Published as: Roger B. Dannenberg and Zhang Chi. "O2: Rethinking Open Sound Control," in *Proceedings of the 42nd International Computer Music Conference,* Utrecht: HKU University of the Arts Utrecht, 2016, pp. 493-496.

O2: Rethinking Open Sound Control

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

Zhang Chi
Carnegie Mellon and Tianjin University
zcdirk@gmail.com

ABSTRACT

O2 is a new communication protocol and implementation for music systems that aims to replace Open Sound Control (OSC). Many computer musicians routinely deal with problems of interconnection in local area networks, unreliable message delivery, and clock synchronization. O2 solves these problems, offering named services, automatic network address discovery, clock synchronization, and a reliable message delivery option, as well as interoperability with existing OSC libraries and applications. Aside from these new features, O2 owes much of its design to OSC and is mostly compatible with and similar to OSC. O2 addresses the problems of inter-process communication with a minimum of complexity.

1. INTRODUCTION

Music software and other artistic applications of computers are often organized as a collection of communicating processes. Simple protocols such as MIDI [7] and Open Sound Control (OSC) [1] have been very effective for this, allowing users to piece together systems in a modular fashion. Shared communication protocols allow implementers to use a variety of languages, apply off-the-shelf applications and devices, and interface with low-cost sensors and actuators. We introduce a new protocol, O2, in order to provide some important new features.

A common problem with existing protocols is initializing connections. For example, typical OSC servers do not have fixed IP addresses and cannot be found via DNS servers as is common with Web servers. Instead, OSC users usually enter IP addresses and port numbers manually. The numbers cannot be "compiled in" to code because IP addresses are dynamically assigned and could change between development, testing, and performance. O2 allows programmers to create and address services with fixed human-readable names.

Another desirable features is timed message deliveries. One powerful method of reducing timing jitter in networks is to pre-compute commands and send them in advance for precise delivery according to timestamps. O2 facilitates this forward synchronous approach [6] with timestamps and clocks.

Finally, music applications often have two conflicting requirements for message delivery. Sampled sensor data should be sent with minimum latency. Lost data is of little consequence since a new sensor reading will soon follow. This calls for a best-effort delivery mechanism such as UDP. On the other hand, some messages are critical, e.g. "stop now." These critical messages are best sent with a reliable delivery mechanism such as TCP.

Our goal has been to create a simple, extensible communication mechanism for modern computer music (and other) systems. O2 is inspired by OSC, but there are some important differences. While OSC does not specify details of the transport mechanism, O2 uses TCP and UDP over IP (which in turn can use Ethernet, WiFi, and other data link layers). By assuming a common IP transport layer, it is straightforward to add discovery, a reliable message option, and accurate timing.

In the following section, we describe O2, focusing on novel features. Section 3 presents related work. Then, in Sections 4 and 5, we describe the design and implementation, and in Section 6, we describe how O2 interoperates with other technologies. Section 7 describes our current implementation status, and a summary and conclusions are presented in Section 8.

2. O2 FEATURES AND API

The main organization of O2 is illustrated in Figure 1. Communication takes place between "services" which are addressed by name using an extension of OSC addressing in which the first node is considered a service name. For example, `"/synth/filter/cutoff"` might address a node in the `"synth"` service. To create a service, one writes

o2_initialize("`application`"); *// one-time startup*
o2_add_service("`service`"); *// per-service startup*
o2_add_method("`address`", "`types`", *handler, data*);

where "`application`" is an application name, used so that multiple O2 applications can co-exist on one network, and *o2_add_method* is called to install a handler for each node, where each "`address`" includes the service name as the first node.

Services are automatically detected and connected by O2. This solves the problem of manually entering IP addresses and port numbers. In addition, O2 runs a clock synchronization service to establish a shared clock across the distributed application. The master clock is provided to O2 by calling:

o2_set_clock(*clock_callback_fn*, *info_ptr*);

where *clock_callback_fn* is a function pointer that provides a time reference, and *info_ptr* is a parameter to pass to the function. The master clock can be the local system time of some host, an audio sample count converted to

seconds (for synchronizing to audio), SMPTE time code, GPS, or any other time reference.

Messages can be sent either with lowest latency or reliably using two "flavors" of send function:

o2_send ("`address`", *time,* "`types`", val_1, val_2, …);
o2_send_cmd ("`address`", *time,* "`types`", val_1, val_2, …);

where "`types`" (in the C implementation) specifies the types of parameters, e.g. `"if"` means val_1 is an integer and val_2 is a float. The first form uses UDP, which is most common for OSC, and the second form sends a "command" using TCP, ensuring that the message will be delivered. Notice that every send command specifies a delivery *time*.

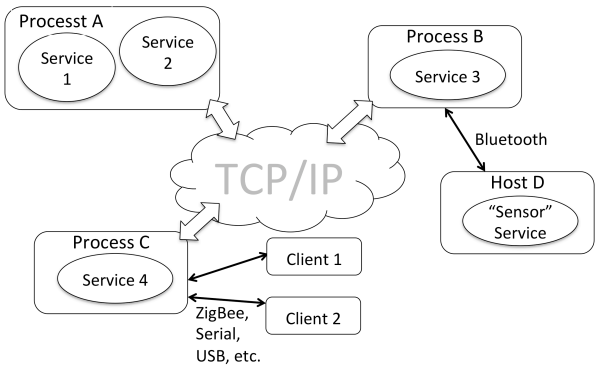

Figure 1. A distributed O2 application showing processes connected by TCP/IP (wireless and/or wired) over a local area network, running multiple services, with additional single-hop links over Bluetooth, ZigBee, etc. to both services and simple clients that do not receive messages. Services on Process A may run within a single process or in separate processes, and all processes may act as clients, sending messages to any service.

3. RELATED WORK

Open Sound Control (OSC) has been extremely successful as a communication protocol for a variety of music and media applications. The protocol is simple, extensible, and supported by many systems and implementations. The basic design supports a hierarchical address space of variables that can be set to typed values using messages. The messages can convey multiple values, and thus OSC may be viewed as a remote function or method invocation protocol. One very appealing quality of OSC, as compared to distributed object systems (such as CORBA [2]), is that OSC is very simple. In particular, the OSC address space is text-based and similar to a URL. It has been argued that OSC would be more efficient if it used fixed-length binary addresses, but OSC addresses are usually human-readable and do not require any preprocessing or run-time lookup that would be required by more efficient message formats. The success of OSC suggests that users are happy with the speed and generally are not interested in greater efficiency at the cost of more complexity.

Clock synchronization techniques are widely known. Madgwick *et al.* [5] describe one for OSC that uses broadcast from a master and assumes bounds on clock drift rates. Brandt and Dannenberg describe a round-trip method with proportional-integral controller [6]. OSC itself supports timestamps, but only in message bundles, and there is no built-in clock synchronization.

Discovery in O2 automatically shares IP addresses and port numbers to establish connections between processes. The liboscqs[1] and OSCgroups[2] library and osctools[3] project support discovery through zeroconf [3] and other systems. Also, Eales and Foss explored discovery protocols in connection with OSC for audio control [4], however their emphasis is on querying the structure of an OSC address space rather than discovery of servers on the network.

Software developers have also discussed and implemented OSC over TCP for reliable delivery. Systems such as liblo[4] offer either UDP or TCP, but not both unless multiple servers are set up, one for each protocol.

4. DESIGN DETAILS

In designing O2, we considered that networking, embedded computers, laptops, and mobile devices have all advanced considerably since the origins of OSC. In particular, embedded computers running Linux or otherwise supporting TCP/IP are now small and inexpensive, and the Internet of Things (IOT) will spur further development of low-cost, low-power, networked sensors and controllers. While OSC deliberately avoided dependency on a particular transport technology to enable low-cost, lightweight communication, O2 assumes that TCP/IP is available to (most) hosts. O2 uses that assumption to offer new features. We also use floating point for simple clock synchronization calculations because floating point hardware has become commonplace even on low-cost microcontrollers, or at least microcontrollers are fast enough to emulate floating point as needed.

4.1 Addresses in O2

In OSC, most applications require users to manually set up connections by entering IP and port numbers. In contrast, O2 provides "services." An O2 *service* is just a unique name used to route messages within a distributed application. O2 addresses begin with the service name, making services the top-level node of a global address space. Thus, while OSC might direct a message to `"/filter/cutoff"` at IP 128.2.1.39, port 3, a complete O2 address would be written simply as `"/synth/filter/cutoff"`, where `"synth"` is the service name.

4.2 UDP vs. TCP for Message Delivery

The two main protocols for delivering data over IP are TCP and UDP. TCP is "reliable" in that messages are retransmitted until they are successfully received, and subsequent messages are queued to insure in-order delivery. UDP messages are often more appropriate for real-time sensor data because new data can be delivered out of

[1] http://liboscqs.sourceforge.net
[2] http://www.rossbencina.com/code/oscgroups
[3] https://sourceforge.net/projects/osctools
[4] http://liblo.sourceforge.net/

order rather than waiting for delivery or even retransmission of older data. O2 supports both protocols.

4.3 Time Stamps and Synchronization

O2 protocols include clock synchronization and time-stamped messages. Unlike OSC, *every* message is time-stamped, but one can always send 0.0 to mean "as soon as possible." Synchronization is initiated by clients, which communicate independently with the master.

5. IMPLEMENTATION

The O2 implementation is small and leverages existing functionality in TCP/IP. In this section, we describe the implementation of the important new features of O2.

5.1 Service Discovery

To send a message, an O2 client must map the service name from the address (or address pattern) to an IP address and port number. We considered existing discovery protocols such as ZeroConf (also known as Rendezvous and Avahi), but decided a simpler protocol based on UDP broadcast messages would be smaller, more portable to small systems, and give more flexibility if new requirements arise.

The O2 discovery protocol uses 5 fixed "discovery" port numbers. We use 5 because we cannot guarantee any one port is unallocated and multiple O2 applications (up to 5) might run on a single host, each requiring a port. When O2 is initialized, O2 allocates a server port and broadcasts the server port, host IP address, local service names and an application name to the 5 discovery ports. Any process running an instance of O2 with the same application name will receive one of these broadcasts, establish TCP and IP sockets connected to the remote process, and store the service name and sockets in a table. Multiple independent applications can share the same local area network without interference if they have different application names. O2 retransmits discovery information periodically since there is no guarantee that all processes receive the first transmissions.

To direct a message to a service, the client simply looks in the lookup table for the appropriate socket and sends the message using TCP or UDP. O2 allows multiple services within a single process without confusion because every message contains its destination service name.

5.2 Timestamps and Clock Synchronization

O2 uses its own protocol to implement clock synchronization. O2 looks for a service named `"_cs"` and when available, sends messages to `"/_cs/ping"` with a reply-to address and sequence number. The service sends the current time and sequence number to the reply-to address. The client then estimates the server's time as the reported time plus half the round-trip time. All times are IEEE standard double-precision floats in units of seconds since the start of the clock sync service. O2 does not require or provide absolute date and time values.

5.3 Replies and Queries

Normally, O2 messages do not send replies, and we do not propose any built-in query system at this time, mainly because queries never caught on in OSC implementations. Unlike classic remote procedure call systems implementing synchronous calls with return values, real-time music systems are generally designed around asynchronous messages to avoid blocking to wait for a reply.

Rather than build in an elaborate query/reply mechanism, we advocate a very simple application-level approach where the "query" sends a *reply-to* address string. The handler for a query sends the reply as an ordinary message to a node under the *reply-to* address. For example, if the *reply-to* address in a `"/synth/cpuload/get"` message is `"/control/synthload"`, then the handler for `"/synth/cpuload/get"` sends the time back to (by convention) `"/control/synthload/get-reply"`. Optionally, an error response could be sent to `"/control/synthload/get-error"`, and other reply addresses or protocols can be easily constructed at the application level.

5.4 Address Pattern Matching and Message Delivery

To facilitate the implementation of O2, we (mostly) adhere to OSC message format. Notice that an O2 server can scan an address string for the "/" after the service name to obtain an OSC-style address pattern. This substring, type information, and data can be passed to many existing OSC implementations for further processing, eliminating the need to implement an all-new message parser. Similarly, existing OSC marshaling code (which converts data to/from messages) can be used to construct messages for O2.

OSC has been criticized for the need to perform potentially expensive parsing and pattern matching to deliver messages. O2 adds a small extension for efficiency: The client can use the form `"!synth/filter/cutoff"`, where the initial "!" means the address has no "wildcards." If the "!" is present, the receiver can treat the entire remainder of the address, "synth/filter/cutoff" as a key and do a hash-table lookup of the handler in a single step. This is merely an option, as a node-by-node pattern match of `"/synth/filter/cutoff"` should return the same handler function.

6. INTEROPERATION

OSC is widely used by existing software. OSC-based software can be integrated with O2 with minimal effort, providing a migration path from OSC to O2. O2 also offers the possibility of connecting over protocols such as Bluetooth[5], MIDI [7], or ZigBee[6].

6.1 Receiving from OSC

To receive incoming OSC messages, call
o2_create_osc_port(`"service"`, *port_num*);
which tells O2 to begin receiving OSC messages on *port_num*, directing them to `service`, which is normally

[5] http://www.bluetooth.org
[6] http://www.zigbee.org

local, but could also be remote. Since O2 uses OSC-compatible types and parameter representations, this adds very little overhead to the implementation. If bundles are present, the OSC NTP-style timestamps must be converted to O2 timestamps before messages are handed off.

6.2 Sending to OSC

To forward messages to an OSC server, call

o2_delegate_to_osc("`service`", *ip*, *port_num*);

that tells O2 to create a virtual service (name given by the `service` parameter), which converts incoming O2 messages to OSC messages and forwards them to the given *ip* address and *port_num*. Now, any O2 client on the network can discover and send messages to the OSC server.

6.3 Other Transports

Handling OSC messages from other communication technologies poses two interesting problems: What to do about discovery, and what exactly is the protocol? The O2 API can also be supported directly on clients and servers connected by non-IP technologies. As an example, let us assume we want to use O2 on a Bluetooth device (we will call it Process D, see Figure 1) that offers the "Sensor" service. We require a direct Bluetooth connection to Process B running O2. Process B will claim to offer the "Sensor" service and transmit that through the discovery protocol to all other O2 processes connected via TCP/IP. Any message to "Sensor" will be delivered via IP to Process B, which will then forward the message to Host D via Bluetooth. Similarly, programs running on Host D can send O2 messages to Process B via Bluetooth where the messages will either be delivered locally or be forwarded via TCP/IP to their final service destination. It is even possible for the destination to include a final forwarding step though another Bluetooth connection to another computer, for example there could be services running on computers attached to Process C in Figure 1.

Non-IP networks are supported by optional libraries, essentially giving O2 a "plug-in" architecture to ensure both a small core and flexibility to create extensions.

In addition to addressing services, O2 sometimes needs to address the O2 subsystem itself, e.g. clock synchronization runs even in processes with no services. Services starting with digits e.g. `"128.2.60.110:8000"`, are interpreted as an IP:Port pair. To reach an attached non-IP host, a suffix may be attached, e.g. Host D in Figure 1 might be addressed by `"128.2.60.110:8000:bt1"`.

7. CURRENT STATUS

A prototype of O2 in the C programming language is running the discovery algorithm and sending messages. Performance measurements show that CPU time is dominated by UDP packet send and receive time, even when messages are sent to another process on the same host (no network link is involved). We were unable to measure any impact of discovery or service lookup in a test where two processes send a message back and forth as fast as possible. In this test, total message delivery (real or "wall") time is about 13μs, or 77,000 messages per second, on a 2.4 GHz Intel Core i7 processor, which is faster than OSC using liblo due to some minor differences in the way messages are accepted from the network.

We believe O2 is a good candidate for OSC-like applications in the future. A number of extensions are possible, and future work includes extensions to allow discovery beyond local area networks, audio and video streaming, and dealing with network address translation (NAT).

O2 is available: https://github.com/rbdannenberg/o2.

8. SUMMARY AND CONCLUSIONS

O2 is a new protocol for real-time interactive music systems. It can be seen as an extension of Open Sound Control, keeping the proven features and adding solutions to some common problems encountered in OSC systems. In particular, O2 allows applications to address services by name, eliminating the need to manually enter IP addresses and port numbers to form connected components. In addition, O2 offers a standard clock synchronization and time-stamping system that is suitable for local area networks. O2 offers two classes of messages so that "commands" can be delivered reliably and sensor data can be delivered with minimal latency. We have implemented a prototype of O2 that is similar in size, complexity and speed to an Open Sound Control implementation. Although O2 assumes that processes are connected using TCP/IP, we have also described how O2 can be extended over a single hop to computers via Bluetooth, ZigBee or other communication links.

Acknowledgments

Thanks to Adrian Freed for comments on a draft of this paper.

9. REFERENCES

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

[2] M. Henning, "The rise and fall of CORBA," *ACM Queue,* vol. 4, no. 5, 2006, pp. 29-34.

[3] E. Guttman, "Autoconfiguration for IP Networking: Enabling Local Communication," *IEEE Internet Computing,* vol. 5, no. 3, 2001, pp. 81–86

[4] A. Eales and R. Foss, "Service discovery using Open Sound Control," *AES 133rd Convention,* San Francisco, 2012.

[5] S. Madgwick, T. Mitchell, C. Barreto, and A. Freed, "Simple synchronisation for open sound control. *41st International Computer Music Conference 2015*, Denton, Texas, 2015, pp . 218-225.

[6] E. Brandt and R. Dannenberg, "Time in Distributed Real-Time Systems," Proceedings of the International Computer Music Conference , 1999.

[7] J. Rothstein, *MIDI: A Comprehensive Introduction, 2nd ed.*, A-R Editions, 1995.