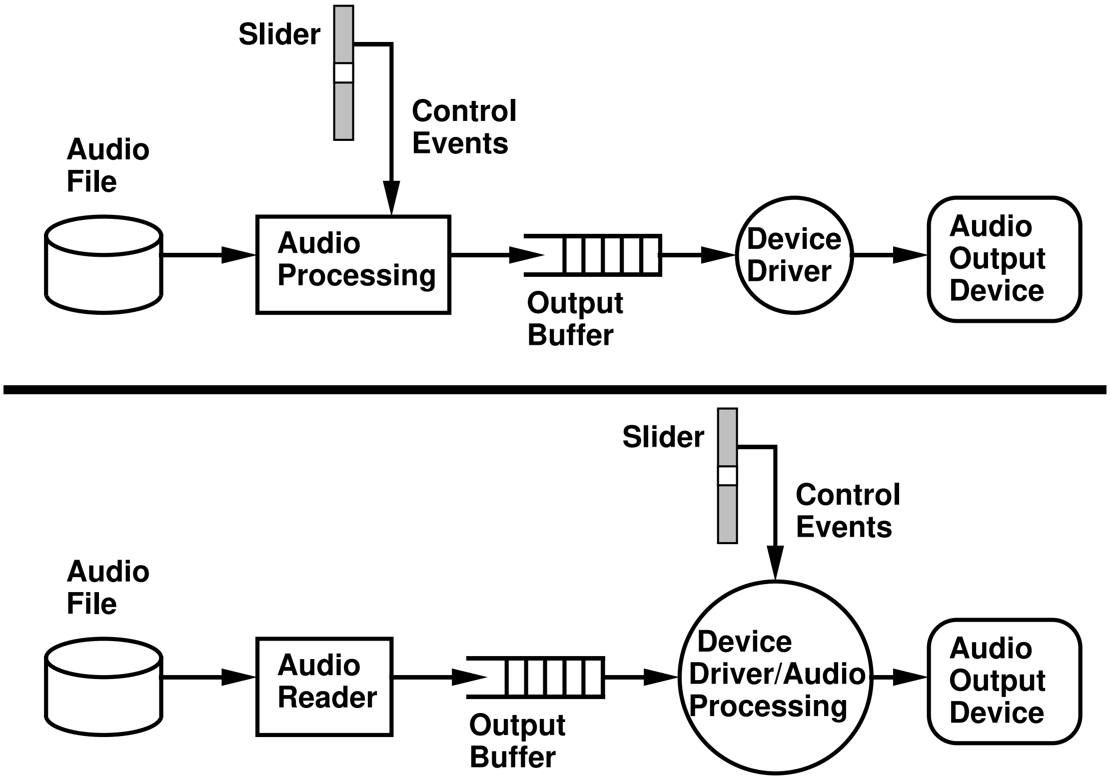

Figure 1: Two implementations of digital audio volume adjustment. If all processing is done in advance (top of figure), there may be lags between moving the slider and hearing the change. In the lower configuration, scaling occurs just before sound output, so volume changes are heard immediately.

7.2. Streams

Because streams also buffer data in advance of real time, we should expect streams and time-advance systems to be similar with respect to interactivity. When a large amount of buffering exists between input and output, the delay will interfere with low-latency interaction. Stream-based systems *can* be designed to minimize latency, typically by using either an eager or lazy evaluation strategy as opposed to an asynchronous one. To reduce latency, data is typically processed in fixed-sized buffers. If a stream passes from a source through several filters and finally to a target, execution will begin with the source and proceed sequentially through each filter and finish with the target. Thus, data is moved from the source, through all the filters, and out to the target as fast as possible. No data is left behind in buffers.

In eager evaluation, the control flow is typically compiled into a straight line of operations that is executed periodically. In lazy evaluation, execution starts at the target with a request for data, which recursively propagates back to the source(s). As the stack of requests ‘‘unwinds,’’ data is moved along the stream from source to target.

Streams offer another possibility for interactivity, related to the ‘‘shortcut’’ approaches discussed earlier. Operators on streams often take parameters and accept control messages. Consider an ‘‘audio fader’’ stream operator that scales audio samples according to a scale factor that can be changed by a message. The previous example (Figure 1) of audio playback with volume control is easy and natural to implement with streams, as shown in Figure 2. The remote file system is the stream source, and due to expected high latency and variance, large buffers are used for this part of the stream. The stream is then passed to an audio fader and from there to audio output, presumably with short buffers. When the volume slider is moved, a message is sent to the audio fader object. Changes are heard immediately because there is relatively little buffering between the fader and the output.

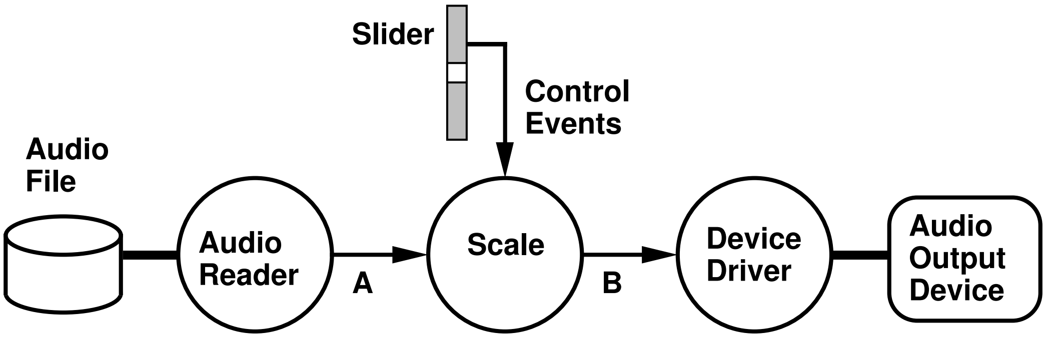

Figure 2: Volume adjustment implemented with streams. The buffering at B should be minimal in order for the output to respond quickly to slider changes. Buffering at A can be large to accommodate a remote file service.

8. Synchronization Issues

Whenever data is buffered, special care must be taken to insure that multiple streams of media remain synchronized. Media synchronization involves the following steps:

1. Develop a mapping from media frames to time. Here, ‘‘frame’’ means an audio sample, a video frame, or even a timestamped animation event.

2. Establish a clock, which is a time reference. The clock may be a hardware real-time clock, or it may be a designated media stream. In the latter case, the time is determined by the mapping from frames introduced in step 1. Note that any media device (audio output, video input) can serve as a clock.

3. Adopt a general clock-synchronization protocol. The word ‘‘general’’ is used because we may be synchronizing real hardware clocks, or we may be comparing the rates of media devices. In a distributed system, clock synchronization keeps source and target data flowing without filling up or underflowing buffers. Locally, clock synchronization is used to, say, compare the time of an audio stream with that of a video stream. Clock synchronization, or at least noticing that clocks are out of sync, is often accomplished by watching the size of stream buffers.

4. Perform rate conversion on media streams to effect synchronization. This may involve changing the frame rate at a source or target, resampling the stream, dropping frames, or doubling frames. Note that steps 3 and 4 may be combined. If clocks being synchronized are media devices, then ‘‘setting the clock’’ may

involve dropping or doubling frames, a form of rate conversion.

The idea that media streams are a special case of clocks and that media synchronization is a special case of clock synchronization is not the common view, but for us it greatly simplifies thinking about synchronization. Separating the problem of detecting skew from the problem of rate conversion is also an important clarification. Note that if atomic clocks become widely available, then the clock synchronization problem may disappear. Atomic clock technology may be carried into the consumer market by the demand for global positioning systems[1] (GPS). Alternatively, GPS technology can be used as a global time reference even without atomic clocks.

8.1. Time Advance

Time-advance systems tend to compute all streams in synchrony and in time order. This simplifies the synchronization problem. Furthermore, data is timestamped, making it easy to see how data is to be synchronized.

In the Tactus system, synchronization is provided automatically by the Tactus server. It is interesting how the underflow detection and recovery mechanisms are integrated with the time-advance approach. For underflow detection, there must be some anticipation of data. This is trivial if the data is a continuous stream of samples, but what if the data consists of graphics and MIDI, which are asynchronous?

In Tactus, the client (media producer) consists of an arbitrary number of *active objects* which schedule themselves to run each time output is needed. When an active object schedules itself to run at some time in the future, this is taken as an implicit assertion that more output will be generated by the object at that time. The client’s runtime system passes this information to the Tactus server. This creates an expectation that more data will arrive at the indicated time. When data does not arrive, underflow has occurred.

A potential problem is that *some* data arrives at the expected time, but not *all* data. This problem is avoided as follows: The Tactus server does not consider data to have arrived at time t until either (1) data arrives with a later time stamp or (2) the client’s runtime system tells the server that no more data will arrive until some time greater than t. (In our system, case 2 is implemented as a null message with a future timestamp; thus, it is handled by case 1).

Loss of synchronization occurs in two cases:

- there are two synchronized clients, and one underflows;
- a device reports to the Tactus server that it did not present data on time.

In either case, the server can stop the presentation until streams and/or devices catch up. Then the presentation can be restarted. Since all data is timestamped, resynchronization can occur without any interaction with the client(s). In practice, clients are informed that the presentation has been delayed so that clients can hold off on computation. Otherwise, buffers might grow too long.

---

[1] When augmented with atomic clocks, GPS systems are much more accurate.

Stopping a presentation is a fairly drastic, though general, form of resynchronization. Other techniques, such as frame doubling and sample rate conversion are also possible within this architecture.

8.2. Streams

Stream-based systems could use similar techniques for synchronization, but it is more common to take a different approach. One difference between streams and time-advance systems is that streams tend to run independently. In the Tactus system, all output is sent to the Tactus server, which serves as a central coordinating site for all media. In contrast, a stream-based system is likely to have an audio stream connected to an audio device driver, a video stream connected to a video device driver, and so on. There must be some communication between these drivers in order to obtain and maintain synchronization.

One approach to stream synchronization, as seen in the OS/2 MMPM, is the use of a master stream to establish a time reference to which other streams synchronize. Streams have ‘‘markers’’ which indicate particular time points. When the master stream consumes data up to a marker, a message is sent to indicate the current location in the stream. In MMPM, the message goes to the Sync/Stream Manager, which relays the message to other streams in the same synchronization group. (A synchronization group is a set of streams that should be synchronized.) Each member of the group receiving a sync pulse message compares its current location to the location specified in the message, and corrective action is taken if necessary.

Notice that in MMPM, it would be difficult to hold up the master stream when other streams underflow. An alternative synchronization architecture, suggested by Tactus, would be to have all streams report positions to the Sync/Stream Manager, and let the Sync/Stream Manager make decisions about starting, stopping, or changing the speed of streams.

Another problem with synchronization by marker messages is that there is no accounting (at least not in MMPM) for the propagation delay of the messages. A more accurate approach would be to communicate position with respect to a global clock. In other words, instead of reporting ‘‘my stream is at time *t*,’’ report ‘‘my stream is running 1.001 times the speed of the system clock and is offset by 2.0 seconds.’’ This form of report does not need to be delivered with low latency because it is likely to remain true for some time.

8.3. Distributed Synchronization

The importance of reporting stream progress with respect to a global clock is very important in a distributed situation, where it may not be possible to transmit marker messages reliably with low latency. It is, however, possible to synchronize clocks in a distributed system, so the clock-based synchronization used in Tactus is a better approach. If we assume that communication in a distributed system may involve substantial delays, then stream startup and recovery from synchronization failures is an interesting problem. The Tactus approach is to designate one server as the ‘‘master.’’ The master server issues control messages to slave servers such as ‘‘start output at time *t*,’’ and servers inform the master when underflow or other failures occur. (Distributed Tactus has not been implemented.) This approach should work for stream-based systems, provided that clock-based rather than marker-based synchronization is used.

While on the subject of distributed systems, it should be noted that these systems often present a range of performance and latencies. In machine-to-machine transfers over a network, performance can be very unpredictable, within a single machine, performance may be more predictable, and within device drivers, performance may be very predictable. The stream model allows extra buffers to be inserted where variation is high. Alternatively, the time-advance model allows greater time advance between components where variation is high. This implies extra buffers, so the two approaches are roughly equivalent in this regard.

9. Modularity Issues

Time-advance systems typically have a single client, or data producer, and a single server, or data consumer. It is difficult to compose processing modules in this model. For example, it is not obvious how to implement an audio filter that can be easily ‘‘plugged’’ into an audio application based on the time-advance model. At this macro-programming level, streams have a definite advantage.

A time-advance system could be extended to have stream-like connections, but there is tension between the two models. Time-advance systems are structured so that all computations are performed in increasing time order. In contrast, a stream source can compute a buffer or several buffers full of data to send to the target. When the stream target is activated, it reads the buffers, essentially going back in time to process data with earlier timestamps. Thus, time does not advance monotonicly. The problems that this causes were discussed in Section 5.1.

Time-advance systems seem to have somewhat better modularity and abstraction at the level of individual media-processing objects. In time-advance systems, the computation is scheduled to take place at some virtual time, but the virtual clock can be a separate object that encapsulates time shifting, time stretching, and time advance. Output instructions can encapsulate timestamping. Section 5.1 illustrated how time-advance programs can be stated very simply, hiding the details of time-advance and timestamping. Section 8.1 discussed how in Tactus, synchronization is achieved by a server. Because synchronization is achieved outside of the media-producing object, the system is simpler and more modular.

In MMPM, in the active objects of Gibbs [Gibbs 91], and in other stream-based systems, stream objects implement timestamping or marking and are responsible for synchronization. Some of the synchronization and scheduling responsibility can be moved outside of stream objects. For example, in MMPM, the Sync/Stream Manager is called to allocate and free buffers. The Sync/Stream Manager throttles stream-processing objects by blocking within these buffer management functions.

10. Streams versus Events

Media computation often involves two sorts of entities: streams and events. A stream represents a continuous signal of some kind, usually represented as a sequence of periodic frames, whereas a frame could be an image, a stereo pair of audio samples, or some other type of data. An event is a non-periodic action such as setting a parameter, instantiating a new stream, or drawing a graphical image. The interaction between streams and events is an interesting, but largely ignored problem.

Events can be synchronous or asynchronous with respect to streams. In the synchronous case, events have timestamps and are generally represented by messages. These messages are enqueued at some stream-processing object. As the stream is processed, the stream time is compared to the time of the first message in the time-ordered message queue. When the times are equal, the message is read and processed. This allows parameter updates to be performed at specific stream times. Synchronous events may actually be implemented as streams. For example, MIDI in MMPM is a stream type, even though MIDI consists of timestamped commands rather than sampled data.

In the asynchronous case, events are processed immediately, and generally take the form of procedure calls (or method invocation in object-oriented systems). Section 7.2, which described an audio fader application, presented an example of asynchronous events. It should be noted that this example works because the audio processing in question is assumed to take place just before samples are output, and the control information is to be acted upon immediately.

Suppose that the slider controls virtual location rather than volume. Creating the illusion of spatial location requires the coordination of panning, reverberation, doppler shift, and filtering. Now, suppose further that these effects are implemented in separate stream-processing objects. That is, one object performs panning, another performs reverberations, and so on. The problem is to update parameters in all of these processing objects when the slider moves. Updates should be coordinated so that they take place at the same stream time, but because of buffers, updates will not take place at the same real time.

There are several solutions to this problem, but none are without drawbacks. In the first solution, synchronous update messages (i.e. messages with timestamps) are sent to each processing object. The timestamps guarantee that updates are all synchronous with respect to the stream. In this approach, there is no guarantee that the updates will arrive early enough. What should the timestamp be? Since upstream objects are computing ahead, the timestamp must be far enough in the future that the samples to be affected have not already been computed.

A second solution is to treat the slider data as a stream itself. By connecting this stream to all the stream-processing objects it can affect, an explicit data dependency is created between the slider and the stream-processing objects. Ordinary stream protocols will ensure that the slider data reaches all stream-processing objects in time to take effect. A drawback of this approach is that many more streams and connections will be created in a complex control system. If controls are used only occasionally, or if control configurations are dynamic, this approach may add considerable overhead. Furthermore, if data does not arrive from the slider, the controlled stream-processing object will block and the output may underflow. With asynchronous input events, it is not clear when new data will arrive unless the source is sampled periodically, but this adds computational overhead.

A third solution, used in the IRCAM ISPW, is to make processing much more synchronous than is the usual case with stream-based systems. On the ISPW, audio processing and event processing alternate in a synchronous fashion. First, 64 samples of audio are processed. (Recall from Section 5.2 that stream processing is statically scheduled to minimize latency from input to output.) Then all event processing is performed to update parameters throughout the system. These two phases alternate, so controls are updated approximately every 1.5ms. At the point where event processing occurs, all stream-operators have processed their input, so all operators are at the same stream time. Control updates to multiple operators are therefore synchronous with respect to all streams.

When the ISPW uses multiple processors, buffers must be inserted where a stream crosses a processor boundary. If an update affects the stream on two different processors, the updates will affect the stream at two different locations, separated by a few milliseconds. The ISPW does not solve this problem, so in practice, processor assignments are made carefully to work around it.

11. Real-Time Requirements

Both stream-based and time-advance systems depend upon real-time operating systems in order to deliver media with accurate timing and synchronization. If timing is provided by hardware such as a sample clock or a video decompression subsystem, the only software requirement may be to prevent buffers from underflowing. When output timing is under software control, other issues arise.

The use of timestamps can allow timing to occur within the interrupt handlers of device drivers, where real-time performance is often easier to achieve. If, on the other hand, devices simply output data as soon as it is available, then it is up to the source of the data to provide the timing. This is often the case in stream-based systems.

Synchronization may also require real-time support. In the stream model, synchronization depends upon the coordination of multiple streams. Often, coordination is achieved by transmitting position information from a master stream to slave streams. In this approach, any latency incurred while informing slave streams of the current position will translate into a loss of synchronization. Alternatively, in the time-advance approach, synchronization depends upon synchronized or shared clocks and accurate output timing, both of which are easier to achieve than real-time communication and processing of streams. For example, OS/2 MMPM streams are not implemented by application threads, and the implementation is distributed across device drivers and high-priority threads, all of which complicate the programming model.

12. Discussion

We have seen that streams and time-advance systems have been developed to address a number of different problems in multi-media systems. In many respects, the two approaches are orthogonal and elements of each could be combined.

The principle advantages of streams are:

- support for modular systems where components can be ‘‘plugged’’ together to implement customized behaviors, and
- decoupling between sources and targets, which allows objects to output media with less jitter and tighter synchronization than exists in the input.

Some disadvantages of streams are:

- relatively high delays due to buffering between processing objects;
- relatively high overhead when stream-processing objects are implemented as processes.

At least a few ideas developed within time-advance systems can be applied to stream-based systems. The following paragraphs consider a variety of issues to be considered in future designs. Whether or not an idea is good or even applicable will depend upon the goals of the system.

One consideration is that buffering between stream-processing objects may not have any real advantage. Buffering requires memory and, if the producer and consumer operate asynchronously, buffering implies synchronization and context switching between the producer and consumer. Thus, there is added memory and processing overhead. A more streamlined approach is to compute data in blocks and to process data sequentially and synchronously. The options are:

- the producer synchronously invokes the consumer to consume blocks as they are produced;
- the consumer synchronously invokes the producer to produce blocks as they are needed;
- a static schedule is executed that guarantees input data is available and output buffers are empty when each operation is invoked. Usually, a static execution schedule is obtained from a topological sort of the data dependencies among the processing objects.

Externally, any of these schemes can look just like a buffered stream system, but the memory and processing loads are reduced, and the input-to-output latency is lowered. This approach is applicable when jitter induced by processing time is not an issue or where jitter can be eliminated by one level of buffering downstream. As examples, the stream-oriented visual programming language MAX (from Opcode) uses this strategy, as does the ISPW described earlier.

Figure 3 shows input, processing, and output in a stream-based system. Buffers are shown explicitly at the input (to prevent overrun) and output (to protect against underflow.) Assuming that all other connections and objects share the same processor and memory, there is little advantage to buffering anywhere else in the stream. Within the circled region, a more synchronous, unbuffered approach can be used. Another issue is the scheduling of many stream operations. In time-advance systems, operations are scheduled according to a virtual clock. If many objects are observing the same clock, they will all run in synchrony. In the distributed case, clocks are useful for flow control. If a producer produces one second of media per second, the receiver’s buffers should not overflow or underflow, and no flow control messages need be exchanged.

This technique could be applied to streams. Stream-processing objects could run according to a virtual clock. External control of the virtual clock could be used to pause or change the rate of data in a stream. If stream-processing objects scheduled their computations to take place at specific virtual times, it would be possible for the system to automatically attach timestamps to data within a stream. Timestamps could then be used, as they are in Tactus, for synchronization.

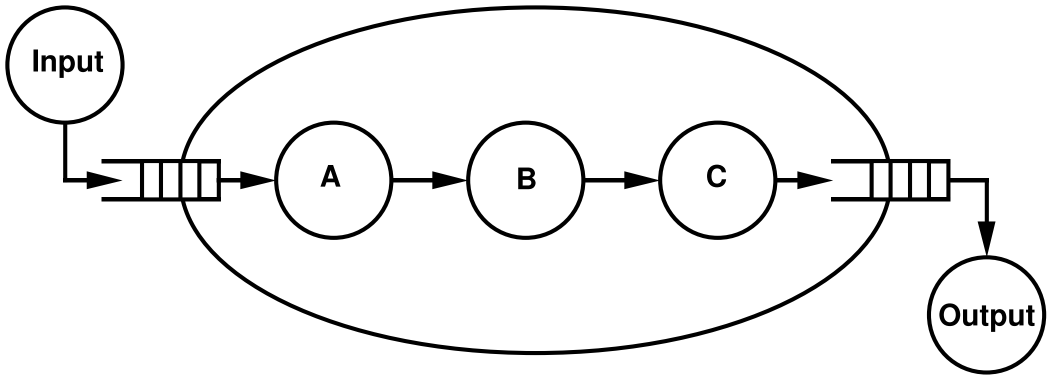

Figure 3: Buffering is required where there is a mismatch between latency or jitter requirements, typically at device or processor-to-processor interfaces. In the stream configuration here, the connections within the large circle could be implemented using synchronous, unbuffered communication.

13. Conclusions

Multimedia systems must process continuous-time data such as digital audio and video. Some of the key concerns are:

- decoupling data sources from output devices to support presentations with low jitter;
- modularity, especially to allow various configurations of sources, filters, and targets;
- synchronization of media presented to users;
- synchronization of parameter updates to signal-processing objects;
- latency from signal input and control changes to signal output.

We have explained and discussed how these issues are addressed by stream-based and time-advance systems. There does not seem to be a single general solution or approach to all of the problems, so understanding the available techniques is important.

A topic that we feel has been underemphasized is the relationship between clock synchronization and media synchronization. Clock-based synchronization, in which media streams are synchronized to local clocks, has the potential to support tighter synchronization with less interaction between streams. This may result in lower computation overhead in single-processor systems, and great simplifications in distributed systems. Note that time-advance systems are most naturally suited to clock-based synchronization because scheduling and time-stamping is all based on clocks. Stream-based systems, however, could easily incorporate clock-based flow control, scheduling, and synchronization.

Overall, we feel the stream model is important from a modularity standpoint, and we expect general systems should support this model. System designers should look closely at how the model is implemented, which aspects of the model provide real benefit, and how other techniques can be incorporated to shore up areas in which the stream model is weak.

References

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

[Dannenberg 93] R. B. Dannenberg, T. Neuendorffer, J. M. Newcomer, D. Rubine, D. Anderson. Tactus: Toolkit-level Support for Synchronized Interactive Multimedia. *Multimedia Systems Journal* 1(2):77-86, 1993.

[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.

[Jeffay 93] K. Jeffay, D.L. Stone, T. Talley, F.D. Smith. Adaptive, Best-Effort, Delivery of Audio and Video Data Across Packet-Switched Networks. *Lecture Notes in Computer Science.* Volume 712.*Network and Operating System Support for Digital Audio and Video Proceedings.* In V. Rangan, Springer-Verlag, 1993, pages 3-14.

[Mathews 69] Mathews, M. V. *The Technology of Computer Music.* MIT Press, Boston, 1969.

[Puckette 91] M. Puckette. FTS: A Real-Time Monitor for Multiprocessor Music Synthesis. *Computer Music Journal* 15(3):58-67, Fall, 1991.

[Weng 75] K. Weng. *Stream-Oriented Computation in Recursive Data Flow Schemas*. Technical Report Technical Memoranda 68, MIT, 1975.

Table of Contents

List of Figures