Published as: Roger B. Dannenberg, “Time-Flow Concepts and Architectures For Music and Media Synchronization,” in *Proceedings of the 43rd International Computer Music Conference,* International Computer Music Association, 2017, pp. 104-109.

Time-Flow Concepts and Architectures For Music and Media Synchronization

Roger B. Dannenberg
Carnegie Mellon University
rbd@cs.cmu.edu

ABSTRACT

Modern real-time media-processing systems increasingly rely on software processing. Increasing speed and parallelism in multi-core and graphics processors has opened the possibility of interconnecting and running multiple applications to process audio, video, graphics, and other data. However, latency can accumulate as media move through multiple stages of processing, timing can be unpredictable, and synchronization is difficult. One solution to many of these problems is explicit and precise timing in which data and sample streams are organized and coordinated by logical time. This decouples media consistency and synchronization from real time. We call this “time flow” by analogy to data-flow systems, and describe several levels of sophistication and capability in time-flow architectures.

1. INTRODUCTION

Time and synchronization are critical for music, video, and other media. In the early days of computer music, building even one real-time system was an achievement; so synchronizing multiple systems was not a common problem. Often, synchronization problems in early technologies were solved with synchronization signals such as MIDI clock, SMPTE, and sample clock, all running directly over copper for low-latency. These are simple solutions for simple systems.

Today’s systems are dominated by software applications, distributed and/or parallel computing, and packet-based networks. Synchronization is more complex, the types of media are more numerous, and systems are moving from experimental labs and performances to high-budget venues including Broadway, opera, stadiums, and the like.

One particular problem in synchronization that has emerged is buffering and the resulting latency. In earlier times, memory and computing were both expensive, so many processes were analog with no storage and very low latency. Now consider a modern software-intensive system. Control might involve network transmission with associated packets and latency. Output will appear in the form of digital audio buffers that add latency. Additional processes might process the audio. Even modern conversion to analog implies delay for oversampling converters and interfacing. There are many opportunities for delays caused by scheduling and buffering that simply did not exist in more hardware-based systems.

The same arguments apply to video and computer graphics. We have pipelines for software encoding, decoding, processing, rendering, and mixing. Projectors and flat-screen displays may have additional frame buffers for frame-rate conversion, interlacing, de-interlacing, and interpolation. GPUs put teraflops of compute power into our hands, making live interactive software video processing more attractive. Video scheduling, timing, buffering, and synchronization now require careful thought.

Not even experimental systems are solving synchronization and timing problems in a disciplined way. Moreover, the lack of solutions or even proposals has affected the design of APIs that often lose important information needed to solve synchronization problems. *This paper aims to describe and analyze systematic solutions to the delivery of synchronized, interactive media using modern software-intensive computer systems*. The ideas in this paper are not fully implemented, and our goal is to promote discussion and future designs.

After describing some related work in the next section, we describe four approaches to synchronization in increasing levels of sophistication. We call these Synchronization Levels 0 through 3. This is followed by a discussion of input, an example and conclusions.

2. RELATED WORK

Synchronization has been widely discussed. Of particular interest are specification schemes and frameworks [1]. (This reference cites a wealth of additional work.) One could view this work as “top-down,” telling us what we want to deliver, but very little has been done on the “bottom-up” problem of implementation. The traditional view of devices – as responsive, dedicated, low-latency hardware – is outdated and must be addressed.

Anderson and Kuivila’s work on FORMULA [2] provides an excellent model for thinking about latency and synchronization in software. In FORMULA, timing is specified by scheduling and computing within a *logical time system* that runs in advance of real time. Output actions and data are tagged with the current *logical* time and delivered to a high-priority thread or interrupt han-

Copyright: © 2017 Roger B. Dannenberg. This is an open-access article distributed under the terms of the Creative Commons Attribution License 3.0 Unported, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

dler for accurately timed delivery. This is essentially the same mechanism as "strong timing" in ChucK [7], was discussed specifically in Liang, Xia, and Dannenberg [6] and inspired time-stamps in OpenSound Control [8]. Brandt and Dannenberg [3] considered logical time and forward-synchronous timing in the face of distributed systems with varying output latency and varying tempo.

Still, this is not enough, because: (1) systems are not necessarily composed of only two layers (e.g. client and server) and may have more (e.g. control, synthesis, mix, spacialization) and/or multiple media (audio, MIDI, video, animation), and (2) current systems, APIs and protocols are not written to share the underlying latency of media synthesis and delivery. When you change a filter parameter or press "go," when does that take effect? If you plug an effects processor into your audio chain, does that change the latency and upset the synchronization?

The PortAudio API (http://portaudio.com/docs/v19-doxydocs/) hints at how useful information might be provided to applications. The audio callback function provides the time corresponding to audio input time, the current real time, and the time corresponding to the audio output time. In principle, this information can be derived from hardware, e.g. USB audio devices can report internal processing delays [9].

3. LEVEL 0 SYNCHRONIZATION

One approach to synchronization is simply to eliminate as much latency as possible, assume that latency is zero, and operate in real time. MIDI is an example of this approach we call "Level 0." MIDI has no layer of timing specification, logical time, or even time-stamps. When you want something to happen, you send a message, and devices perform the command as quickly as possible.

While Level 0 minimizes latency, synchronization between multiple devices or media is lost when the various output channels have different latencies. In addition, if there are variations in the amount of time it takes to compute or communicate control information, the timing jitter that results is passed along to the output.

4. LEVEL 1 SYNCHRONIZATION

The next level of sophistication in synchronization is to apply time-stamps to events, computing the events in advance within a "control stage." The computed events are delivered with time-stamps to a "rendering stage." There, events are delayed according to time-stamps in order to produce accurately timed output.

Level 1 employs this "forward synchronous" [3] timing to mask the timing jitter that arises during the computation of control information and also from communication delays. In addition, if control information is delivered in advance, rendering stages *with different latencies* can adjust time-stamps to compensate and produce synchronized outputs as long as the latencies are known.

Where networks are involved, time-stamps only have meaning if both systems agree on the time. This problem can be solved with clock synchronization protocols, and previous work has shown that very accurate clock synchronization can be obtained with low overhead [3].

Level 1 synchronization does not address the problems of audio delivery except in the simple case of directly rendering audio from control information and delivering the audio with a fixed, known latency. Moreover, Level 1 does not deal well with *any* situation in which real-time data propagates through multiple stages of processing.

5. LEVEL 2 SYNCHRONIZATION

In multi-stage media processing, software modules accept control information and media streams as input, process them, and pass control and media along through further stages and ultimately to output devices. The paths of data may vary, may hop from one software application to another, and may stream across local networks to take advantage of additional computing resources or I/O devices. We would like to consider all digital media including audio, video, computer animation, and even robots.

The guiding principle of this approach will be the separation of timing specifications (when data should ultimately be delivered) and real time (the current time reported by a clock synchronization protocol). We use forward synchronous timing as in Level 1, but carefully manage and forward time-stamps through many stages.

Level 2 introduces some new problems to solve:

1. We do not restrict computing to two stages,
2. We consider "continuous" audio and video streams along with discrete, time-stamped control information, and
3. We need to combine control information and media streams from different sources.

Each of these problems is considered in the following subsections.

5.1 Handling Multiple Stages

In the simplest case of multiple stages, consider a single source, a single sink, and a multi-stage pipeline in between. In Figure 1, we see a source, three stages of processing, and a sink (the output). Control information is computed by the source at time $t-L$, where t is the logical time and L (for Latency) is the time advance – how early to compute things relative to their output times. Since t is the desired real output time, t is the time-stamp, but since the logical time system runs L seconds ahead of real time, the message is delivered to the first stage at approximately $t-L$. Each stage adds some delay (Δ_1, Δ_2 and Δ_3), so the data (or data derived through processing) is forwarded to successive stages at times $t-L+\Delta_1$ and $t-L+\Delta_1+\Delta_2$. Since the final stage is the output stage, it holds and delivers output according to the time-stamp at time t. Note that this assumes $\mathrm{L} \geq \Delta_1+\Delta_2+\Delta_3$, otherwise the output would be late and should be delivered as soon as possible.

We do not assume processing stages or communications take fixed amounts of time, so information could arrive at the last stage as early as $t-L$ (assuming no delays whatsoever) or as late as $t-\mathrm{L}+\Delta_1+\Delta_2+\Delta_3$. The key is that the time-stamp allows the final stage to compensate for any delays encountered along the way.

How can the source discover the overall latency (which among other things is the basis for determining L)? We propose that protocols for sending timed events can offer this information through messages when connections are made. As shown by the dashed lines in Figure 1, each stage computes and sends the overall output delay to its immediate upstream stage(s). In practice, these delays might include expected, recommended and worst case delays, allowing the source to make the tradeoff between eliminating all late output (large *L*) and achieving lower latency (small *L*).

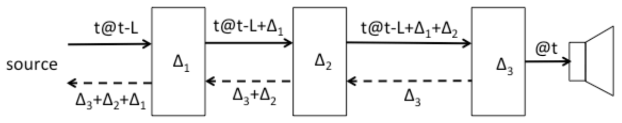

Figure 1. Forward-synchronous time-stamped data in multiple stages. *x@y* means data is time-stamped with *x* and delivered at real time *y*. The dashed lines show delay information passed upstream.

5.2 Audio and Video Streams

Now suppose the data in Figure 1 is sampled data, either audio samples or constant frame-rate video frames. Because the data is "synchronous," we do not place time-stamps on the data. Instead, we expect samples to emerge from the output at exactly the sample rate. Current systems essentially use Level 0, attempting to solve synchronization problems by lowering latencies and controlling timing at the source. (The output time will only be a short time interval later, and we ignore the difference or adjust timing at the input after estimating the latency).

Our approach is different: we start the stream in a known relationship to real-time (and recall that we can have synchronized clocks across all stages even if they are on separate computers). A simple way to synchronize the stream is to specify the starting time (*t*) in the future, start producing samples *L* seconds ahead (at $t{-}L$) and rely on the output stage, e.g. the device and device driver, to deliver the first sample at the correct time.

As with event synchronization, each audio stage in this example might run with lower latency than the anticipated Δ_i. Because samples are synchronous, this would lead to samples accumulating in buffers in the output stage, but this is good because additional buffering will lower the likelihood of underflow in the event that some upstream stage is delayed. Regardless of actual delays and buffer sizes, each stage keeps track of the *logical* time of the stream, i.e. when each sample is due to be output. This information will be used when combining streams as described in the next subsection.

5.3 Combining Streams from Different Sources

The third problem mentioned earlier is combining data from multiple streams. Consider Figure 2, where data from a single source is routed to two different processing stages A and B, then mixed together. For now, refer to the solid lines and ignore the dashed lines. If A has high latency and B has low latency, then without some compensation, the data processed by A will effectively be delayed relative to the data processed by B. This could cause unwanted phasing effects in audio, or in more complex situations, a loss of synchronization between video and audio that derive from a common source.

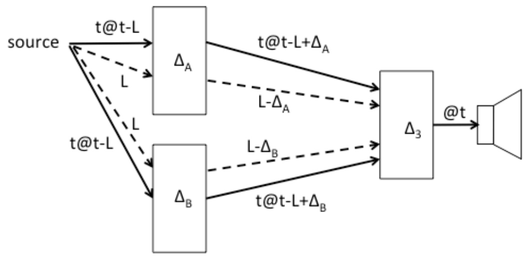

Figure 2. A simple example illustrating the combination of synchronized streams. The source is processed by two stages A (top) and B (bottom) and combined in the output stage (right).

This problem is solved using the Level 2 scheme of starting streams with a known timing state and keeping track of stream times. Assuming audio streams, when samples arrive from B at the output stage, they are held in buffers until the intended output time *t*. Thus, even if A has high latency, the samples from B will not yet be consumed, their intended timing is known, and they can be aligned with A's samples and combined properly.

Notice that Processes A and B do not need any information about the overall configuration. Thus, we preserve the idea that audio processing systems can be constructed modularly from independent software components. Also, the same idea of reporting cumulative delay upstream to source stages applies here (although not shown in Figure 2). Where streams split (as in the source connecting to both A and B), the cumulative delay seen by the upstream stage is the *maximum* of all the downstream stages to which the upstream stage connects.

5.4 Combining Streams of Discrete Timed Events

Control updates, note-on messages or messages that trigger some kind of processing are not quite as simple in this architecture as sampled streams. Timed "events" delivered at discrete times cause the stream time to advance by arbitrary jumps, and we cannot know when the next event will occur. Thus, there is some danger that – in an effort to process events as quickly as possible – events from two sources will be processed out of order.

Referring again to Figure 2, if B delivers an event with time-stamp *t* to the output stage, it may be safe to process the event right away (for example, this might result in inserting the event into a time-stamped FIFO output queue). On the other hand, at that moment, stage A might deliver an event with time-stamp $t{-}\varepsilon$, in other words, just before the event from B. This is possible because A has greater latency and simply took longer to produce an event, even though the event has an earlier time-stamp.

Assuming no other information is available, the stage combining events from A and B should wait long enough to ensure any events from A for time $<t$ have been received before processing any events from B for time $\geq t$. How long is that? In this case, A has latency Δ_A, so we expect to receive events from A at time $t{-}L{+}\Delta_A$ at the latest. Thus, it is safe to process events from B after $t{-}L{+}\Delta_A$. The latency *after which* we can expect all input

to be available is shown as dashed lines in Figure 2. Thus, in addition to communicating data upstream about latency (dashed lines in Figure 1), Level 2 protocols should communicate the maximum allowable time advance for discrete timed data to *downstream* processes (dashed lines in Figure 2).

5.5 Time-Flow Systems

Throughout this discussion, we are building upon the idea of timed data. If all information is time-stamped and processed in time order, we have a conceptually simple model that leads to specifiable and predictable computation. A key feature of this overall model is that if we know all the input in advance of real time, we can perform the computation immediately. Just as "data-flow" systems permit computation to proceed as soon as input data is available, we coin the term "time-flow" to describe systems where computation can proceed as soon as the next *logical time* of all inputs has been reached, and where logical time can be ahead of real time.

This idea that computation can run ahead of real time is not just an optimization but an absolute requirement in the case of audio samples and video frame buffers – data must be pre-computed and buffered before the output takes place – so the model fits nicely with reality.

6. LEVEL 3 SYNCHRONIZATION

An interesting development in the direction of modular software-intensive audio processing systems is the JACK Audio Connection Kit (www.jackaudio.org), which allows multiple applications in separate address spaces to exchange audio in real time. What are the latency implications of spreading computation across multiple applications? JACK takes the view that latency is most critical, and originally, JACK serialized audio computation. In a pipeline configuration like Figure 1, JACK would send input to stage 1, run it, send stage 1 output to stage 2, run it, send stage 2 output to stage 3, etc. This means that each stage effectively adds no additional buffers or delay to the total audio processing chain.

6.1 Concurrency vs. Latency

Concurrency was added to JACK [12] so that if the signal graph has parallel paths, such as A and B in Figure 2, JACK can deliver samples to both A and B together, allowing them to run in parallel. In principle, parallelism can also be achieved in a pipeline configuration like Figure 1, provided that pipeline stages are working on *different logical times* within the audio stream. This requires additional buffers and increases latency, and therefore is not supported in JACK.

6.2 Allocating Latency

In Level 3 synchronization, we address the problem of trading off concurrency with latency. Unfortunately, a truly independent, modular solution does not seem to exist. Instead, we need some sort of planning and coordination to optimize the "allocation" of latency, in other words the placement of buffers to meet overall latency and performance requirements.

For example, imagine that stages 1 and 2 in Figure 1 run on separate computers. Due to communication costs, it might not even be possible to take the JACK approach of running stages sequentially, waiting to send data to the remote computer, processing the data, and returning it, all before continuing. Instead, we need some buffers to allow computation and communication to run in parallel. The added latency here might preclude additional buffers to allow for concurrency elsewhere in the pipeline.

We define Level 3 synchronization as a network of processes using time-flow techniques – time-stamped data and sample streams that offer precise timing – *with the addition* of global optimization of buffers and processors to find a computationally feasible configuration that either minimizes latency or produces an acceptable latency.

As with all real-time systems, performance guarantees depend upon an accurate characterization of resources and the workload, but often in music and creative applications, information is limited and requirements change rapidly during the creative process. While optimal configuration might not be possible, we can at least build systems that can measure performance, spot the most time-critical elements, and assist us in tuning to get the best performance even if it is not completely predictable.

7. SYNCHRONIZING WITH INPUT

One apparent disadvantage of time-stamp-based synchronization is that, to hide variations in latency, output is intentionally delayed. Almost by definition, input and output cannot be synchronized due to delay. However, time-stamps and synchronization techniques are not the real problem here. Even Level 0, with no time-stamps or explicit synchronization, exhibits delay from input to output. In audio-input-to-audio-output systems, the delay is normally constant, just as in a time-stamped forward-synchronous approach. In event-input-to-event-output systems, the delay is minimized in Level 0 systems by running as fast as possible, but at the expense of greater jitter. In contrast, time-stamped forward-synchronous (Level 1 and 2) systems can reduce jitter through precise timing. However, they can also achieve minimal delay on each event by setting the time advance (L) to zero. Zero time-advance makes time-stamps rather pointless, but by adjusting time-advance upwards, one can achieve a compromise: events that can be delivered within the time advance (L) are delayed, eliminating jitter, while events that are delayed longer are delivered as soon as possible. There is no absolute requirement to eliminate all jitter by adding excessive delay.

What can be done to synchronize output to input? There are at least two possibilities, both forms of prediction. The first is anticipation of musical timing, which is necessary even for humans to perform together. Humans require more than 100ms to react physically to sound [10], so clearly we must anticipate future actions. Similarly, software systems can schedule events in the future based on (1) models of tempo, (2) timed sequences (scores), (3) tracking human gestures, e.g. using position and velocity to predict when a drum stick will hit the drum [11].

The second possibility, introduced in FORMULA, is to "launch" sequences with no time advance. Imagine that you want to press a key to launch a musical sequence in time with a live drummer. (And assume that matching tempo is not a problem.) Ideally, you would just press the key in time with the drummer, and your music would begin with zero latency and be synchronized. In reality, let's say there is a 40ms latency to produce output, so your music will be 40ms late. Within our synchronization framework, if the key is pressed at *t*, we would schedule the output as early as feasible: at *t*+40ms. Alternatively, we could schedule the output at time *t*, in which case the first note would begin late but would start as soon as possible. Subsequent note events in the sequence are scheduled more than 40ms in advance, and therefore the musical sequence will "catch up" and be synchronized.

In practice, none of these approaches offers a full solution, but with time-stamps and the synchronization techniques proposed here, various methods can be combined. For example, we can play fixed sequences or smoothly adjust parameters with enough time advance to mask high latency, and we can simultaneously manage more interactive, low-latency streams, using precise time-stamps to coordinate and synchronize everything.

8. A DEMONSTRATION OF CONCEPTS

To illustrate concepts further, we describe some real examples. In 2016, we created an extensive implementation for an "Internet Drum Circle." Successful group drumming has very low latency requirements compared to wide area network latency, so we accurately record and time-stamp drumming (as events, not as audio), send the data to remote sites, and reproduce the drumming accurately after exactly one cycle (usually 8 beats)[1]. In principle, this approach, similar to [5], can ensure millisecond-level accuracy. It is also a good illustration of a Level 1 time-flow architecture: compute and send data with time-stamps, potentially with high-latency, then render the data precisely according to the time-stamps.

Unfortunately, in our system, rendering took place in a variety of different laptops with different software synthesizers and different latencies, some of which were quite high. This illustrates the shortcomings of Level 1 architectures when working with multiple processing stages. Level 2 offers a solution (although not supported in current applications and APIs): First, the synthesizers should accept not just MIDI events but time-stamped messages.[2] Second, the audio output from the synthesizer should be timed so that the synthesizer can introduce sound events into the audio stream according to *stream logical time* rather than real time to ensure output is heard at the right time. Ultimately the variability in audio latency across different computers interfered with the sense of pulse and entrainment, so we are working on an improved lower latency audio version also using Level 2 concepts.

In a more recent distributed performance implementation, we modified our software to give every laptop a 100ms "rendering latency budget." Even though we cannot know through software and APIs what will be the synthesis latency, we can adjust a slider on the user interface by listening in order to synchronize with other laptops.[3] In this more successful performance, which was based on synchronized compositional algorithms in a laptop orchestra, we used O2 [4] for clock synchronization and communication. A central control "conductor" sent tempo, meter, harmony, style and play/rest parameters to about 20 connected laptops. There, human performers controlled additional parameters interactively as algorithms generated precisely timed MIDI and audio. A video is posted at `https://youtu.be/icLUJMM-11M`.

To summarize, in the distributed system, a clock synchronization protocol gives a common time reference to every machine. Time-stamped control data are delivered to different machines well in advance of the time-stamps. Rather than waiting for the time-stamp to expire, the laptops subtract the synthesis latencies and deliver the data ahead of the time-stamp by that amount. Potentially, every laptop sends information to their synthesizer at a slightly different time to obtain synchronized output. Thus, we achieve end-to-end synchronization through multiple processing stages and across multiple output devices.

9. SUMMARY AND CONCLUSIONS

Current systems for media processing seem to have been developed with hardware and analog models in mind. Rather than treat latency as something to reduce, eliminate or simply ignore, we should treat timing as a fundamental property of our data and manage time, scheduling, and buffering throughout our systems and applications.

To encourage new approaches to timing and synchronization in creative multimedia systems, we describe several levels of sophistication. We define Level 0 as "best effort" low-latency systems that do not keep track of time. Level 1 systems are 2-stage "forward synchronous" systems where control information and media streams are computed with time-stamps that specify the delivery time. Even Level 1 is an advance over most current systems because precise timing can help to coordinate control updates and coordinate the delivery of audio, video, and other media.

Level 2 extends Level 1 by considering multiple stages of processing, addressing the needs of modern software-intensive, modular media processing systems, where multiple applications might be employed together. In Level 2, applications propagate information about latency to both upstream and downstream stages, allowing applications to synchronize and merge multiple input streams and

[1] We make no claims about this as a mode of telematic music performance, and we hope to report on musical and social aspects of this project in the future. For now, we merely present it as a source of technical challenges.

[2] In fact, on Mac OS X, MIDI events *are* time-stamped, and many AU and VST plug-ins interpret them.

[3] This is similar to Ableton's External Instrument Hardware Latency control; see https://help.ableton.com/hc/en-us/articles/209774265-Using-external-hardware-with-Live, except we are synchronizing multiple stages from "conductor" to "performer" to synthesizer.

events, predict the total end-to-end latency, and synchronize multiple media output devices.

Level 3 describes systems that globally optimize and configure in order to obtain adequate performance while minimizing latency (or maximize performance while keeping latency to an acceptable level). We discussed some of the problems and motivation for Level 3 systems, but leave a detailed design to future work.

9.1 What Does This Work Suggest?

One of the main results we offer is the possibility of much better timing and synchronization using a systematic approach. In particular, our Level 2 model offers some very nice properties including:

1. Synchronized delivery of multiple media streams,
2. Modular design supporting inter-operation of applications even across multiple operating systems,
3. Feedback to users/composers/designers about latency and performance,
4. Coordination of timed events with timed media streams as opposed to asynchronous updates.

Achieving all this requires a re-thinking of the API's and models used in current software. There are many indications that software designers are concerned and aware of these problems. For example, the USB protocol for audio offers latency information from hardware to device drivers [9], and JACK allows audio applications to inform JACK if they insert delay into the audio stream.

9.2 Future Work

While this paper has discussed principles and architectural considerations, there is much more work to be done. One way to proceed would be to create an experimental system embodying these concepts and demonstrating end-to-end synchronization across media and applications. The lessons learned could be useful in creating future Audio API's.

It would also be interesting to see how some existing APIs such as Core Audio, JACK, PortAudio, PortMidi, etc. could be extended to support time-flow systems.

Another concern is how will programmers build real systems around time flow? It is a lot to ask a DSP coder to correctly perform scheduling, manage time-stamps and track logical time in a time-flow network. However, languages like FORMULA and ChucK already encapsulate explicit timing into very usable computational frameworks, as have many toolkits, so it should be possible to integrate a more complete Level 2 approach into existing software frameworks.

These concepts also have implications for streaming protocols and web delivery, where time-stamps and time flow might help with synchronization, especially as audio, MIDI, and video begin to be incorporated as both inputs and outputs in web applications.

Acknowledgments

I would like to thank reviewers for their helpful comments, the School of Computer Science at Carnegie Mellon for its support, and my spring 2017 Computer Music Systems and Information Processing class for their implementation and participation in the laptop orchestra performance described above.

10. REFERENCES

[1] G. Blakowsky and R. Steinmetz, "A Media Synchronization Survey: Reference Model, Specification, and Case Studies," *IEEE Journal on Selected Areas in Communications*, vol. 14, no. 1, 1996, pp. 5-35.

[2] D. P. Anderson and R. Kuivila, "A system for computer music performance," *ACM Transactions on Computer Systems*, vol. 8, no. 1, pp. 56-82, 1990.

[3] E. Brandt and R. B. Dannenberg, "Time in Distributed Real-Time Systems," *Proceedings of the 1999 International Computer Music Conference*, Beijing, 1999, pp. 523-526.

[4] R. B. Dannenberg and Z. Chi, "O2: Rethinking Open Sound Control," in *Proceedings of the 42nd International Computer Music Conference*, Utrecht, September 2016, pp. 493-496.

[5] M. Goto, I. Hidaka, H. Matsumoto, Y. Kuroda, and Y. Muraoka, "A Jazz Session System for Interplay among All Players–VirJa Session (Virtual Jazz Session System)," *Proceedings of the 1996 International Computer Music Conference*, pp.346-349, August 1996.

[6] D. Liang, G. Xia, and R. B. Dannenberg, "A Framework for Coordination and Synchronization of Media," *Proceedings of the International Conference on New Interfaces for Musical Expression*, Oslo, Norway, 2011, pp. 167-172.

[7] G. Wang, P. Cook and S. Salazar, "ChucK: A Strongly Timed Computer Music Language," *Computer Music Journal,* vol. 39, no. 4, 2015, pp. 10-29.

[8] M. Wright, A. Freed and A. Momeni, "OpenSound Control: State of the Art 2003," *Proceedings of the 2003 Conference on New Interfaces for Musical Expression (NIME-03)*, Montreal, 2003, pp. 153-159.

[9] USB Implementers Forum, "USB Device Class Definition for Audio Devices, Release 1.0," http://www.usb.org/developers/docs/devclass_docs/, 1997.

[10] A. Welford (Ed.), *Reaction Times*. Academic Press, New York, 1980.

[11] M. Benning, M. McGuire and P. Driessen, "Improved Position Tracking of a 3-D Gesture-Based Musical Controller Using a Kalman Filter," *Proceedings of the 2007 Conference on New Interfaces for Musical Expression (NIME07)*, New York, 2007, pp 334-337.

[12] S. Letz, D. Fober, Y. Orlarey, "Jack Audio Server For Multi-Processor Machines," *Proceedings of the International Computer Music Conference*, Barcelona, 2005.