Implementation and Performance Issues with OpenSound Control

Matthew Wright (matt@cnmat.berkeley.edu)
Center for New Music and Audio Technologies, 1750 Arch Street, Berkeley, CA 94709, USA

OpenSound Control (OSC) is a new protocol for high-level, expressive control of sound synthesis and other multimedia applications. It includes time-tagged messages, guaranteed atomicity of messages with the same time tag, and regular expression address patterns that can match multiple messages in a receiving application. We hope OSC becomes a standard, and towards this end have made available at no cost the OpenSound Control Kit, a C or C++ library that adds OSC addressability to an application. This Kit is designed to be added to a sound processing application without degrading reactive real-time performance. This paper accompanies the Kit and provides a high-level overview of the Kit's interfaces to the rest of an application and how performance issues are addressed.

1. Introduction To OpenSound Control

OpenSound Control is described in detail in last year's ICMC paper [Wright 97] and on the OSC home page (http://www.cnmat.berkeley.edu/OpenSoundControl). OSC is a transport-independent protocol optimized for modern networking technology. It relies on some underlying transport to deliver *packets*, blocks of arbitrary binary data with a size count, among OSC-aware applications.

Unlike MIDI [MIDI 95], ZIPI's MPDL [McMillen 94] and other musical control languages, OSC does not enforce any model of “channels,” “notes,” “orchestras,” “velocity,” etc. OSC's model is musically neutral and much more general: an OSC application consists of a dynamically changing set of objects arranged hierarchically, and each of these objects has a set of messages that can be sent to it to control its behavior. Thus, the goal of making an application OSC-addressable is not to come up with features that match predefined OSC messages, but to provide a set of OSC messages that match the features of the application and organize them into a meaningful hierarchy.

An OSC message consists of an *address pattern*, a slash-separated ASCII string similar to a URL but with a powerful pattern-matching syntax similar to regular expressions [Friedl 97], and *arguments*, an arbitrary amount of binary data that will be interpreted by the receiving application as some combination of 4-byte floats, 4-byte two's complement integers, and ASCII strings. These native machine representations facilitate real-time performance by eliminating the need to reformat received data. An OSC server expands an address pattern into a list of all the addresses in the current hierarchy that match the pattern, similar to the way UNIX shell “globbing” interprets special characters in filenames. Each address in the list then receives a message with the given arguments. For example, the pattern “/filterbanks/4 /*/set-freq” might match the addresses “/filterbanks/4 /lowpass/set-freq,” “/filterbanks/4 /notch1/set-freq,” and “/filterbanks/4 /notch2/set-freq.”

OSC messages can be grouped into *bundles* that are sent in the same packet. A bundle has a sub-nanosecond resolution time tag that tells the receiving device when the messages are to take effect.

Some OSC messages are *queries*, which allow clients to request information from the server. Queries allow a client to determine what objects and messages exist in the address space, request documentation, determine the expected argument types for a given message, etc.

2. Introduction to the OpenSound Control Kit

The OSC Kit (http://www.cnmat.berkeley.edu/OpenSoundControl/Kit) implements as many of the features of an OSC-addressable application as possible. (We have also made available a library that constructs OSC via a procedural interface that hides the OSC byte format.) The following issues are handled internally and need not concern users of the Kit: byte format of OSC, time tags and scheduling of messages in the future, atomicity of messages with the same time tag, pattern matching, memory management of OSC objects, and automatic answering of certain queries.

The Kit is available as both C and C++ libraries. The APIs use an object-oriented style with opaque objects; the C version represents objects as pointers to structs. All communication between the Kit and the rest of the application is via arguments and return values; the interface does not use global variables.

3. Interface to OSC-addressable features

OSC is a tool that is useful only insofar as there are interesting features that can be controlled by it. Therefore, the Kit's API for making features controllable via OSC is designed to be as straightforward and convenient as possible.

This paper will use the term “user” to refer to the OSC client, presumably a musician, who controls some feature via OSC, and “implementor” to refer to the programmer who implements that feature and makes it OSC-addressable.

The hierarchical OSC address space is modeled as an object-oriented tree of *containers*, each of which contains subcontainers (for tree structure) and *methods*, which can implement the actions that OSC messages call for, e.g., updating the value of a parameter. A container's methods and subcontainers share a single namespace. The root of this tree, corresponding to the address “/,” is returned by the procedure that initializes the address space. All operations on this tree, including adding or removing a container or method, are O(1) and can be performed dynamically with no risk of compromising reactive real-time performance.

Each method contains a *callback procedure*, written by the implementor, which the Kit will invoke at the time that the OSC message is to take effect. The arguments to a callback procedure are:

- `void *context` - Supplied by the implementor when the method was added to the address space
- `int arglen` - Number of bytes of argument data that was sent by the user
- `void *const args` - The argument data itself
- `OSCTimeTag when` – The time tag used to schedule this message
- `NetworkReturnAddressPtr returnAddr` - An opaque object that can be used to send an OSC message back to the user.

For maximum efficiency, the Kit never copies argument data in memory. The `args` pointer points to data in the buffer where the packet was received. It is the implementor's job to interpret this data according to the argument types expected by this method. Most methods take a single number or a list of numbers of the same type (e.g., 3 floats) as arguments, so their callback procedures simply cast `args` to the appropriate pointer type and treat it as an array. The Kit provides helper procedures for dealing with OSC-style (4-byte aligned) ASCII strings, including one that turns `args` and `arglen` into an array of `char *` pointers. Callback procedures typically copy their argument data into wherever they keep the state of the application, or compute new values based on the arguments to the callback. They should not store any pointer derived from the `args` pointer, because eventually the buffer holding the packet will be reused.

When a single address pattern expands into multiple method addresses in the hierarchy, each of their callback procedures is called with the same `args` and `arglen` arguments. The `context` pointer allows for an object-oriented style that can handle polyphony and other “multiple instances of the same functionality with different state” features by registering the same callback procedure with multiple `context`s.

The OSC Kit automatically detects the standard OSC queries and generates a response. The raw data needed as the answer to queries like “what are the argument types for this message” or “give me the documentation for this feature” are provided by the implementor when registering containers and methods.

4. Interfaces for Adding OSC to a Real-Time System

There are many ways that the OSC Kit needs to tie into the rest of a reactive realtime system. It must interact with the network services to send and receive packets. It interacts with the scheduler and control flow of the overall real-time system to ensure that the Kit has enough time to process incoming messages but that it defers processing of messages scheduled to occur in the future until the system has some otherwise idle time. Finally, the Kit needs memory to store the address space hierarchy and to receive and process incoming messages; this memory must be available dynamically in real-time, but the Kit does not assume that the real-time system will necessarily have a real-time memory allocator.

All of these interfaces were designed with the following goals:

- Minimal assumptions about what the rest of the system will look like
- Maximum real-time performance
- Simplicity and ease of use
- Maximum flexibility to change underlying implementations in the future (e.g., for performance optimization) without breaking code written to the API

4.1 Interface to low-level networking code

Network services are usually provided by the operating system via an API. These APIs generally expect to be given a buffer of memory in which to place incoming data from the network. So the OSC Kit manages a pool of PacketBuffer objects that consist mainly of a large buffer for this purpose. (They also store an implementationdependent network return address.) The Kit never copies OSC data in memory; OSC data is parsed in the PacketBuffer, and eventually a number of callback procedures are invoked with `args` values that point into the PacketBuffer. Because an OSC packet may contain time-tagged messages to take effect in the future, these PacketBuffers cannot be reused until the last message in the packet takes effect.

4.2 Control Flow/Scheduling Model

The Kit does not include its own scheduler; we assume that the design and implementation of the main part of the sound processing application will determine the overall flow of control. The OSC Kit does its work when the main part of the application calls procedures in the Kit's scheduling API. Low-latency digital sound processing requires a scheduling model in which code to compute output samples is run frequently, leaving many short periods of time for the processor to do other things like process OSC input. Therefore, the Kit's scheduling API is based on the assumption that its procedures will be called often and that they should do a small amount of work and return quickly.

Here is a gross approximation of the inner loop of an OSC-addressable sound synthesizer:

```
while (1) {
    OSCTimeTag now = GetCurrentTime();
    while (WeHaveEnoughTimeToInvokeMessages()) {
        if (!OSCInvokeMessagesThatAreReady(now)) break;
    }
    SynthesizeSomeSound();
    while (NetworkPacketWaiting()) {
        OSCPacketBuffer p = OSCAllocPacketBuffer();
        if (!p) {
            Bummer();
        } else {
            NetworkReceivePacket(p);
            OSCAcceptPacket(p);
        }
    }
    while (TimeLeftBeforeWeHaveDoSomething()) {
        if (!OSCBeProductiveWhileWaiting()) break;
    }
}
```

`OSCAcceptPacket` “accepts” a newly received packet. When a time tag indicates a message that needs to happen immediately (or should already have happened), it has no choice but to process it immediately: parse, pattern-match the address pattern against the addresses in the current hierarchy, and invoke the necessary callback procedures. When a bundle's time tag indicates a time still in the future, there is time between receipt of the bundle and when it must be executed. In this case, `OSCAcceptPacket` defers all work, inserting the bundle into the priority queue of messages to be invoked in the future. This queue uses a standard heap data structure [Bentley 86] with O(log(n)) insertion and deletion, but it is easy for the implementor to create a custom queue, e.g., to take advantage of knowing that the main scheduler always operates on fixed-size time blocks [Dannenberg 89].

`OSCBeProductiveWhileWaiting` performs a small amount of “background” processing on messages in the queue. The idea is to invoke this procedure when the processor would otherwise be idle. (In a thread-based system, a low-priority thread would continuously call this procedure.) This procedure's return value indicates whether there is more work to do, so it can be called as many times as possible during these idle times.

`OSCInvokeMessagesThatAreReady` takes the current time as an argument, and invokes the callbacks whose time has arrived. (This is the Kit's only interface to the part of the system that knows what time it is.) This should be called frequently to ensure low control latency. Atomicity is guaranteed by this procedure, because it invokes all of the callbacks with a particular time tag before returning. (In a thread-based system, the thread invoking the callbacks must prevent the audio-generating thread from running until `OSCInvokeMessagesThatAreReady` returns.)

We have integrated the OSC Kit into a real-time system running under IRIX and based upon the `select()` system call. An example application using this arrangement is available for download. We have also designed a thread-based architecture using the Kit, with separate threads for accepting incoming packets, background processing , producing the list of callback procedures that are ready to be invoked , and invoking callback procedures. Pseudocode for the thread design is also available for download.

4.3 Memory Model

The OSC system needs dynamic memory allocation for the address space, for PacketBuffers, for the priority queue, and for lists of methods that match address patterns. To avoid the problems associated with real-time memory management, the OSC Kit preallocates pools of fixed size memory chunks for each of these objects. A very fast custom memory manager provides O(1) allocation and freeing via these pools. Users of the Kit pass arguments to the initialization procedures that say how many of each kind of object to allocate.

The Kit does not expose implementation details like the contents of these data structures, so in order for users of the Kit to decide how many objects to preallocate we provide a procedure that takes the same arguments as the initialization procedure and returns the number of bytes of memory that would be allocated.

Initialization procedures also take two function pointers as arguments: a pointer to the “initialization-time memory allocator” and a pointer to the “real-time memory allocator.” Both of these procedures, like `malloc()`, take a number of bytes as argument and return a pointer to that much free memory. The Kit invokes the initialization-time allocator only from the initialization procedure. It invokes the real-time allocator if any of the preallocated object pools runs out. If the real-time allocator fails, the Kit has to drop a message, refuse to add to the address space, or otherwise fail in a graceful manner. If a system does not have real-time memory allocation, the real-time allocator can simply be a procedure that always returns 0. Because the memory allocator used by the kit is an argument rather than an internal procedure, it is possible to tune the memory system without recompiling the Kit.

Although much more sophisticated designs would be possible, this system performs well and allows for the important case of static overall memory limits. The ability to convert free memory for one kind of object into needed memory for another kind of object could avoid certain “not enough memory” situations, but managing pools of different-sized objects would drastically increase the complexity of the memory allocator.

5. Conclusions

OSC is general enough to control a variety of musical and other applications, and has features such as time tags, address patterns, and queries that provide high level control. The OSC Kit efficiently adds OSC addressability to an application and automatically handles most of the features of OSC. The kit makes it easy to define an OSC address space for an application and provides a very efficient mechanism to get incoming OSC message data to the appropriate part of an application. The main issues that need to be understood to integrate the Kit into a real-time application are memory management and the sharing of CPU time between the Kit and the rest of the application.

The Kit has been used at CNMAT with the CAST synthesizer [Freed, 95] and with a host of other systems based on the HTM kernel [Freed, 94] including vector panning for sound spatialization, resonances, multi-tap delay lines, and singing voice simulations. Each of these systems has its own unique OSC address space.

We expect to refine the interface and improve the implementations according to experience as the protocol is used on other platforms. We invite anyone building a sound processing application to make it OSC-addressable with the Kit and to give us feedback.

6. References

[Bentley 86] Bentley, J., 1986. *Programming Pearls*, Reading, Massachusetts: Addison-Wesley, pp. 125-137.

[Dannenberg 89] Dannenberg, R, 1989. “Real-Time Scheduling and Computer Accompaniment,” in M. Mathews and J. Pierce, Editors, *Current Directions in Computer Music Research*, Cambridge, Massachusetts: MIT Press, pp. 225-261.

[Freed 94] Freed, A., 1994. “Codevelopment of User Interface, Control and Digital Signal Processing with the HTM Environment,” Proc. International Conference on Signal Processing Applications & Technology, Dallas.

[Freed 95] Freed, A., 1995. "Bring Your Own Control to Additive Synthesis," Proceedings of the ICMC, Banff, Canada, pp. 303-306. See also `http://www.CNMAT.Berkeley.EDU/CAST/Server/CAST-OSC-implementation.html` which documents the OSC address space used in CAST.

[Friedl 97] Friedl, J., 1997. *Mastering Regular Expressions : Powerful Techniques for Perl and Other Tools*, Sebastopol, CA: O'Reilly & Associates

[McMillen 94] McMillen, K, D. Wessel, and M. Wright, 1994. "The ZIPI Music Parameter Description Language," *Computer Music Journal*, Volume 18, Number 4, pp. 52-73.

[MIDI 95] MIDI Manufacturers Association, 1995. *The Complete MIDI 1.0 Detailed Specification*, Los Angeles.

[Wright 97] Wright, M., and A. Freed, “OpenSound Control: A New Protocol for Communicating with Sound Synthesizers,” pp. 101-104, Proceedings of the ICMC, Thessaloniki, Hellas, 1997.