Time in Distributed Real-Time Systems

Eli Brandt and Roger B. Dannenberg
School of Computer Science,
Carnegie Mellon University
{eli, rbd}@cs.cmu.edu

Abstract

A real-time music system is responsible for deciding *what happens when*, when each task runs and each message takes effect. This question becomes acute when there are several classes of tasks running and intercommunicating: user interface, control processing, and audio, for example. We briefly examine and classify past approaches and their applicability to distributed systems, then propose and discuss an alternative. The shared access to a sample clock that it requires is not trivial to achieve in a distributed system, so we describe and assess a way to do so.

1 Existing approaches

The goal of a real-time music system is to produce certain time-structured data. Ideally, this output's timing would be exactly as specified by the musician. In practice, the timing is imprecise: the system's tasks can't all run when they ought to, nor can they communicate with perfect timing. The music system's job is to minimize the consequent harm to its output, through its control over how tasks are run and how they communicate. Broadly speaking, there are several approaches to this problem: synchronous, asynchronous, and our *forward-synchronous*.

Groups of tasks having the same timing requirements can be formalized as *zones* (Dannenberg and Rubine, 1995); a system might have audio, MIDI, and user-interface zones. Any zone can send data to any other: for example, UI controlling audio, audio rendered on the screen, or audio abstracted to MIDI and echoed to disk. For concreteness we'll talk about one particularly common path, from a sequencer to a synthesizer.

What we call the synchronous approach is exemplified by MAX (Puckette, 1991). This approach interleaves the control and audio zones into a single thread of control, running each to completion. Therefore control updates always apply atomically and to the proper block of audio. Unfortunately, long-running control computation causes audio to underflow. One might prefer to defer this computation, delaying an event onset but preserving audio flow.

The asynchronous approach permits this by running audio and control in separate threads, audio having either priority or the use of an independent processor. They communicate through messages or shared memory, updates taking effect immediately. As a result, events may be early or late, depending on what points of execution the two threads have reached. MIDI ensembles and the IRCAM 4X (Favreau et al., 1986) are examples of this approach. It extends to distributed systems, whereas the synchronous approach does not.

2 Forward-synchronous

The proposed forward-synchronous approach is so called because it has synchronous timing for events in the future, and asynchronous for those in the past. It permits events to be late, but not early.

Control messages are stamped with a time when they are to take effect. Those arriving with a future timestamp are queued, and later dispatched synchronously with audio computation; those with a past timestamp are dispatched immediately.

The timestamp refers to a particular sample, because time is based on the DAC clock:

$$\text{time} = \big(\text{samples written} + \text{samples buffered}\big) / f_{\text{nom}}$$

where f_{nom} is the nominal clock frequency. Though the system clock is quite likely more accurate, it must not be used directly: its very accuracy constitutes a lack of synchronization with the DAC, precluding sample-precise timing.

A sequencer can request precise timing (at the price of added latency) by sending with an offset into the future. If the synthesizer runs a little late, the messages are still in the future, still precisely timed. If it runs so late as to overrun the offset, the events start to happen late. The sequencer sets its own tradeoff between latency and the chance of jitter; it can rule out jitter if the synthesizer's zone has a hard scheduling bound. An accompaniment system, on the other hand, would more likely choose minimal delay and take its chances with jitter.

This approach offers for every message the choice of synchronous or asynchronous behavior, with graceful degradation under uncooperative thread scheduling.

Distributed operation requires a distributed shared timebase, discussed below.

Anderson and Kuivila (1986) also augment the asynchronous model with timestamps, but in a less flexible way. Their approach introduces a fixed latency L between the control and audio zones. Each control message is generated E milliseconds early, which the application cannot schedule but can bound between 0 and L. It is then sent timestamped for E ms into the future, canceling the scheduling jitter.

Think of this as running the control zone L ms ahead of the audio zone. In this way we can view all of the timestamps within a globally consistent timeline. (Without this, events are acausal and time becomes deeply confusing.) Each zone is offset into the future by a different amount. Notice that this view requires that zone connectivity be all 'downhill', from higher offset to lower. In particular, there can be no cycles. This can be problematic, as when a system hopes to both receive from and send to the outside world.

ZIPI messages are timestamped, and the forward-synchronous approach is one option discussed by its inventors (McMillen et al., 1994). They also discuss yet another model in which time-stamped messages are processed and forwarded early for timely arrival at a final destination. This is not allowed in the forward-synchronous model, which (except for late messages) always processes messages synchronously according to timestamps. ZIPI does not seem to assume sample-precise timing and synchronization.

2.1 Implementation

We have applied the forward-synchronous approach to our sound synthesis system Aura (Dannenberg and Brandt, 1996). Its system-specific audio layer provides an estimate of the DAC's read pointer into the buffer. There is a periodic message which triggers computation of an audio block; buffering the block advances time by one block's worth, up to and just past the next trigger message. (Time starts a fraction of a sample advanced from zero, to bootstrap the process.) This results in block-precise timing. Sample-precise timing would be achieved by having every message cause audio computation up to the point of its timestamp.

3 Distributed music systems

Networked computers can form a distributed system with impressive aggregate signal-processing power and I/O. We discuss configurations with multiple A/D/A devices; those with only one or none at all are ready simplifications. Ideally, these multiple devices are synchronized, driven by a master word clock through AES/EBU or the like. This is essential for communicating with most multi-channel digital devices, such as a tape deck or a mixer. It also makes it simple for nodes, by basing time on sample clock, to maintain accurate and consistent global time.

We treat the cases where this is not possible, when sample-synchronizable devices are not available or when nodes are networked but not wired for word clock. This situation is too common to be dismissed as intractable, tempting though it may be. Note that we have all for years made use of devices with less than sample-accurate synchronization: MIDI synthesizers, or recording devices driven by SMTPE or MIDI Time Code. Multiple computers are just another example. The degree of synchronization depends on parameters discussed below, but can easily be better than SMPTE quarter-frame resolution, 1/120 sec.

In this model, events cannot be triggered with sample precision between nodes, and nodes' sample rates differ slightly. This rules out, for example, playing an eight-channel sound file over four stereo sound cards. But these four nodes can play independent audio tracks, or generate quadruple the polyphony of a single node.

Our goals for a distributed audio system are the synchronization and the stability of time. Formally, between all pairs of nodes we constrain the maximum difference in local time, and the maximum difference in rates of local time flow, with the addendum that local time is monotonic. We propose a particular way to achieve these goals (space does not permit exploration of the many options in design), and give a timing analysis. We assume throughout that nodes' sample clocks are constant and identical to within a small fraction of one percent.

4 Clock synchronization

Exact and observable variables

	master's		slave's	
	true	obs.	true	obs.
global time	t_g	$t_{g\|m}$		$t_{g\|s}$
sample count	n_m	$n_{m\|m}$	n_s	$n_{s\|s}$
sample period	p_m		p_s	
sample rate	f_m		f_s	

The obstacle to distributed forward-synchronous operation is maintaining suitably synchronized clocks. We designate one node as the master and define *global time* as a linear function of the master's sample count. Each slave node has a local clock, a mapping to global time from local sample count. Periodically these mappings are updated to maintain synchronization.

Assume that the master sample rate is exact, e.g. a constant $f_{\text{nom}} = 44.1$ kHz. If n_m is the master's sample count, and p_m its sample period $1/f_m$, then global time t_g is

$$t_g = p_m n_m$$

The slave, whenever it needs the time, makes an approximate prediction $t_{g|p}$:

$$t_g \approx t_{g|p} = \int p_c n_s' \, dt$$

Notice that the period used is not the slave's actual sample period p_s— it is p_c, a signal controlled so as to keep $t_{g|p}$ tracking t_g. In practice, the equation in use is the backward-difference discretization

$$\Delta t_{g|p} = p_c \Delta n_s$$

4.1 Observational errors

The slave's task, maintaining a proper p_c, is complicated by limitations of measurement. The master may not know its sample count exactly; it might for example update it only at DMA transfer interrupts. Its measured estimate $n_{m|m}$ differs from the exact n_m by an error $\varepsilon(n_{m|m})$, which we model as a uniform random variable with range $\pm\mathcal{E}(n_{m|m})$. The slave likewise has error $\varepsilon(n_{s|s})$ in measuring its sample count. The master then estimates t_g as $t_{g|m} = p_m n_{m|m}$, so $\mathcal{E}(t_{g|m}) = p_m \mathcal{E}(n_{m|m})$.

The slave doesn't have direct access even to this estimate. Instead, it measures the local sample count $n_{0|s}$ and queries the master for $t_{g|m}$; upon reply it measures the new sample count $n_{1|s}$. For minimax error, we decide that this $t_{g|m}$ corresponds to the midpoint $n_{s|s} = \frac{1}{2}(n_{0|s} + n_{1|s})$. So this replied $t_{g|m}$ becomes the slave's estimate $t_{g|s}$ for $n_{s|s}$.

If the machines and the network were infinitely fast, n_1 would equal n_0 and $t_{g|s}$ would equal $t_{g|m}$. In practice n_1 is greater than n_0. Knowing the system characteristics, we can set a feasible threshold Δn_{01}; the slave retries until the difference is below this threshold. This lets it disregard packet collisions. It also ensures on successive attempts that the master's responding code was running and in the cache, for fast and deterministic response time. These techniques are related to those used in NTP (Mills, 1985) and SNTP (Mills, 1996).

The replied $t_{g|m}$ was measured somewhere in the interval, so we err by at most $\frac{1}{2}\Delta n_{01}$ in assigning it to the midpoint $n_{s|s}$. We err by the query jitter J_{query} in assigning the quoted $t_{g|m}$ to the midpoint of the interval. Our error in locating the midpoint is

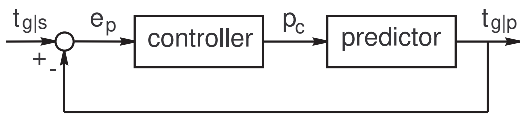
$$\frac{1}{2}\left(\mathcal{E}(n_{0|s}) + \mathcal{E}(n_{1|s})\right) = \frac{1}{2} 2\mathcal{E}(n_{s|s}) = \mathcal{E}(n_{s|s})$$

Therefore the total error is

$$\mathcal{E}(t_{g|s}) = \mathcal{E}(t_{g|m}) + \mathcal{E}(n_{s|s}) + J_{\text{query}}$$

4.2 The controller

Figure 1: a PI control system



To generate the signal p_c we apply techniques from control theory (Franklin et al., 1994; Messner and Tillbury, 1996). The slave's *predictor* generates $t_{g|p}$. The *controller* observes the prediction's error e_p versus the observed $t_{g|s}$, and updates p_c to be used in the next prediction. We use a discretization of a proportional-integral controller:

$$p_c = K_p e_p + K_i \int e_p \, dt \qquad (e_p = \Delta t_{g|s} - \Delta t_{g|p})$$

Coefficients for the controller were found by trial and error aided by simple-minded optimization routines.

Synchronization steps happen nominally at the rate f_{sync}. The actual times will vary due to OS scheduling behavior, and to the necessity of retrying slow queries for t_g. Within reason, these variations do not matter, as the controller looks only at deltas.

4.3 Extension: synthetic sample clocks

If a node suffers high sample-count error but has access to a low-jitter local clock such as CPU cycle count, it can run another PI controller (with coefficients K_{yp} and K_{yi}) predicting a low-jitter estimate of its sample count. This controller can run at a multiple of f_{sync} without adding any network traffic or time-query overhead on the master.

In an extension to our system, we apply this technique to both master and slave sample clocks.

4.4 Results

We can say *a priori* that $t_{g|p}$ is continuous, though $f_c = 1/p_c$ is not. While f_c can in principle be negative, this only happens in implausibly extreme parameter ranges.

Specific numbers are of course dependent on the regime the controller is running under. A/D/A clocks come in many grades, but even many cheap sound cards use crystal oscillators, which according to a review of manufacturers' specifications are ordinarily stable to

±100 parts per million and better. We do not know the time course of the clock drift. Our simulation assumes that each crystal independently swings from one extreme to the other several times in an hour; we expect that this is pessimistic for warmed-up machines.

Network propagation delay determines Δn_{01}. An Ethernet can support a Δn_{01} of 1 ms or less. (Wide-area networks would have longer delays, but synchronization is beside the point once nodes are out of earshot.) The network and the master both contribute to J_{query}; we use a value of 200 μs.

The critical quantities turn out, because we make time depend directly on the sample clock, to be $\mathcal{E}(n_{m|m})$ and $\mathcal{E}(n_{s|s})$ — for simplicity, say they're the same. We consider two cases: sample-accurate sample counts, and errors of ±5 ms. The larger number, implying a block size of 10 ms, should be more than generous for real-time work.

For each regime we give the controller coefficients used. With them we simulate a master and one slave for 24 hours, synchronizing once a second, and report the largest desynchrony seen in time and in frequency. For the high $\mathcal{E}(n)$ we also simulate with the synthetic-clock extension.

Controller coefficients

	$\mathcal{E}(n)$	K_p	K_i	K_{yp}	K_{yi}
MkI	$\frac{1}{2}/f_{\text{nom}}$	0.1	0.01		
	5 ms	0.1	0.01		
MkII	5 ms	0.1	0.01	0.02	0.0002

Maximum simulated errors

	$\mathcal{E}(n)$	Δtime	Δfrequency
MkI	$\frac{1}{2}/f_{\text{nom}}$	0.16 ms	34 ppm
	5 ms	12	1900
MkII	5 ms	1.1	80

4.5 Assessment

Notice that if two slaves are each ±0.16 ms away from the master, they may be 0.32 ms from one other. This doubled number is still quite satisfactory, being well below the perceptible jitter for non-simultaneous events (Michon, 1964; Lunney, 1974). The ±34 ppm frequency variation is a fraction of that inherent in using crystal-based time at all.

The high $\mathcal{E}(n)$ causes severe problems for the standard model. A jitter of 24 ms is easily audible. The frequency variation, almost 2000 ppm, is also too high.

The extended model, however, can tolerate high $\mathcal{E}(n)$. It delivers synchronization to within ±1.1 ms and ±80 ppm. This time shift corresponds to moving one's head about a foot between speakers, and the frequency difference to varying a 120-bpm tempo by 0.01 bpm; we find both acceptable.

5 Conclusions

The forward-synchronous model supports precise timing, graceful degradation, arbitrary connectivity, and distributed operation. A master/slave system based on PI controllers solves the problem of distributed clock synchronization, without the need for special SMPTE or MIDI Time Code connections and processing.

References

Dannenberg and Rubine (1995). Toward modular, portable, real-time software. In *Proc. International Computer Music Conference*, pages 65–72. ICMA.

Dannenberg, R. and Brandt, E. (1996). A flexible real-time software synthesis system. In *Proc. International Computer Music Conference*, pages 270–273. International Computer Music Association.

Favreau, E. et al. (1986). Software developments for the 4X real-time system. In *Proc. International Computer Music Conference*, pages 369–373.

Franklin, G. F., Powell, J. D., and Emani-Naeini, A. (1994). *Feedback Control of Dynamic Systems*. Addison-Wesley, Reading, Massachusetts.

Lunney, H. M. W. (1974). Time as heard in speech and music. *Nature*, 249:592.

McMillen, K., Wessel, D., and Wright, M. (1994). The ZIPI music parameter description language. *Computer Music Journal*, 18(4):52–73.

Messner, B. and Tillbury, D. (1996). Control tutorials for MATLAB. Via WWW. `http://www.me.cmu.edu/matlab/html/`. On CD-ROM with ISBN 0-201-36194-9.

Michon, J. A. (1964). Studies on subjective duration 1. differential sensitivity on the perception of repeated temporal intervals. *Acta Psychologica*, 22:441–450.

Mills, D. L. (1985). Network Time Protocol (NTP). IETF RFC 958.

Mills, D. L. (1996). Simple Network Time Protocol (SNTP) version 4. IETF RFC 2030.

Puckette, M. (1991). Combining event and signal processing in the MAX graphical programming environment. *Computer Music Journal*, 15(3):68–77.