A System Supporting Flexible Distributed Real-Time Music Processing

Roger B. Dannenberg and Patrick van de Lageweg

School of Computer Science, Carnegie Mellon University
email: dannenberg@cs.cmu.edu

Abstract

Local-area networks offer a means to interconnect personal computers to achieve more processing, input, and output for music and multimedia performances. The distributed, real-time object system, Aura, offers a carefully designed architecture for distributed real-time processing. In contrast to streaming audio or MIDI-over-LAN systems, Aura offers a general real-time message system capable of transporting audio, MIDI, or any other data between objects, regardless of whether objects are located in the same process or on different machines. Measurements of audio synthesis and transmission to another computer demonstrate about 20ms of latency. Practical experience with protocols, system timing, scheduling, and synchronization are discussed.

1 Introduction

Local-area networks offer a fast, general, reliable way to interconnect personal computers. Unlike MIDI or AES-EBU, which require special interfaces and are restricted to a single format, networks communicate any digital data, and high data rates are available. Many other researchers have used local and wide-area networks for music data, including MIDI and audio (see references). Most of this work has focused on point-to-point transmission only. Open Sound Control (Wright and Freed 1997), for example, is a network-oriented protocol for controlling synthesis engines. We introduce a system, *Aura*, designed to take advantage of general-purpose PCs and ordinary low-cost local area networks to implement powerful real-time music programs.

The main thrust and feature of this work is a software foundation that supports networking and distributed real-time processing from the ground up. This means more than simply providing a real-time transport from one machine to another or accommodating music data types such as MIDI and digital audio over Ethernet. Imagine a truly connected system where sensors are in two-way communication with decision-making software, where physical models and compositional algorithms engage in a dialog to generate music, and where graphical virtual worlds exchange information with sound worlds to create a coherent and synchronized multimedia presentation. These sorts of applications require a flexible approach where software objects can communicate easily, regardless of their function and location. Ideally, there should be just one communication mechanism, whether objects are located together or on separate computers. This property, called *transparency*, enables systems to be reconfigured without rethinking the entire communication structure and possibly recoding objects to use different communication mechanisms. All this must be done in a way that enables low-latency, real-time performance. If communication even occasionally blocks computation for just a few milliseconds, the system will be limited in some applications.

Aura differs from other systems in some important regards. First, Aura manages a collection of objects that communicate over logical connections. There may be hundreds or thousands of objects and connections in an Aura system, not just a small number of point-to-point virtual MIDI or audio channels. Second, Aura imposes very little structure on messages and objects. Simple data in the form of numbers, strings, and object names are sent across connections to control sound, graphics, devices, and computation. More structured data, including buffers of audio samples and MIDI messages can also be sent. Third, all objects communicate in the same way, so network communication is simply the consequence of sending any message to a remote object.

Because Aura has been described previously in terms of objects and messages (Dannenberg and Rubine 1995, Dannenberg and Brandt 1996), this paper focuses on extensions to provide communication over local area networks. The mechanisms described here are based on Aura, but the general techniques and results should be applicable to other real-time systems.

2 Spaces, Zones, Objects, and Names

Spaces. Aura's abstraction for a machine is the *Space*, short for Address Space, representing a shared address space with multiple threads.

Zones. Aura computation takes place in zones. A zone consists of a single thread and a collection of objects that share the thread. (See Figure 1.) Because the thread is shared, objects cannot execute long loops or suspend; however, they can easily defer computation by sending

Roger B. Dannenberg and Patrick van de Lageweg, "A System Supporting Flexible Distributed Real-Time Music Processing." In *Proceedings of the 2001 International Computer Music Conference.* San Francisco: International Computer Music Association, ISBN 0-9713192-0-0, 2001, pp. 267-270.

themselves a timed message. As long as objects do not compute for too long, other objects in the same zone will be able to meet their real-time requirements.

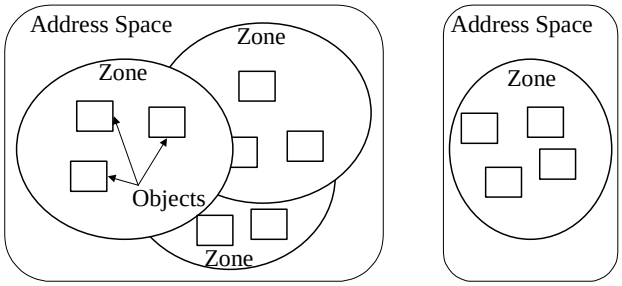

Figure 1. Aura system with two address spaces (on separate machines), each with zones containing objects.

Typically, objects are assigned to zones according to real-time requirements: low-latency audio computation goes in one zone, music control computation goes in another, and graphics and user interfaces go in a third, for example. By using several zones, time-critical zones can achieve low latency by preempting other zones, and long-running computation can run in lower-priority zones without blocking time-critical audio computation.

Objects. Aura objects send and receive asynchronous messages, perform computation in response to a message, and send messages. Objects are based on C++ objects, which are extended by the Aura preprocessor with symbol-table information. The typical Aura message sets an attribute to some value. The default message handler looks up the attribute in a symbol table, maps the attribute to the location of a corresponding C++ member variable, and sets the variable to the value carried in the message.

Aura applications are created by interconnecting objects. Objects have input and output ports, inspired by familiar MIDI and A/V components, by which objects are interconnected. Since interconnections are external to objects, collections of objects can be reconfigured to different locations without changing code within the objects.

Names. In most programming languages, objects are referenced by their address in memory, but an address is not sufficient to designate an object when there are multiple address spaces. Therefore, Aura uses 64-bit integers to provide globally unique names for objects. An object name tells where to deliver a message; in fact, the high-order 16 bits encode the object's address space number and zone.

Because Aura is a real-time system, it is important for an object in process *A* to be able to create an object in process *B* without actually waiting on *B*. In practice, this requires that process *A* can create the unique name for the new object in *B* without any knowledge of the memory configuration of *B* and without consulting a centralized name resource. To accomplish this, *A* combines its own address space and zone (16 bits) with a locally unique 32-bit sequence number to form a globally unique 48-bit identifier (see Figure 2). Using this scheme, *any zone can generate a globally unique name for a new object anywhere, in constant time.*

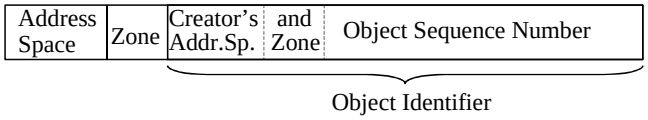

Figure 2. Aura names are globally unique object references. *Address Space:* small integer corresponding to a computer. *Zone:* one of many zones in the Address Space. *Object Identifier:* names an object within a Zone. (Object address is obtained from a per-zone hash table.)

3 Message Delivery

To make a connection from one object to another, the 64-bit name of the receiver object is placed on the *receivers list* of the sender object. When an object sends a message, the Aura runtime system delivers a copy of the message to every object on the list. The first step is to compare the address space and zone bits of the receiver's name to those of the sender. If they match, the message is destined for the same zone: A hash lookup is performed to find the address of the receiver, and a receive method is called to deliver the message immediately (and synchronously) to avoid copying overhead. If the message differs in the zone bits but not the address space, the message is copied to a shared-memory FIFO queue that is read by the receiver's zone. The receiver's zone thread polls the queue, removing and delivering messages to local objects.

If the address space bits do not match, the message is delivered to a special proxy object found in a table indexed by the address space bits. The proxy object forwards the message over a network to a companion object in the receiving address space. The message is then delivered using the local delivery mechanisms presented above. This design allows new proxy objects to be plugged in to support any interprocess communication mechanism desired.

The pseudo-code in Figure 3 illustrates the message send procedure, although it omits the handling of timestamps. In the full implementation, messages with future timestamps are held in a priority queue in the destination zone and delivered at the designated time. Messages to proxies are "wrapped" so that they are delivered immediately, and then "unwrapped" so that the final delivery obeys the timestamp.

```
function SendTo(dest, msg):
  if (dest.addr_space != my_addr_space):
    SendTo(addr_space_proxy[dest.addr_space],
           Wrap(dest, msg))
  elif (dest.zone != self.zone): //interzone
    out_queue[dest.zone].enqueue(dest, msg)
  else: //intrazone
    target = hash_lookup(dest.object_id)
    target.receive(message)
```

Figure 3. Message delivery pseudo-code. Inter-address-space messages are sent to a (local) proxy object for forwarding; inter-zone messages are delivered via a shared-memory FIFO queue, and local messages are delivered by invoking the receive method of the target.

4 Distributed Operation

We have implemented several versions of this system, and there have been at least a few interesting and somewhat surprising results, which are described below.

4.1 Timing Issues Under Linux

The Linux kernel normally uses a 100 Hz clock for scheduling, which makes it difficult for a process to perform timed actions with better than 10ms of precision. In Aura, we can get around this limitation by using the audio device to wake up the highest priority audio zone every 32 sample frames. This zone, in turn, can signal other zones at lower priority so that they wake up frequently. It should also be possible to recompile the kernel with a different HZ setting, but our approach works with unmodified kernels.

4.2 UDP vs. TCP

In our first implementation, we used UDP (Comer 2000), a simple network protocol, to transmit data over the network. UDP was chosen because it seemed to be the most suitable protocol for a real-time system. Also, UDP has been used in other real-time MIDI and control systems reported in the literature. (Goto, Neyama, and Muraoka, 1997) UDP has the potential drawback that message delivery is not guaranteed, but previous studies have found UDP to be reliable across local area networks in controlled situations. In our case, perhaps because there were multiple machines transmitting messages in a less orderly fashion, we observed dropped UDP packets. This makes UDP unusable without additional protocols to retransmit lost packets. For this reason, we switched to TCP/IP (Comer 2000), a reliable protocol.

TCP/IP is said to have problems in real-time systems because of its buffering and retransmission policies. We ran into just one difficulty. The default behavior of TCP is to attempt to merge messages into network packets to achieve greater efficiency. This can be avoided using the `TCP_NODELAY` option, which eliminates merging and delays, but increases the number of messages. With this option, the timing behaviors of TCP and UDP appear to be identical. Of course, if a packet is lost, TCP will stop and recover, causing a delay, while UDP will simply lose the message. To regain some of the lost efficiency due to `TCP_NODELAY`, we send audio in packages of 320 samples even though audio is normally computed in 32-sample blocks. It might also be a good idea to merge messages at the Aura level, before sending them to the operating system, to avoid too many short messages. (Fober 1994)

4.3 Global Clocks and Time Management

All messages carry timestamps, and messages destined for remote zones are delivered to the zone as soon as possible. They are then held locally and delivered to the object within the zone at the designated time. As described by Brandt and Dannenberg (1999), clocks are synchronized so that timestamps are consistent across the network. This allows messages to be delivered with precise timing if they are computed early, and otherwise messages are delivered as soon as possible.

This earlier work does not consider the case of delivering audio across a network. In our current implementation, we designate a *master* system with an audio interface assumed to be running at exactly 44.1kHz (or any other standard rate), and one or more *slave* systems, whose audio sample rates may drift above or below the master rate. The slave systems can be configured in two ways: (1) Audio computation is synchronized to the local DAC, but time is synchronized to the master. This allows the slave to operate more or less like a MIDI synthesizer. Control information is passed asynchronously via Aura messages and audio is converted to analog locally. (2) Audio is synchronized to the global clock. Since the global clock is synchronized to the master's DAC, audio may be returned to the master, mixed with other audio, and output to the master DAC. Audio computation can be sample-accurate, although some latency will be incurred as audio is shipped over the network. Without option (2), it is impossible to stream audio without either resampling or adding additional synchronization hardware.

4.4 The Virtual Patchbay

Because processes do not initialize simultaneously, it is necessary to wait for all Aura processes to start running before creating a distributed set of objects. Furthermore, if an Aura process crashes, it may leave objects with "dangling" references to non-existent objects. In many cases, it would be nice if the system could automatically make connections between objects as they are created, and disconnect "dead" objects when their process is terminated.

A solution to this problem is the *virtual patchbay*, a replicated distributed database of virtual patch points, which are simply named by text strings. The virtual patchbay essentially follows the "publish/subscribe" metaphor. Programs can issue requests to connect from an object to a patch point and from a patch point to an object. If the source for a patch point does not exist at the time of a request, the request is saved in the distributed database until a source becomes available. When a process terminates, address space proxy objects notify their local virtual patchbay, which releases the affected connections.

With this facility, object connections can be made asynchronously and systems can become more fault-tolerant.

5 Current Status

The distributed Aura system is now in its third generation, and development is focused on Linux, where we have run with audio buffers as short as a few milliseconds, even with substantial network loads. This is possible partly

because of the multithreaded structure of Aura, which in turn is enabled by the distributed object model. We can create and join multiple instances of Aura on different processors, create objects remotely, and connect objects across address spaces. We constructed new audio "patch" objects that stream audio packets from one machine to another for further processing or playback.

We are currently modifying the system to make configuration and reconfiguration simpler. The idea is use logical names for zones and to map logical names to actual zones at runtime according to a configuration description. For example, one might have logical zones named AudioOutput and AudioDSP which might be the same on a single processor system for debugging but separated in a distributed production version. Also, the virtual patchbay is designed, but not yet implemented.

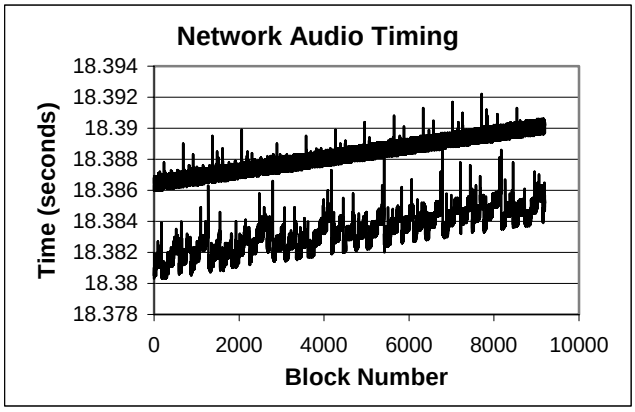

Figure 3. Arrival time of audio from network (lower curve) and the time data is written to audio device (top).

For convenience, we have been running on Linux machines without low-latency kernel patches using 10MHz Ethernet. Figure 3 shows timing measurements at the receiver of an audio stream. The lower curve is the time at which a 320-sample message arrives, and the upper curve is the time at which the first sample of that message is written to the audio device driver. The horizontal axis is the message number, and the vertical axis is the difference between the measured time and "nominal time" (sample number / sample rate). We subtract nominal time to emphasize timing jitter over absolute time. The overall slope is due to drift between the system clock and the DAC clock. The lines are parallel in the long term because of clock synchronization on the two machines. The short-term variations in the lower curve show the combined effects of operating system scheduling, network latency, and internal buffering. Overall, Aura delivers messages from source to sink in about 2 to 6 ms, with an additional few ms of jitter due to the operating system. In addition, the source buffers 10 blocks of audio, an additional 7ms (not shown), before sending a message over the network. Allowing 5ms to get audio through the device driver, this gives a total of around 20 ms from the time a sample is computed on one machine to the delivery of analog output at the other machine. Future work will explore the use of fast Ethernet and other network media to reduce this latency further. It should also be noted that additional buffering might be necessary to recover from packet loss (although none was observed in these simple tests). Network utilization is assumed to be low, and Aura uses one priority for all network traffic. More sophistication would be required for high network load factors.

6 Conclusions

We find Aura to have several distinct advantages over other software systems for interactive music: Aura messages are open-ended, with no particular built-in notions of instruments, voices, etc., allowing Aura to reflect the needs of the application. Aura object location is largely transparent to the application, so no difficult steps are required to reconfigure a program to run on a network of computers. Finally, transparency and network-wide naming allows new connections between objects to be created as needed without rethinking or violating any formal system structure. These properties seem to support the creation of interesting interactive systems for music.

Latency, particularly for audio connections, is still an important concern. One of the advantages of Aura is that the architecture hides the network details from the application. Since networking is implemented using ordinary Aura objects, it will be easy to incorporate networking improvements into Aura or to experiment with different networking strategies.

I am very grateful to IBM Research and their Computer Music Center for financial, technical, and moral support and many enthusiastic, stimulating conversations.

References

Brandt, E., and R. B. Dannenberg. 1999. "Time in Distributed Real-Time Systems." *Proceedings of the 1999 International Computer Music Conference*. ICMA, pp. 523-6.

Comer, D. 2000. *Internetworking with TCP/IP Vol. I: Principles, Protocols, and Architecture.* Prentice Hall.

Dannenberg, R. B. and E. Brandt. 1996. "A Flexible Real-Time Software Synthesis System," *Proceedings of the 1996 International Computer Music Conference*, ICMA, pp. 270-3.

Dannenberg, R. B. and D. Rubine. 1995. "Toward Modular, Portable, Real-Time Software." *Proceedings of the 1995 International Computer Music Conference*, ICMA, pp. 65-72.

Fober, D. 1994. "Real-time Midi data flow on Ethernet and the software architecture of MidiShare." *Proceedings of the 1994 International Computer Music Conference.* ICMA, pp. 447-50.

Goto, M., R. Neyama, and Y. Muraoka. 1997. "RMCP: Remote Music Control Protocol—Design and Applications", *Proceedings of the 1997 ICMC*, ICMA, pp.446-449.

Wright, M. and A. Freed. "OpenSound Control: A New Protocol for Communicating with Sound Synthesizers." *Proceedings of the 1997 ICMC.* ICMA, pp 101-104.

Young, J. P., and I. Fujinaga. 1999. "Piano Master Classes via the Internet." *Proceedings of the 1999 International Computer Music Conference*, ICMA, pp. 135-137.