A VIRTUAL PATCHBAY FOR ROBUST DISTRIBUTED INTERACTIVE MUSIC SYSTEMS [1]

Roger B. Dannenberg
School of Computer Science,
Carnegie Mellon University
dannenberg@cs.cmu.edu

ABSTRACT

Multiple computers and/or processors offer interactive music systems more processing power, more inputs and outputs, and more tolerance to failure. Systems based on multiple computers require careful design paying particular attention to communication and configuration. The *virtual patchbay* is a new structure in Aura that simplifies the configuration and interconnection of objects in a distributed computer music application. The virtual patchbay also optimizes configurations to reduce the number of duplicate messages sent over the network and helps the system tolerate crashes and rebooting while other components continue to function.

1. INTRODUCTION

As computer music systems become more complex, interprocess communication offers one approach to managing complexity. For example, the OSC protocol is used to create client-server systems for music processing [10], MAX [7] and Pd [8] run their graphical user interface and audio processing in separate threads, and CSL is intended for a network of processors communicating via CORBA [6]. Communication facilitates the construction of distributed systems, which have the advantages of more power, more input and output devices, and fault-tolerance.

This paper explores various ways to structure a computer music application for distributed- or multi-processing systems. Designs are considered in terms of ease-of-use, efficiency, and robustness in the face of program failures (crashes). This work led to the design and implementation of a "virtual patchbay" mechanism for Aura. This virtual patchbay is described in detail.

1.1. Process-to-process Communication

Perhaps the simplest way for processes to communicate is the standard client-server structure using sockets. In this organization, the client requests a two-way connection to a server via sockets, which represent buffered streams of bytes. The client can write messages or data which are sent asynchronously to the server. The server can read bytes as they arrive and optionally send replies or new requests through another socket.

In simple systems, where there are one or more clients and a single server, and clients are sending commands, this organization works well. However, as requirements change and control becomes richer, the programmer/composer must extend the protocol understood by both the server and the clients, leading to problems of version control, documentation updates, and program maintenance.

1.2. Communication with OSC

OSC was designed to address this problem. In OSC, the server, its operations, and its parameters are represented by a hierarchical name space. To offer a new function in the server, the server registers the new function with OSC. Clients can then construct and send messages to call the new function, but old client code can (ordinarily) continue to work without change. Functions are named with strings, and strings are mapped to functions at run-time to minimize version control problems and to allow even run time extensions to be created.

In spite of its popularity, OSC has some limitations. OSC assumes a global address space, so in general, all messages are sent to the same server port and must include a complete URL-style destination address. Of course, the server name space can be divided into any number of sub-spaces, but then the client must prepend the subspace name to the destination. For example, addresses could be /subsys1/a, /subsys2/b, and /subsys3/c, but then a client designed to interact with subsystem 1 must prepend "/subsys1" to names such as "/a". OSC could potentially use a port per object or use multiple ports to reach multiple servers, but there is no built-in management for these connections.

OSC names tend to be hard-wired. To connect midi input to a synthesizer, a standard practice is to write mapper programs to accept, say, midi data (/midi-in/control-change), map the data to a destination, and send a new message (/tone5/spectral-centroid). This could be regarded as a feature [10], but writing mapping programs may be a cumbersome approach for some.

1.3. Communication with CORBA

CORBA [9] is a general communications architecture that addresses many of these issues. In particular, CORBA has mechanisms for binding names or descriptions of services to actual servers. CORBA is especially useful in distributed systems with multiple servers that may fail. After a failure, the clients can (potentially) locate an alternative server and continue working.

Unfortunately, CORBA has a large specification, and while there are real-time CORBA implementations, "real-time" generally means "on-line transaction processing" where delays of 100ms or more are considered to

[1] Published as: Roger B. Dannenberg. "A Virtual Patchbay for Robust Distributed Interactive Music Systems," in Proceedings of the 2005 International Computer Music Conference, San Francisco: International Computer Music Association, (2005), pp. 571-574.

be "real time." In practice, there are fast implementations of CORBA suitable for real-time music processing [6], and real-time extensions to CORBA are an active area of research and development.[1] However, CORBA is fairly complex, and it is difficult to adopt only selected parts. For example, one might want to have tight control over scheduling and memory allocation, but a CORBA implementation might create a thread, wait on a socket, and allocate memory for sending and receiving messages.

1.4. Communication with Aura

In contrast to CORBA's wide scope, generality, and diversity of implementations, Aura [1] supports only a narrow model of system organization but achieves high performance with modest code size. Aura is carefully written to support very efficient communication and real-time control. For example, Aura avoids locks altogether to eliminate the possibility of priority inversion, a known weakness in many operating systems, and Aura uses its own real-time memory allocator rather than the default allocators provided in the C and C++ runtime systems.

Aura is first and foremost a general object system. Music objects and functions are built on top of this model, so it is important to explain the basics of the Aura system. Aura objects are written in C++ or Serpent, a real-time scripting language. All Aura objects inherit some behavior from the *Aobject* class, including globally-unique 64-bit identifiers and the ability to send asynchronous messages to invoke a method (called "remote method invocation" or RMI). To call the *set_hz* method of some object, one can write:

send_set_hz_to(*osc*, 440.0)

This automatically-generated macro builds a message and sends it to *osc*, which must be a 64-bit object ID.

Aura objects are partitioned according to a two-level hierarchy. At the top level are "machines" which represent processes. At the next level are "zones" which represent threads within processes. An object exists in the memory pool of, and its methods are executed by, one and only one thread (or Aura zone).[2]

Getting back to messages, Aura messages are delivered using three possible mechanisms (see Figure 1). When a message is destined for an object in the same zone, the message is delivered synchronously using the C++ call stack. If the message is sent to another zone in the same process, the message is copied to a circular buffer for the target zone. That zone's thread then reads the message and performs a local send to call the method. If the message is bound for another process, a network proxy object sends the message over a socket to the destination process, which then forwards the message locally. [3]

While it is normal for one object to address a message to a specific target object, Aura objects also have output ports and input ports. By connecting an output port to another object, messages can be directed from senders to receivers. In Figure 2, Obj1 is connected to Obj2, Obj3, and Obj4. If Obj1 performs

send_set_hz(440.0)

it will send out copies of a *set_hz* message to Obj2, Obj3, and Obj4. If a receiver does not recognize a message, it is ignored. This style of connecting objects is intended to mimic MIDI connections, streams of text messages, control streams from graphical objects such as sliders, and other forms of control. The interesting thing here is that connections can be created and destroyed externally to the connected objects. In designing Aura, we envisioned that a user might interconnect objects at run time the way one patches audio and midi components to build and troubleshoot complex systems. [4]

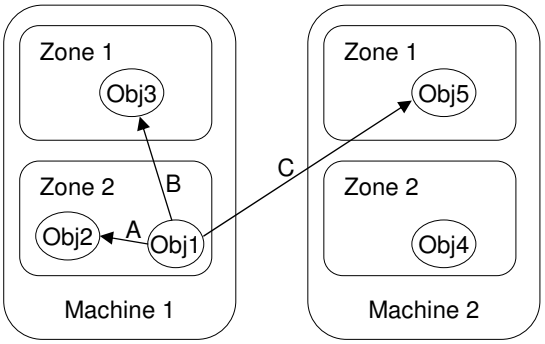

Figure 1. An object (Obj1) can send messages (A) locally, (B) to another zone in the same process, and (C) across the network to a different process.

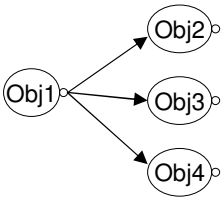

Figure 2. Obj1 has an output port that can be connected to any number of other objects. Messages can be "broadcast," that is, a copy of the message is delivered to each connected receiver. Connections can be created and destroyed at runtime by any object.

An advantage of this approach is that any two objects can be connected, even if they are on different machines. E.g. a sensor that runs on machine 1 can be connected to a local message logging object, a graphical display object on machine 2, and a synthesis object on machine 3. In contrast to OSC or some other client/server scheme, the location of Aura objects is transparent. Local sends use the same syntax as remote sends, so the network topology is isolated from and independent of the program logic.

1.5. Problems to be Solved

Transparency, however, comes at a price. Because objects communicate directly with one another, objects must hold the unique identifiers of other objects. If a process fails (crashes or terminates), objects in other

[1] C.f. http://www.cs.wustl.edu/~schmidt/TAO.html.

[2] Aura's model of computation and message passing is quite similar to the "Single-Threaded Apartment" model of DCOM (http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndcom/html/msdn_dcomarch.asp).

processes may be left holding invalid object identifiers. If the failed process is restarted, newly created objects may not have the same unique identifiers, so communication paths will not be restored automatically.

In contrast, a scheme like MIDI or OSC allows a synthesizer or controller to be rebooted because the MIDI namespace (channels, key numbers, controller numbers) or OSC namespace (a hierarchical space of identifiers) is fixed and exists externally to the communicating objects.

Another, related problem with the Aura scheme is that objects must exist before they can be connected. In practice, this means that configuring a distributed Aura system can require several phases to make sure that processes are created first, then objects, and then connections between objects. To some extent, these problems exist in all distributed systems, but work in Aura can at least be simplified, as we shall see.

2. THE VIRTUAL PATCHBAY

It is these problems that the *virtual patchbay* is designed to solve. The idea is simple: a *patchbay* object maintains a set of logical *patchpoints.* Any number of objects may be designated as *sources* for a patchpoint, and any number of objects may be designated as *sinks* for a patchpoint. Whenever a patchpoint source object sends a message, the message is delivered to all of the patchpoint sinks. This can be viewed as a simple instance of the publish/subscribe communication paradigm [5]. The interesting aspect of this work is how the virtual patchbay can be implemented to support (1) machine failures and restarts, (2) efficient local communication, and (3) efficient non-local communication.

2.1. The Virtual Patchbay Interface

The virtual patchbay interface consists of only a few methods whose semantics should be obvious:

create_patchpoint(*string*)
add_source(*string, objectID*)
remove_source(*string, objectID*)
add_sink(*string, objectID*)
remove_sink(*string, objectID*)

For example, to create a connection from Obj1 to Obj2 through a patchpoint named "env_sensor", one could write:

create_patchpoint("env_sensor")
add_source("env_sensor", *Obj1*)
add_sink("env_sensor", *Obj2*)

Now, messages sent from Obj1 will arrive at Obj2.

2.2. Basic Implementation

Aura systems must include a designated "master" process that gives unique names to "slave" processes, establishes a time reference, performs distributed clock synchronization, and optionally establishes a global sample clock. [3] The virtual patchbay runs on the master as an ordinary Aura object. It maintains a simple database that associates patchpoints with sinks and sources. When a new sink is added to a patchpoint, the patchbay makes a connection from each of the patchpoint's sources to the new sink. When a new source is added to the patchpoint, the patchbay makes a connection between the new source and each of the patchpoint's sinks. Connections are deleted when a sink or source is removed from the patchpoint.

Notice that *messages are not delivered to or through the patchbay.* The patchbay merely makes and breaks direct connections so that, for example, local connections require only local processing with no indirection or forwarding.

2.3. Robustness to Crashes and Rebooting

To deal with crashes and rebooting, the patchbay must: (1) detect when a slave has crashed, (2) delete connections to and from "dead" objects, (3) reestablish connections when slaves are rebooted. Notice that the master is a single point of failure; the entire distributed application must be restarted if the master fails.

To detect that a slave has terminated (for any reason), the patchbay relies upon a network proxy object. This proxy maintains bidirectional TCP/IP sockets to each slave. These sockets are used to forward Aura messages between processes. When a process terminates, the network proxy object gets a notification from the operating system that the corresponding socket has closed. The network proxy object informs the virtual patchbay object of any change in status via an Aura message.

When the patchbay object gets a notice that a slave has terminated, it scans its database of connections for objects that were allocated in the slave (and are now presumed dead). Each such object is removed, prompting the appropriate messages to delete connections that are no longer valid. (There are various race conditions that may cause messages to be sent to non-existent objects. Aura simply drops these messages so they do not cause further problems.)

When a slave is rebooted, it contacts the master and reestablishes network connections with the master and other slaves. If the slave makes *add_source*() and *add_sink*() calls to connect new objects to patchpoints, then connections between objects will be reestablished.

For example, suppose a slave captures data from a MIDI-based controller and sends the data to a software synthesizer running on another machine. The data is sent via the logical patchpoint named "controller1." The synthesizer object is the sink for "controller1." When the slave attached to the MIDI controller is (re)booted, it establishes the MIDI input object as a source for "controller1." The virtual patchbay then makes a connection from the MIDI input object (the source) to the software synthesizer object (the sink). Notice that the MIDI input object and the synthesizer object do not need to know about one another since their connection is managed by the virtual patchbay. Neither source nor sink depends upon the other, so no special sequencing is required when the system is initialized.

The patchbay is most useful for streams of data (e.g.. audio, MIDI, sensor data, and text messages) where the loss of a message is not a disaster. Ideally, each message

is independent of previous messages and carries enough information to bring the receiver up-to-date. Messages that contain the complete state of the sender are ideal in this respect, while messages that contain incremental updates create a problem if the sender is restarted. If the loss of a message can cause an unrecoverable error, then crash recovery is much more complicated than simply restarting a process and restoring a connection. This is a tricky problem even in non-real-time systems, and Aura does not try to solve it automatically. However, well-known techniques are available to the application builder, including "active sense" or "heartbeat" messages to monitor process behavior, resetting objects to a known state, and query messages to retrieve lost state information.

2.4. Message-Passing Optimization

One shortcoming of the Aura message-passing architecture is that when messages are broadcast from a sender to multiple receivers, messages are copied and duplicates may be sent over the network to the same machine (but to different final destinations). If the sender is connected to a dozen objects on a remote machine, then a dozen copies of the message are sent over the network. It would be much more efficient to send one message and make copies at the receiving end, but this would greatly complicate the message passing protocol, which is optimized for fast, local sending.

The virtual patchbay offers a solution. Since the virtual patchbay has a global view of the connections, it can detect when a source is sending to multiple sinks in a remote zone. When this happens, the virtual patchbay creates a "message forwarder" object that simply forwards any incoming message to its output port. The virtual patchbay uses the forwarder object to make local copies of a message after it reaches the remote zone. Figure 3 illustrates the configuration without (left) and with (right) the use of forwarder objects.

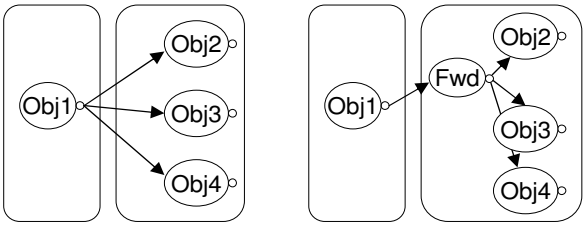

Figure 3. Ordinarily, Obj1 must send three copies of each message to reach remote objects Obj2, Obj3, and Obj4, as shown at left. The virtual patchbay creates a forwarding object, Fwd, so that copies are only made locally in the receiving zone, which is more efficient, as shown at right.

2.5. Performance

Because the virtual patchbay merely establishes and deletes connections, it introduces no overhead to ordinary message passing and actually reduces overhead when connections can be optimized with a forwarding object. The additional overhead to create a connection is just one message per patchpoint, one message per source or sink, and two messages per forwarding object. Assuming that connections are long-lived, the overhead is negligible. The patchbay is implemented in Serpent, the Aura scripting language. [2]

3. CONCLUSIONS

Distributed systems will become increasingly common in interactive music. At present, few distributed music processing systems have been constructed, and most use very simple client/server architectures, often connecting just a pair of machines. Aura offers an interesting architecture based on communicating objects that extends quite readily to a multiple-processor or multiple-computer configuration. Communication in Aura offers the advantage of location transparency. The virtual patchbay gives Aura a simple means for coni

figuration and crash recovery as well as the ability to optimize message distribution.

4. REFERENCES

[1] Dannenberg, R.B. "Aura II: Making Real-Time Systems Safe for Music", *Proceedings of the 2004 Conference on New Interfaces for Musical Expression (NIME04)*, Hamamatsu, Japan, 2004, 132-137.

[2] Dannenberg, R.B. "Combining Visual and Textual Representations for Flexible Interactive Audio Signal Processing", *Proceedings of ICMC 2004: The 30th Annual International Computer Music Conference*, Coral Gables, Florida, 2004, 240-247.

[3] Dannenberg, R.B. and Lageweg, P.v.d. "A System Supporting Flexible Distributed Real-Time Music Processing", *Proceedings of the 2001 International Computer Music Conference*, Havana, 2001, 267-270.

[4] Dannenberg, R.B. and Rubine, D. "Toward Modular, Portable, Real-Time Software", *Proceedings of the 1995 International Computer Music Conference*, Banff, Canada, 1995, 65-72.

[5] Eugster, P.T., Felber, P.A., Guerraoui, R. and Kermarrec, A.-M. "The Many Faces of Publish/Subscribe", *ACM Computing Surveys*, *35*, 2 (June), 2003, 114-131.

[6] Pope, S. and Engberg, A. "Distributed Control and Computation in the HPDM and DSCP Projects", *Proceedings of the Symposium on Sensing and Input for Media-Centric Systems*, Santa Barbara, 2002, 38-43.

[7] Puckette, M. "Max at Seventeen", *Computer Music Journal*, *26*, 4, 2002, 31-43.

[8] Puckette, M. "Pure Data", *1997 International Computer Music Conference*, Thessaloniki, Greece, 1997, 224-227.

[9] Vinoski, S. "CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments", *IEEE Communications Magazine*, *35*, 2, 1997.

[10] Wright, M., Freed, A., Lee, A., Madden, T. and Momeni, A. "Managing Complexity with Explicit Mapping of Gestures to Sound Control with OSC", *Proceedings of the 2001 International Computer Music Conference*, Havana, 2001, 314-317.