Aura as a Platform for Distributed Sensing and Control

Roger B. Dannenberg
Carnegie Mellon University
School of Computer Science
Pittsburgh, PA 15213, USA
1-412-268-3827

rbd@cs.cmu.edu

ABSTRACT

Aura is an evolving software architecture and "real-time middleware" implementation that has been in use since 1994. As an integrated solution to many problems encountered in the design of distributed, real-time, interactive, multimedia programs, experience with Aura offers lessons for designers. By identifying common problems and evaluating how different systems solve them, we hope to learn how to design better systems in the future. Aspects of the Aura design considered here include message passing, how objects are interconnected, the avoidance of shared memory, the grouping of tasks and objects according to latency requirements, networking and communication issues, debugging, and the scripting language.

Keywords

Aura, Languages, Real-Time Systems, Architecture, Networks

1. INTRODUCTION

Aura is a software platform for building sophisticated applications involving real-time, concurrent activity, multiple sensors, and multiple modes of output. Aura supports distributed systems using local-area and wide-area networks. Although these requirements may appear to be "pie-in-the-sky" desiderata, Aura was designed to overcome real limitations encountered in previous systems. Aura has achieved all of these goals in real applications.

Aura is essentially "real-time middleware" that includes a distributed object system. As such, Aura has many similarities and parallels to systems such as CORBA, DCOM, constraint systems, cooperative multitasking systems used for graphical user interfaces, and publish/subscribe systems. In general, Aura is intended for single applications running on dedicated, reliable hardware as opposed to shared services in an unreliable, distributed network.

1.1 Complications in the Design of Sensing and Control Systems

It is not simple to describe Aura because there are many seemingly disparate factors that constrain its design. Perhaps the most important message of this paper is that sensing and control systems are more complex than they appear on the surface. Many factors enter into the design, and there are many subtle interactions between different design goals.

To give just one example, portability requires that designs avoid making too many assumptions about the underlying operating system. Many operating systems do not handle priority inversion, which means that something as simple as a critical section protected by semaphores can cause real-time failures. Aura's solution is to avoid semaphores completely, and this decision has ramifications for memory allocation and communication mechanisms.

Different designs will place more emphasis on different problems or on solving different classes of applications. This tends to lead to many different approaches. Since we have few if any ways to evaluate designs of complex systems objectively, the best we can do is to gather experience using various designs and report on how well they work in practice. Problems and successes relating to different design decisions can guide future work. Throughout this paper, I will try to explain some of the key issues in distributed, interactive, real-time systems and how they are solved in Aura.

The focus of this paper is on lessons learned. The question is not so much whether Aura is good or bad, but what can we learn from it? I am especially interested in making complex interactive systems easier to build, especially to support creative work in the performing arts. As technical requirements increase to require distributed systems, low-latency, and a variety of media, the implementation task can become daunting, yet resources for the performing arts are always very limited. The typical big-system software engineering approach is not an option. Aura must make application building seem more like building simple algorithms in a high-level scripting language and less like low-level real-time systems programming. Some of the features and approaches of Aura are listed in the sections that follow. I present a critical assessment of how well these features work in practice, based on experience building several applications with Aura.

2. MESSAGE-BASED COMMUNICATION

Perhaps the main feature of Aura is the use of "real" messages for communication between objects. By "real," I mean that messages exist as data objects as opposed to SmallTalk-style messages that are essentially stack-based function calls. [9]

To send a message, the sender calls

```
send(msg_ptr, len, timestamp)
```

which sends an arbitrary message. The timestamp is used to determine when the message is delivered. Notice that the message destination is not specified (see Section 3). The most common

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

Originally published as: Roger B. Dannenberg, "Aura as a Platform for Distributed Sensing and Control." In *Symposium on Sensing and Input for Media-Centric Systems (SIMS 02)*, Santa Barbara: University of California Santa Barbara Center for Research in Electronic Art Technology, (2002), pp. 49-57.

message is a “set” message that sets the value of an attribute in the receiver. Although `send` can be used, it is convenient to call one of the `set` functions, e.g.

```
set_int(attribute, value, timestamp)
```

sets an integer attribute to an integer value at the specified time. (Attributes in Aura are globally unique 32-bit identifiers managed by a preprocessor.)

Interactive music and animation programs tend to have many parameters that can be adjusted. The output and behavior of these programs is determined by artistic choices rather than by a formal specification that allows only one correct answer. Therefore, it is common to make many adjustments after the program is running correctly. Providing convenient access to these parameters is an important goal of the Aura design.

It is convenient to store parameters as values within objects and to express object behavior in terms of these values (instance variables). In typical object-oriented languages, values are hidden and can be accessed only by invoking methods. In Aura, values are directly accessible via `set` messages. Thus, it is easy to control the behavior of objects and to adjust their parameter values interactively by sending `set` messages.

In practice, this works quite well. The programmer only writes a simple comment to make a C++ member variable “settable” by Aura messages. This is a big advantage over the precursor to Aura, the CMU MIDI Toolkit [5], where many lines of code were typically required to support parameter updates. In addition, messages offer a single real-time mechanism that works well for: (1) local communication within a single thread, (2) asynchronous communication between threads, and (3) network communication. This is a drastic simplification over typical program organizations where multiple asynchronous threads must communicate and synchronize using carefully designed protocols.

3. EXTERNAL CONNECTIONS

The Aura design evolved from earlier work on constraint systems. [13] Since global constraint solvers are not consistent with real-time performance in a distributed system, we tried to simplify, first with one-way equality constraints between attributes, and then by expressing constraints as connections between objects. Conceptually, when an object changes its state, it sends the state change as a `set` message. To constrain another attribute to this value, one makes a connection to the dependent object. For example,

```
connect_attributes(from,fromattr,to,toattr)
```

connects the `fromattr` attribute of object `from` to the `toattr` attribute of object `to`. Thus, connections between objects are managed exernally to the objects. This is quite different from standard object-oriented systems, where the sender must have a reference to the receiving object in order to send a message.

Although Aura was inspired by constraint systems, we realized later that we had come quite close to reinventing MAX semantics. [16] Objects can be “wired together” more like stereo components than typical programming objects. This facilitates rapid prototyping and end-user programming, but the hidden state in the form of connections can lead to obfuscation.

Different programmers may have different opinions about this, but my belief is that the important concept here is *visual continuity in the representation of program flow*. To illustrate this, consider the visual program in Figure 1.

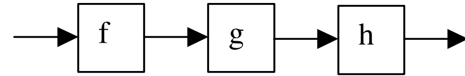

Figure 1. A simple visual programming example.

This is simple to read and understand. A direct translation into text might be:

```
a = new f; b = new g; c = new h;
connect(input, a); connect(a, b);
connect(b, c); connect(c, output);
```

I believe this is hard to read because the functions, as implemented by `f`, `g`, and `h`, and their connections, as specified by `connect` functions, are visually discontinuous. The following is much easier to read:

```
output(h(g(f(input))));
```

not so much because the program is shorter but because it is easy to tell how the input is operated on step-by-step to form the output. Thus, a programming style that seems to work well in a visual programming language like MAX may be inappropriate in a textual programming language like C++ or Aura even if the two are semantically the same.

This aspect of Aura’s design is still being explored. One possibility is a visual interface to make connections easier to manage, but visual languages tend to lead to static configurations, whereas a strength of Aura is the ability to dynamically change connections and to create new objects on the fly. Alternatively, graphical debuggers might display connections visually at runtime.

The other direction is to move away from these constraint-like connections. Aura already offers a variation on the `set` functions where the caller specifies the destination. This is often used, even though it makes it difficult to tap into message streams, e.g. to display them or save them to a log file for debugging. Recently, I developed a way to specify audio “instruments” using a very readable functional style that automatically constructs objects and their connections. [3]

4. NO SHARED MEMORY

Shared memory refers to memory regions that can be accessed by more than one thread. Although shared memory is used in the Aura implementation for efficiency, the use of shared memory at the application level is strongly discouraged in Aura. Instead, shared state is copied via messages as needed. In my opinion, this approach is critical for simplifying programs and for program reliability.

In a more traditional system, multiple threads would have free access to many shared objects. This creates the danger that two threads will try to execute within the same object at the same time, causing various kinds of race conditions and unpredictable behavior. To prevent these problems, objects must place locks around code that accesses data, preventing concurrent access. In some cases, objects limit execution to a single thread at a time. These objects are commonly called *monitors*. [8]

Locks and concurrency are difficult for programmers. If you forget to use locks, data values may be changed at unexpected times and lead to errors. If you forget to release a lock, all threads

may block forever waiting to gain access to an object. Interactions between locks can lead to deadlock, where all threads are waiting on one another in a set of circular dependencies. If a high priority thread is blocked waiting for a low-priority thread, real-time performance may degrade in ways that cannot be reproduced. Locks must be implemented by operating systems in order to solve the priority inversion problem that can be encountered in a real-time system. [19] Not all operating systems prevent priority inversion, and even if they do, the overhead of making a system call for every lock and unlock operation is considerable.

Aura avoids all of this by never exposing shared memory to the application and by never using locks. Instead, memory is divided into *zones*, each of which is exclusively managed by a single thread. Objects are allocated within a zone and therefore an object's methods can only be executed by a single thread, hence no locks are needed. To communicate, objects send messages to another zone. Incoming messages are delivered and processed one-at-a-time, eliminating any concurrency within a zone. Messages are delivered via single-writer/single-reader queues, which can be implemented correctly without locks or system calls.

This approach, which works in conjunction with the Aura object model and Aura messages, is probably the single most important feature of Aura supporting and simplifying programming. These techniques are often used in other applications, especially to connect a graphical user interface to a real-time "back end" [20], but Aura goes further by offering a general-purpose run-time structure supporting this programming model.

Because there are at most a few fixed threads, debugging is simplified. Also, because most bugs occur in the context of a few objects running in a single zone, eliminating a real-time Aura bug is often no more difficult than finding a bug in a single-threaded, non-real-time program. In my experience, the most common difficult bugs in concurrent systems have to do with synchronization and communication. Aura practically eliminates all of these bugs. Of course, there are sometimes timing-dependent bugs or strange interactions between objects running in different threads. In practice, these are quite rare.

5. LATENCY-BASED RATHER THAN TASK-BASED THREADS

In Aura, objects are allocated to different threads (called *zones*) according to their real-time requirements. A time-critical audio object will reside in a high-priority zone, while a graphical user interface object will reside in a low-priority zone. This approach minimizes the number of threads, stacks, and task switches, resulting in great efficiency and simple structure.

Thus, rather than arbitrarily deciding that the user interface runs in one thread and all "application" code runs in another thread, Aura objects can be distributed across many zones/threads according to their real-time requirements. A good example is running OpenGL-based animation and file I/O in the same zone as the user interface because the desired latencies are all on the same order (tens of milliseconds).

Another example of the use of zones is that, commonly, MIDI processing and "control" processing is performed in one zone, while audio processing is performed in another. This means that compositional algorithms that generate notes and control-rate updates may occasionally compute for 10 or 20ms because these delays will not be noticeable. Meanwhile, audio processing takes place at higher priority and runs with lower latency and hard deadlines. If these tasks were merged into a single thread (as in various implementations of MAX with sound extensions [15]), then either the MIDI and control processing would have to be optimized to lower the worst-case latency, or audio buffering would need to be added to prevent audio buffer underflow. In the latter case, the audio latency would suffer.

After working with this zone structure, I can report that it works very well. In one composition, Aura was able to devote most of the processor to animations that took nearly 40ms to compute. At the same time, it ran interactive MIDI algorithms with typical response times less than 3ms. More recent work with audio has demonstrated audio latency in Aura below 10ms. This was achieved under Linux, with real-time kernel modifications [10].

6. NETWORKING

Aura objects can reside on different machines. They use the same basic message-passing mechanisms to communicate between machines as within a single machine. Important network issues that Aura does not address are efficient utilization of bandwidth and latency-sensitive congestion control.

6.1 Networks and Timing

One of the important simplifications in programming supported by Aura is explicit and precise timing. Aura uses double-precision floating-point numbers to represent time, and messages are scheduled for delivery at a particular time. Using timestamps, the order of message delivery can be precisely controlled, and often, timing can substitute for intricate synchronization. For example, if it is important to set audio attenuation before generating a sound, the attenuation can be set with a slightly earlier timestamp than the message that triggers sound generation.

To get the same advantages using networks, clocks must be synchronized. Aura clock objects adjust the local time to track that of the "master" Aura system. Given that all machines have stable crystal clock hardware, it is simple to maintain synchronization within a few milliseconds. Algorithms for even tighter synchronization have been simulated but not yet implemented. [2]

Synchronized clocks and accurate scheduling offers a tradeoff between latency and jitter. In the low-latency case, the goal is to deliver information from one machine to another as soon as possible. This is accomplished by sending a message for immediate delivery. The message will arrive quickly, but the exact delay is unpredictable. In the low-jitter case, a message is sent with a future timestamp that is greater than the expected delivery time. The message arrives early and is delivered according to the timestamp, making delivery times very predictable, but at the cost of added delay (latency). We call this the forward synchronous model. [2]

One standard trick developed by the computer music community is to base timestamps on logical time, the time at which an event is scheduled to execute, rather than physical time, the time on the local clock. The idea is that if an event is scheduled for time t, then output generated by the event should be scheduled for time $t+\delta$, where δ is a constant representing latency. Suppose the system has fallen slightly behind schedule so that the event actually executes at time $t+\varepsilon$ and the output command is delivered across a network at $t+\varepsilon+\lambda$. As long as $\varepsilon+\lambda < \delta$, the output

command will be executed at precisely $t+\delta$; thus, the timestamp masks the variation in timing caused by computation time and message delivery time. Some systems, notably Formula [1], automatically delay output to $t+\delta$, which simplifies the programming model. In Aura, since messages do not flow strictly from input to output, adding a delay of δ to all messages is very confusing. If jitter-reducing delay is desired, the user must decide where this should occur and add the delay explicitly.

6.2 Network Transparency

One of the advantages of Aura's design is that object location is largely transparent to the programmer. Two objects on separate machines can be connected just as two objects running in the same zone. Of course, there may be timing and performance differences, but most algorithms continue to function correctly.

This opens the possibility that programs can be first debugged and tested on a single machine, even in a single zone, and then later distributed over multiple machines. Distributed systems might achieve better performance or gain access to additional sensors, screens, audio interfaces, and other resources.

In its present form, Aura is not as easy to reconfigure as it should be. Device interfaces exist in particular zones, and objects are allocated in specific zones. It would make more sense to denote zones by symbolic names and to make a compile time or even load time association between symbolic zones and actual zones. Then, some sort of configuration file could be used to decide how objects are to be distributed among various machines and threads.

6.3 Real-Time Communication

Aura assumes communication is inexpensive and does not need to be scheduled. In reality, if there is network contention, we might want the threads that are executed at high priority to get priority on the network as well. In the current implementation, messages do not carry priority (which would be specified in terms of allowable latency), so all messages receive equal treatment.

Messages are sent immediately without consolidating them into a single large network packet, even though this would be more efficient in terms of the number of network packets sent. The simple networking model is simple to reason about, but not difficult to overload with too many messages. Ethernet in particular has a minimum message size, and tests indicate that problems occur when we try to send more than a few hundred messages per second using 10Mbps Ethernet.

Using off-the-shelf networks and software for distributed real-time control is a relatively unexplored area, so further work is needed to improve performance, especially throughput of small messages. The obvious approach is to batch short messages into network messages that depart on a periodic basis. This adds latency and jitter but improves network throughput. When different interconnects are available (Ethernet, Firewire, MIDI, etc.), network routing and scheduling become more complex.

7. DEBUGGING

There are many issues related to debugging. What new mechanisms are required? How do we observe or log real-time behavior? How do we verify or even observe that desired distributed run-time structures are created as intended? These are some of the problems that cannot be solved easily by traditional debuggers. Aura offers some interesting debugging features, including the observability of message streams, built-in per-object debugging state, and "real-time" print statements.

7.1 Observing Messages and Objects

Typical Aura objects exchange information via messages. Because connections can be created at runtime, it is simple to connect any object to a `Trace` object, which prints messages as they arrive. This helps the programmer to insert probes into a running system to verify that proper messages are being sent. Alternatively, messages can be directed to a logging object that converts one or more message streams into a text log file and writes the data to disk. By placing the logging object in a low-priority zone, logging can take place without affecting with low-latency computation.

Another debugging tool built upon Aura's message system allows users to dump a snapshot of an object's state for inspection. Objects respond to the `AURA_MSG_REQUEST_SET_ALL` message by sending the object's state in the form of `set` messages to a designated receiver. This mechanism provides access even to remote objects with minimal disruption to real-time execution because formatting and display typically take place in a low-priority zone.

By default, all objects implement an integer attribute called `debug_level`, and standard macros help programmers add debugging print statements that can be enabled or disabled at runtime via Aura `set` messages. In C++, the object `cout` normally designates standard output to the console. In Aura, text directed to `cout` is converted into set messages and delivered to the user interface zone. This provides a low-overhead, real-time "print" function that can be used even in time-critical computations.

7.2 Observing Configurations

As mentioned in Section 3, a key to debugging and simply understanding programs is a representation where program flow is easily to visualized. Unfortunately, a dynamically configured network of objects distributed over multiple zones and machines is not clearly specified by typical Aura code. I hope to experiment with debugging tools that can construct and even manipulate connections visually. Additional work is needed to develop better ways to express and observe configurations.

8. SERPENT AND C++

Originally, Aura was designed and implemented in C++. Since performance is often critical in real-time systems, this was a logical choice because C++ offers both high performance and a fairly high-level, expressive language. In addition, using C++ (or some other existing language) allows us to build upon existing compilers, debuggers, and programming environments.

We found that C++ worked well for most of Aura and for many applications built with Aura. Problems arose, however, using C++ to define clusters of objects for audio processing. It seems that DSP algorithms are best defined using a functional style of programming and that functions should be polymorphic to allow a mix of scalars and signals. For example, to multiply `a` and `b`, it is awkward to write:

```
multiplier = new Multiply
multiplier.in1 = a
multiplier.in2 = b
```

It is much easier to write `multiply(a, b)`. Now, `a` and `b` can denote either streams of audio rate samples, streams of control rate samples, or scalar (constant) values, so `multiply` should be

polymorphic. This can probably be achieved in C++, but it is awkward, and since these functions typically create structures to process streams over time, there are difficult memory allocation and garbage collection issues to solve. (Freed [6] discusses advanced C++ programming techniques for DSP.)

To overcome these problems, I designed a new language, Serpent, based on Python. [3] Serpent is portable and is written entirely in C++. Serpent includes a real-time garbage collector that never suspends computation more than one out of every two milliseconds (these parameters are adjustable to even smaller values). When used with Aura, an instance of the Serpent run-time system runs in each Aura zone, and Aura is used for all scheduling and communication among these instances. Python and other scripting languages could not be used because they do not allow multiple instances of the virtual machine to run concurrently in a shared address space. This is necessary to allow preemption, which is necessary in turn to achieve low-latency.

Although Serpent is a recent addition to Aura, it seems to work very well and has greatly simplified the task of specifying audio computations. Now, a functional description of "instruments" or general networks of unit generators can be given in Serpent. Serpent also allows functions to be invoked by sending a list of parameters as a message, something that is awkward with C++'s stronger typing and more restricted function call semantics.

Very high-level and/or application-specific languages are already found in Music V [11], Nyquist [4], SuperColider [12], and various SmallTalk [14, 17] and Lisp-based [18] music systems, so it should be no surprise that Serpent offers advantages over C++. A challenge is dealing with a dual-language system. C++ is still used for low-level DSP, communications and scheduling. C++ is more prominent in Aura than in the other systems mentioned above, most of which also use C and C++ in their implementation.

The design could probably be simplified by making all Aura objects a subclass of Serpent objects (or vice versa) and thereby hiding C++, but this would mean all computation would have to involve the Serpent interpreter, which is about 100 times slower than C++ (this is typical for interpreted languages). On the other hand, leaving a thin interface between Aura and Serpent exposes the programmer to many C++ details that should be hidden. I plan to make the Serpent layer in Aura a more complete programming system so that at least beginning users can view the system completely in terms of Serpent. Ideally, C++ should only be used when performance is an issue.

9. AURA AND OSC

Although references to related work are offered throughout this discussion of Aura, it seems appropriate to comment on OSC [7], especially since OSC is mentioned elsewhere in these proceedings and OSC is used in many similar applications.

OSC has a much more limited purpose than Aura. OSC provides an open-ended mechanism for communicating with and controlling a synthesizer. OSC can also represent the changing values of multiple sensors conveniently. Thus, it makes sense to compare OSC to the Aura message system, especially with regard to naming.

OSC and Aura both use attributes and values to represent control changes, commands, and changes in sensor values. OSC attributes are strings, whereas Aura attributes are globally unique 32-bit values. The use of relatively small, fixed-sized attributes is at least a slight advantage for the low-level implementation, but it complicates some things at higher levels. Aura uses preprocessors to manage the translation from strings to 32-bit values at compile time, and, while the preprocessor sometimes detects typos and other errors, failure to declare attributes properly is a common error. Designers of both OSC and Aura have discussed switching to the opposite approach, so it seems there is no clearly winning strategy here.

Another important difference is that OSC uses path names to access attributes, whereas Aura messages must be sent directly to the receiving object. We envisioned a programming model in Aura where a controlling process would typically keep references to objects and subobjects, and therefore send messages directly to the receiver, as illustrated in Figure 2.

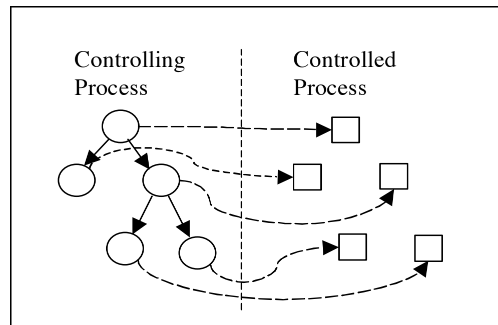

Figure 2. Controlling process in Aura maintains references to a network of objects and sends messages directly to receivers.

If necessary, a path could be followed within the controlling process to find the reference to the controlled object. This seems especially appropriate if the "controlled process" is not a hierarchically organized system or if there are many unrelated lines of communication. In contrast, OSC messages are directed to the root of a hierarchical space of controlled objects and routed by decoding symbolic paths, as shown in Figure 3.

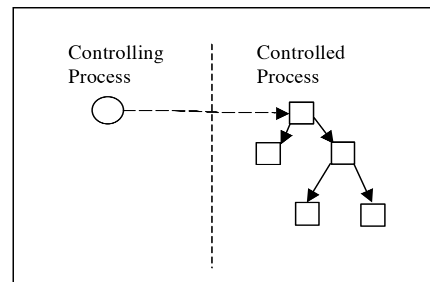

Figure 3. Controlling process in OSC sends messages to controlled process where paths are decoded to locate receivers.

Naming and structure are very important for programmers. Aura programs have contained many structures where the OSC model would be constraining. On the other hand, there are many examples where a hierarchical name space for attributes would be convenient to access the corresponding hierarchical structure of a sound or animation synthesis algorithm. Messages with OSC-like path names would be an interesting addition to Aura.

Finally, both Aura and OSC started out without type specifications for attributes. Experience has shown that typing is critical, and both systems now use types. Since Aura deals with attribute/value pairs, it is important that all values of a given

attribute have the same type. Thus, each Aura attribute has an associated type, and types are globally enforced. Types are indicated by a suffix in the attribute identifier, e.g. `lengthi` is an integer length, and `hzd` is a double representing frequency in Hz.

10. SUMMARY AND CONCLUSIONS

In summary, Aura provides the foundations for distributed, real-time, highly concurrent applications. Its multiple zone architecture facilitates a mixture of tasks with varying latency requirements. My goal in this paper is to present the many issues that arise in these applications and how they are successfully (or unsuccessfully) handled in Aura. Experience with Aura gives us some insights into what works and what does not, and where further experimentation or design changes might prove beneficial.

Some of the useful lessons include:

1. Objects can be extended to use “real” asynchronous messages, and this provides an effective way to organize distributed real-time programs.
2. Object communication based on attributes and values is a viable alternative to method invocation, especially for music and animation where there are many control parameters.
3. Visual continuity in the representation of program flow is important for readability.
4. Systems should avoid shared memory and associated synchronization problems.
5. Use threads so that low-latency computations preempt long-running computations. Do not separate tasks into separate threads when they could share a single thread.
6. Network communication can be the result of basic message passing, eliminating special network code and API’s from the concern of the application programmer.
7. Debugging support is important, especially the ability to monitor real-time activity. (Otherwise, use conventional debuggers.)
8. Large portions of applications can (and should) often use a high-level scripting language like Serpent.
9. Message data should be strongly typed.

I hope that this paper will serve as a checklist for others who might be designing related platforms, architectures, or applications. I also hope that the remaining problems and weaknesses of Aura might inspire solutions in future designs, either in new versions of Aura or in other systems.

Aura source code is available from the author, and collaboration is welcome, although Aura should be viewed as research in progress rather than a finished product. Aura has run under Windows, Irix, and Linux, and Aura should be relatively easy to port to Mac OS X. Current development is under Linux.

11. REFERENCES

[1] Anderson, D.P. and Kuivila, R. A System for Computer Music Performance. ACM Transactions on Computer Systems, 8 (1). 56-82.

[2] Brandt, E. and Dannenberg, R.B., Time in Distributed Real-Time Systems. in Proceedings of the 1999 ICMC, (1999), International Computer Music Conference, 523-526.

[3] Dannenberg, R. A Language for Interactive Audio Applications, (submitted for publication), 2002.

[4] Dannenberg, R.B. Machine Tongues XIX: Nyquist, a Language for Composition and Sound Synthesis. Computer Music Journal, 21 (3). 50-60.

[5] Dannenberg, R.B. Software Design for Interactive Multimedia Performance. Interface - Journal of New Music Research, 22 (3). 213-228.

[6] Freed, A. and Chaudhary, A., Music Programming wtih the new Features of Standard C++. in ICMC, (1998), International Computer Music Association.

[7] Freed, A. and Wright, M., Open SoundControl: A New Protocol for Communicating with Sound Synthesizers. in Proceedings of the 1997 ICMC, (1997), International Computer Music Association.

[8] Hoare, C.A.R. Monitors: An Operating System Structuring Concept. Communications of the ACM, 17 (10). 549-557.

[9] Ingalls, D.H., The Smalltalk-76 Programming System Design and Implementation. in POPL, (1978), ACM, 9-16.

[10] LAD. Low Latency, http://www.linuxdj.com/audio/lad/resourceslatency.php3, 2002.

[11] Mathews, M. The Technology of Computer Music. MIT Press, 1969.

[12] McCartney, J., SuperCollider: A New Real Time Synthesis Language. in Proceedings of the 1996 International Computer Music Conference, (1996), International Computer Music Association.

[13] Myers, B., Giuse, D., Dannenberg, R., Zanden, B.V., Kosbie, D., Pervin, E., Mickish, A. and Marchal, P. Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. IEEE Computer, 23 (11). 71-85.

[14] Pope, S.T., Siren: Software for Music Composition and Performance in Squeak. in Proceedings of the 1997 International Computer Music Conference, (1997), International Computer Music Association.

[15] Puckette, M. Combining Event and Signal Processing in the MAX Graphical Programming Environment. Computer Music Journal, 15 (3). 68-77.

[16] Puckette, M. and Zicarelli, D. MAX Development Package. Opcode Systems, Inc., 1991.

[17] Scaletti, C. and Johnson, R.E., An Interactive Graphic Environment for Object-oriented Music Composition and Sound Synthesis. in Proceedings of the Conference on Object-Oriented Programming Languages and Systems, (1988), ACM.

[18] Schottstaedt, W. Machine Tongues XVII. CLM: Music V Meets Common Lisp. Computer Music Journal, 18 (2). 30-38.

[19] Sha, L., Rajkumar, R. and Lehoczky, J.P. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. IEEE Transactions on Computers, 39 (9). 1175-1184.

[20] Zicarelli, D. M and Jam Factory. Computer Music Journal, 11 (4). 13-23.