Hindawi
Wireless Communications and Mobile Computing
Volume 2019, Article ID 8424381, 12 pages
https://doi.org/10.1155/2019/8424381

Research Article

O2: A Network Protocol for Music Systems

Roger B. Dannenberg

School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213, USA

Correspondence should be addressed to Roger B. Dannenberg; rbd@cs.cmu.edu

Received 2 January 2019; Revised 18 March 2019; Accepted 4 April 2019; Published 6 May 2019

Guest Editor: Federico Fontana

O2 is a communication protocol for music systems that extends and interoperates with the popular Open Sound Control (OSC) protocol. Many computer musicians routinely deal with problems of interconnection, 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, making it easy to migrate existing OSC applications to O2 or for developers familiar with OSC to begin using O2. O2 addresses the problems of interprocess communication within distributed music applications.

1. Introduction

Music software and other artistic applications of computers are often organized as a collection of communicating processes. Simple protocols such as MIDI [1] and Open Sound Control (OSC) [2] 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. In addition, mobile applications intrinsically run on multiple distributed host computers and require a communication protocol for any kind of coordination.

Figure 1 illustrates several common organizations for networked music systems. Figure 1(a) shows the "input/mapper/output" structure, where sensors and input devices stream values to a control system that maps sensor values to control parameters, which are passed on to a music synthesizer or audio signal processor [3]. Examples of this approach include SensorChimes [4] and play-along mappings of Fiebrink *et al.* [5]. The libmapper system is a communication protocol designed to support this approach [6].

Figure 1(b) illustrates a "conductor/ensemble" structure commonly used in laptop orchestra and mobile device music systems. Multiple performers can join and leave the ensemble by connecting to a central conductor or coordinator that directs the performers. Examples are described by Essl [7], Trueman *et al.* [8], and Dannenberg *et al.* [9]. This same configuration is used in wide-area networked music performances on a global scale such as quintet.net [10] and the Global Network Orchestra [11].

Figure 1(c) illustrates a peer-to-peer organization that is used in networked performances characterized by autonomy and emergent behavior as opposed to the top-down control seen in the conductor/ensemble model. Gresham-Lancaster offers an interesting early history and discussion of this approach [12]. More examples of all three structures are described by Wright [13].

In all of these patterns, we see networking has advanced from point-to-point communication to more flexible and comprehensive communication substrates often used as much for software modularity and resilience as for communication. We introduce the protocol, O2, that provides for communication and coordination among music processes and offers some important new features over previous protocols such as OSC.

A common problem in any distributed system is how to initialize 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

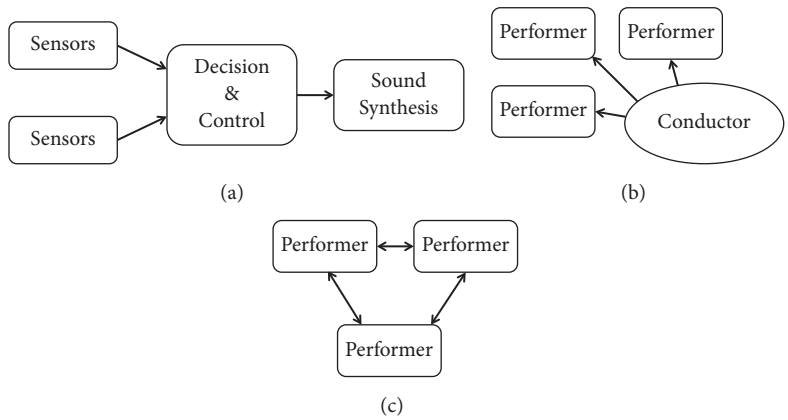

Figure 1: Common structures for networked music systems: (a) the "input/mapper/output" structure; (b) the "conductor/ensemble" structure; (c) peer-to-peer structure.

eliminates most network configuration problems, allowing programmers to create and address services with fixed, human-readable names.

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, and retransmitting stale sensor data serves little purpose. This calls for a best-effort delivery mechanism such as UDP. On the other hand, some messages are critical and one-time only, e.g., "stop now." These critical messages are best sent with a reliable delivery mechanism such as TCP.

Another desirable feature is timed message delivery, especially for music where timing is critical. One powerful method of reducing timing jitter in networks is to precompute commands and send them in advance for precise delivery according to timestamps. O2 facilitates this forward-synchronous approach [14] with timestamps and clocks.

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 relatively straightforward to add discovery, a reliable message option, and accurate timing.

In the following section, we describe important and novel features of O2. This is followed by a section on related work. Then, we describe the design and implementation, focusing on novel aspects of O2. A communication protocol is mainly useful as "glue" between different systems. In the section "Interoperation," we describe how O2 interoperates with Open Sound Control, Web applications, and various languages. Finally, we present some performance measurements, a summary, and conclusions.

2. O2 Features and API

The main organization of O2 is illustrated in Figure 2. We will first introduce some O2 terminology. An O2 *host* is a computer with an IP address. An O2 *process* is a running program. There may be multiple processes running on a single host. An O2 *ensemble* is a named collection of *processes* that communicate and share services. Every O2 process belongs to one and only one ensemble. This allows multiple independent performers or groups to use O2 over the same network without interference. An O2 *address* is a URL-like string designating a function or method. For example, `/synth/filter/cutoff` might address a function to set the filter cutoff frequency in the `synth` service.

2.1. Creating a Service. The top-level (first) node in an *address* names an O2 *service*. A service is an abstraction for a set of functions or a resource such as a synthesizer, a display, a sensor, or a controller. A service is accessible via O2 *messages*, which consist of an *address* and a set of typed parameters. Services can be created dynamically by any process, services are "owned" by a process and automatically discovered by all other O2 processes, and all messages addressed to a service are delivered by invoking a registered callback function within the process.

To create a service, one writes

```
o2_initialize(ensemble); // one-time O2 startup
o2_service_new(service); // create a new service
```

where *ensemble* is a unique ensemble name.

Typically, each full address in the hierarchical name space represents a function. To associate a function with an O2 address, call

```
o2_method_new(address, types, handler,
info, coerce, parse);
```

where *address* is the full address, e.g. `/synth/filter/cutoff`, *types* gives the expected parameter types (for example, `"si"` specifies that a string and a 32-bit integer parameter are expected), *handler* is the address of the callback function to which the parameters are passed, and *info* is an additional parameter to pass to this handler function.

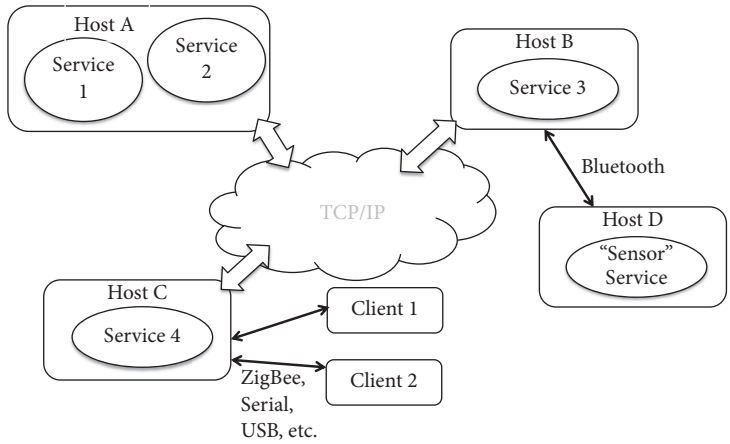

Figure 2: A distributed O2 ensemble showing processes connected by TCP/IP (wireless and/or wired) over a local area network, running multiple services, with additional single-hop links over Bluetooth, ZigBee, etc. Services on Host A may run within a single process or in separate processes, and all processes may act as clients, sending messages to any service.

The *coerce* and *parse* parameters give additional control over message handling.

2.2. Discovery. Services are automatically detected and connected by O2. This solves the problem of manually entering IP addresses and port numbers. The discovery process is detailed in Section 5.

Messages in O2 are delivered only when the address exists, so if a service has not been created, a network connection is lost, or a process terminates, the message will be dropped without raising any exceptions. This is often a great simplification for applications. For example, a sensor process can send sensor data to a consumer service whether or not the service exists. It is not necessary to carefully start the server process before starting the client (sensor process). When the service is active, it gets the data; when it is not, the service lookup will fail locally, and no message will be sent. In practice, this can cause problems when a client needs to configure a service or request important information from it. In these cases, it is common for clients to call

`o2_status(`*service*`)`

until the *service* is created and discovered, at which point communication can begin. It might be noted that similar problems arise even with simple client-server communication with TCP/IP: the client's `connect` call will fail (after at least a round-trip across the network) if the server does not exist.

2.3. Sending Messages. Messages can be sent either with lowest latency or reliably using two different functions:

`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. The first form (`o2_send`) uses UDP, which is most common for OSC, and the second form (`o2_send_cmd`) sends a "command" using TCP, ensuring that the message will be delivered.

2.4. Timed Message Delivery. O2 runs a clock synchronization service to establish a shared clock across the distributed ensemble. The master clock is provided to O2 by calling

`o2_set_clock(`*clock_callback_fn*, *info*`);`

where *clock_callback_fn* is a function pointer that provides a time reference and *info* 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 into seconds (for synchronizing to audio), SMPTE time code, GPS, or any other time reference. Notice that every send operation (`o2_send, o2_send_cmd`) specifies a delivery time. A send operation always transmits a message immediately to the receiver without regard for the timestamp. If the message arrives early, it is held in a priority queue until the delivery time. Thus, if messages can be sent early (e.g., by increasing overall latency), message delivery times can be very precise (reducing timing jitter) [14]. This is often important for accurate timing in music.

3. Related Work

Table 1 summarizes some properties and features of various systems for networked music applications. Many simple systems implement application-specific protocols using TCP or UDP to carry text or binary data. Open Sound Control (OSC) offers a simple but very successful standard protocol for a variety of music and media applications [2]. The protocol is 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

Table 1: Summary of properties of O2 and some alternative communication systems for real-time music networks. "Location Transparency" means that processes or services can be addressed by name or function rather than by direct IP addresses and/or port numbers. "Data Streams" refers to libmapper's unique ability connect producers of numerical data streams to consumers, including mapping and filtering options to adapt sensor outputs to controller system inputs.

	Location Transparency	Typed, Named Parameters	Discovery	Mixed Reliable & Best Effort	Timed Delivery	Data Streams	Comments
TCP or UDP							
OSC		√					Timed delivery is possible; rarely implemented.
O2	√	√	√	√	√		
libmapper	√	√	√			√	
Landini	√	√	√	√	√		Messages sent indirectly through local servers; N^2 ping traffic limits ensemble size.

to distributed object systems (such as CORBA [15]), is its simplicity. In particular, the OSC address space is text-based and similar to a URL, which means that programmers can write human-readable addresses directly without the need for interface description languages or preprocessors to translate strings to binary codes. It has been argued that OSC would be more efficient if it used fixed-length binary addresses, but the success of OSC suggests that users are not interested in greater efficiency at the cost of more complexity.

Discovery in O2 automatically shares IP addresses and port numbers to establish connections between processes. The liboscqs (http://liboscqs.sourceforge.net) and OSCgroups (http://www.rossbencina.com/code/oscgroups) library and osctools (https://sourceforge.net/projects/osctools) project support discovery through Zeroconf [16] and other systems. Malloch [6] describes the use of multicast for discovery, but this requires an agreed-upon and reserved multicast address. Essl [7] advocates the use of Bonjour (Apple's implementation of Zeroconf), and included Bonjour-based discovery into networked mobile-phone-based music software. Bonjour has been slow to become a standard cross-platform service, but it offers a good solution to discovery. Eales and Foss explored discovery protocols in connection with OSC for audio control [17]; however their emphasis is on querying the structure of an OSC address space rather than discovery of servers on the network.

LANdini [18] addresses many of the problems that O2 is designed to solve. To solve the problem of discovery, LANdini runs a server on each host, and servers discover other servers using UDP broadcast messages. A sending process delivers a message to the local LANdini server, which forwards the message to the destination host's LANdini server, and from there the message is forwarded again to the receiving process. This means that three messages are sent for each application-level message delivery. Since LANdini is built using OSC, which in turn uses UDP, LANdini implements its own retransmission scheme for reliable message sending. An implementation with n hosts sends $3n(n\text{-}1)$ messages per second, limiting the practical ensemble size. LANdini also performs clock synchronization among servers, but there is no additional synchronization between servers and the ultimate destination processes.

The libmapper system [6] is particularly aimed at "input/mapper/output" systems (Figure 1(b)) and directly supports connections with specified mappings and filters, which is beyond the scope of O2. However, libmapper seems to be less suited to more general communication including over wide area network systems.

Software developers have also discussed and implemented OSC over TCP for reliable delivery. Systems such as liblo (http://liblo.sourceforge.net/) offer either UDP or TCP, but not both unless multiple servers are set up, one for each protocol.

Clock synchronization techniques are widely known. Cristian [19] described a simple method that is the basis for clock synchronization in O2. Madgwick *et al.* [20] describe a method 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 [14]. OSC itself supports timestamps, but only in message bundles, and there is no built-in clock synchronization.

4. Design Considerations and Details

In designing O2, we considered that computing technology is not as limited today as it was when OSC was designed. 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, enabling 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 to simplify 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 versus 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 immediately rather than waiting for delivery or even retransmission of older data. O2 supports both protocols.

To illustrate the need for both delivery protocols, we wrote simple O2 programs to send 20,000 short messages at 20 messages per second, alternating use of TCP and UDP between two personal computers sharing a local WiFi network. Five of 10,000 UDP messages (0.05%) were lost by the network, and the maximum delay between receive times of two consecutive UDP messages was 343 ms. Of course, all TCP messages were delivered, and the maximum delay between messages was also 343 ms. However, the delay between TCP messages was greater than 110 ms 303 times (3%) but only 89 times (0.9%) with UDP. Thus, TCP retransmissions generate a significant number of delays that might be avoided using UDP when packet loss is not critical. These numbers are highly dependent upon network behavior, but it is clear that TCP and UDP are both useful.

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 a designated master clock process.

4.4. Taps and Debugging Support. Debugging a distributed application is difficult in part because there is no single point of control. When a component fails to behave as expected, it is helpful to know what messages, if any, are being received there. O2 has a message monitoring facility:

```
o2_tap(tappee, tapper);
```

installs a "tap," where messages delivered to service *tappee* (a string) are copied, the message address is modified by replacing *tappee* with *tapper*, and the new message is delivered (to service *tapper*). This facility supports the construction of a remote message monitoring program. A simple monitor has been implemented and is described below in the "Interoperation" section.

5. Implementation

The O2 implementation is small and leverages existing functionality in TCP/IP. The OS X library, for example, is about 130KB, compared to a popular OSC library, liblo, which is 100KB. 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 Bonjour, 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. In particular, ZeroConf must be installed and configured as a server on some operating systems, whereas discovery in O2 is integrated with the O2 library.

Figure 3 illustrates the discovery sequence. When an O2 process is initialized, it allocates a server port and broadcasts its server port, host IP address, and an ensemble name. Any process running an instance of O2 with the same ensemble name will receive one of these broadcasts, establish a TCP connection to the remote process, and exchange service names. Multiple independent ensembles can share the same local area network without interference if they have different ensemble names. O2 retransmits discovery information every *discovery period* since there is no guarantee that all processes receive the first transmissions. The discovery period is short enough to avoid long discovery latency and long enough to avoid too much network traffic. (See "Scaling Issues" below.)

To direct a message to a service, the sender extracts the service name from the full address and uses a hash table to find an entry for the service name. The entry contains the corresponding TCP socket or address for a UDP message. The message is then sent to the process. The receiver uses another hash table to find a handler for the message and invokes the handler.

5.2. Discovery Ports. As described above, each O2 *process* performs discovery directly without relying on a separate server process. Unfortunately, this requires the use of multiple ports (one per O2 process). In an early implementation, it was thought that each process would allocate 1 of n predefined discovery port numbers. Then, discovery messages would be broadcast to all n port numbers. Since message traffic grows linearly with n, there is pressure to keep n small, but n is the upper bound on the number of processes that can run on a single host, so it seems that n should be at least 10, increasing discovery message traffic by a factor of $n \geq 10$.

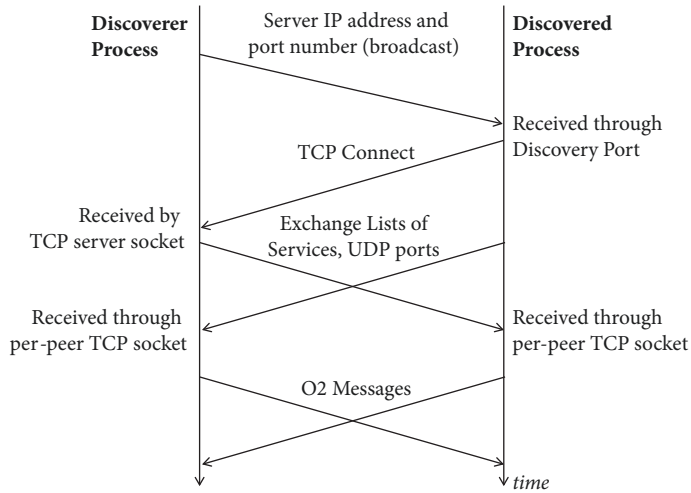

Figure 3: Discovery protocol. After receiving an O2 process's IP address and port number, a peer establishes a TCP connection and the peers exchange service names and UDP ports (for messages sent via UDP), after which processes can exchange O2 messages. There is a chance in a fully symmetrical protocol that each process can connect to the other simultaneously. To avoid this, the TCP Connect is only issued from the process with the lower IP:port combination, breaking the symmetry.

Perhaps surprisingly, we can make n arbitrarily large without necessarily increasing discovery message traffic, using the following method. Discovery port numbers are ordered, and processes allocate the first available port from an ordered list: $port_1$, $port_2$, ... $port_n$. Now, consider two processes A and B, which have allocated $port_{PA}$ and $port_{PB}$. Process A will only broadcast to ports 1...*PA*, and process B will only broadcast to ports 1...*PB.* One of the following must be true: either $PA < PB$, $PA > PB$, or $PA = PB$. If $PA < PB$, then B will broadcast to $port_{PA}$. If $PA > PB$, then A will broadcast to $port_{PB}$. If $PA = PB$, then A will broadcast to $port_{PB}$ and B will broadcast to $port_{PA}$. In all cases, a discovery message will be sent from one process to the other, and both will be connected. If A and B are on separate hosts, each will typically open $port_1$ and only broadcast to $port_1$. In the case of $m \leq n$ processes on the same host, they will typically open $port_1$ through $port_m$ requiring $m(m+1)/2$ broadcast messages every period, but we do not expect m to be large.

The default O2 configuration sets $n = 16$, allowing up to 16 O2 processes on one host. Ports are not reserved through the Internet Assigned Numbers Authority (IANA), but since O2 discovery will work as long as *any* port from the set of 16 is available, reserved ports are not critical. In summary, the advantage of our port allocation and discovery scheme is that rather than broadcasting from m processes to every possible discovery port, costing 16 m messages, a distributed ensemble with 1 process on each of m hosts will broadcast only m messages per discovery period. Furthermore, this peer-to-peer system does not require a separate discovery server process.

Unfortunately, not all networks allow broadcast messages, and broadcasts are (usually) only available within a local area network. As an alternative to discovery, any O2 process can find all services by calling

```
o2_hub(ip_address, port);
```

which provides the IP address and port for *one* other O2 process. O2 then contacts the other process and receives a list of the current services as well as any future updates. A typical use case is a wireless network where broadcast is disabled and IP addresses are assigned dynamically, making it impossible to use fixed IP addresses in the application software. Instead, a designated "hub" O2 process posts its IP address and port to a simple web server. Other O2 processes request the hub information from the server and then call `o2_hub` to form a fully connected ensemble. Posting an address to a server in this way is performing a function similar to DNS. An alternative to creating this specialized web service is to configure hosts to use a dynamic DNS service and have at least one O2 process register its address there.

5.3. Timestamps and Clock Synchronization. O2 implements 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 service. O2 does not require or provide absolute date and time values.

This basic synchronization protocol suffers when messages are delayed, so O2 retains information from the last 5 pings and uses the one with the lowest round trip time. Another problem, especially in music systems, is that when clocks are adjusted, carefully timed music sequences can literally skip a beat. In O2, when the local clock is not synchronized, it is sped up or slowed down by 10% until it matches the estimated master clock time. While 10% may seem large, it is not a perceptually large change in terms of musical tempo where the just-noticeable difference is 6-8% [21], and the speed-up or slow-down period typically lasts

for only tens of milliseconds. In the case of very large clock adjustments exceeding 1 second, it is considered musically not useful to remain so far out of synchronization, so the local clock is set to the correct time immediately.

5.4. Scaling Issues. Discovery, clock synchronization, and reliable transmission all add network traffic to an O2 ensemble. O2 is designed to support up to 100 processes in one ensemble. We assume that the ensemble itself does not exceed the network capacity, so our only design constraint is that the *overhead* of using O2 does not overly tax the network or processing time as the number of processes scales up to 100.

One source of sustained network traffic is the clock synchronization protocol, in which each client periodically sends a round-trip UDP packet to the master clock process. We estimate clock time based on the fastest round-trip time to the master in 5 tries. With a polling period of 10 s, the clock will be updated within 50 s. Assuming clock rate differences of 40 PPM (based on standard ±20 PPM oscillators (https://www.ctscorp.com/wp-content/uploads/CTS-Corporation-Clock-Oscillator-Timing-Frequency-Electronic-Component-Manufacturer.pdf)), the worst-case drift in 50 s is 2 ms, which is low in perceptual terms [22]. With a send and reply message every 10 s, 99 clients will generate only 19.8 messages per second. This low polling rate has the disadvantage that processes would take about 40 s to establish synchronization. Instead, the protocol sends the first 15 messages more rapidly, achieving initial synchronization typically within 0.5 s without increasing the long-term network traffic.

Discovery messages are sent periodically and are necessary since a new process could join the ensemble at any time. There is a trade-off between the mean time to join and the density of network traffic. O2 uses a period of 4 seconds, resulting in 25 messages per second in a 100-process ensemble. As with clock synchronization, O2 uses a "fast start" with a 0.133 s initial period between messages. In the worst case, a discovery message is broadcast to all 16 discovery ports in the first 4 seconds, but typically, in a distributed system, O2 processes will use the first discovery port and discovery will take place almost immediately. If the `o2_hub` function is used by a process, discovery messages are not needed and none are sent by the process.

A final source of network overhead is the TCP connections that are maintained between each pair of processes. With 100 processes, there will be 4950 connections, but of course these will be distributed across processes, with each process making 99 connections. These connections are created as part of the discovery process, so when a process joins an ensemble of 99 other processes, there will be a burst of network traffic (about 297 packets) to establish 99 TCP connections. Aside from this somewhat bursty setup behavior, the TCP overhead is limited to acknowledgement (ACK) messages, which only grow in proportion to the number of application-level messages.

5.5. Replies and Queries. O2 has no built-in query system, and, normally, O2 messages do not need replies. Queries have been proposed for OSC but never became widely used, indicating this is not a critical feature. 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.

Although O2's discovery protocols reveal all the active services in an ensemble, there is no facility to query the namespace of each service or find the parameter types or documentation. However, a directory service was implemented as an O2 service, allowing any other service to register addresses and descriptions [23].

5.6. Address Pattern Matching and Message Delivery. An option in both OSC and O2 is the use of "wildcards" and patterns in addresses, allowing a single message to control multiple parameters. For example, the address `/synth/chan*/alloff` can be used to send a message to `/synth/chan01/alloff`, `/synth/chan02/alloff`, ..., `/synth/chan16/alloff`, assuming these addresses all exist. 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. Performance

O2 is implemented in the C programming language for portability. Performance measurements show that CPU time is dominated by network send and receive time, even when messages are sent to another process on the same host (no network link is involved). Table 2 summarizes measurements where two processes send a message back and forth 2 million times. The fastest time is with O2 and TCP. Perhaps TCP slightly outperforms UDP in these tests because a stateful connection can cache routing or other information. The OSC over TCP performance was about half that of O2. This is likely due to the fact that OSC connections are one-way, and thus *two* TCP connections were opened to send messages back and forth. This prevents acknowledgement (ACK) signals from "piggy-backing" on data packets, doubling the total number of packets. These measurements were run on a 3 GHz Intel Core i7 processor, running OS X v.10.13.6. The main

Table 2: Small message send time (just the destination address and a 32-bit integer) for O2 versus OSC and TCP versus UDP. The same communication was also implemented directly in TCP and UDP without any additional layers, sending only one 32-bit integer per message. Run times are wall time, with all messages between two processes on the same host. Averages from multiple runs are reported. Individual runs vary by ±1.5%.

	UDP		TCP	
	Time/Message	Messages/Second	Time/Message	Messages/Second
OSC	29 μs	35,000 s^{-1}	56 μs	18,000 s^{-1}
O2	30 μs	34,000 s^{-1}	28 μs	36,000 s^{-1}
Direct	22 μs	46,000 s^{-1}	20 μs	44,000 s^{-1}

conclusion is that O2 features have a negligible impact on performance relative to OSC.

Clock synchronization is difficult to measure. Any technique that can accurately compare clocks on remote machines can be used to synchronize them! However, we can get a good idea of how well clocks are synchronized by observing *estimated* clock differences that are produced by the clock synchronization protocol itself. For example, if the protocol estimates the clock is behind by 3 ms, then the actual clock error is probably 3 ms or less (e.g., it could have been behind by 1.5 ms and is now set to be ahead by 1.5 ms). We ran O2 for 2 hours (740 clock sync periods) on two personal computers sharing a wireless hub/modem that was also being used for Internet access. The median round-trip time was 5.5 ms, but there were 94 round trip times in excess of 100 ms. Nevertheless, the maximum absolute *estimated* clock difference was only 21 ms. The median correction was 0 ms (times were recorded in whole milliseconds). Considering that the just-noticeable difference for rhythmic timing is about 10 ms [22], we conclude that the clock synchronization performance is adequate for music applications, but the algorithm could probably be improved by detecting outliers in the round-trip time.

7. 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 (http://www.bluetooth.org), MIDI [1], or ZigBee (http://www.zigbee.org), although each of these requires extensions to be implemented within the O2 library. Finally, it is possible to create servers to bridge between O2 and other protocols, as illustrated by a WebSockets bridge server.

7.1. Receiving from OSC. To receive incoming OSC messages, call

`o2_create_osc_port(`*service, port*`);`

which tells O2 to begin receiving OSC messages on *port*, directing them to *service*, which is normally 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 are converted into O2 timestamps before messages are handed off.

7.2. Sending to OSC. To forward messages to an OSC server, call

`o2_osc_delegate(`*service, ip_address, port, tcp_flag*`);`

which tells O2 to create a virtual service (name given by the *service* parameter) to convert incoming O2 messages into OSC messages and forward them to the given *ip_address* and *port*, using a TCP connection if *tcp_flag* is set. Now, any O2 client on the network can discover and send messages to the OSC server.

7.3. Other Transports. Handling O2 messages from other communication technologies poses two interesting problems: What to do about discovery, and what exactly is the protocol? Our goal is to allow the O2 API to be supported directly on clients and servers connected by non-IP technologies. We do this by having an O2 process forward messages to and from non-IP hosts. As an example, let us assume we want to use O2 on a Bluetooth device (we will call it Process D; see Figure 2) 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 2. The same approach is used for other transports such as ZigBee or serial links such as RS-232.

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 2 can be addressed by "`128.2.60.110:8000:bt1`."

"Other transports" are not limited to networks. Recent work has explored the use of shared-memory lock-free queues to send O2 messages to high-priority threads in the same process, providing synchronized communication without locks. This is particularly useful in real-time music audio

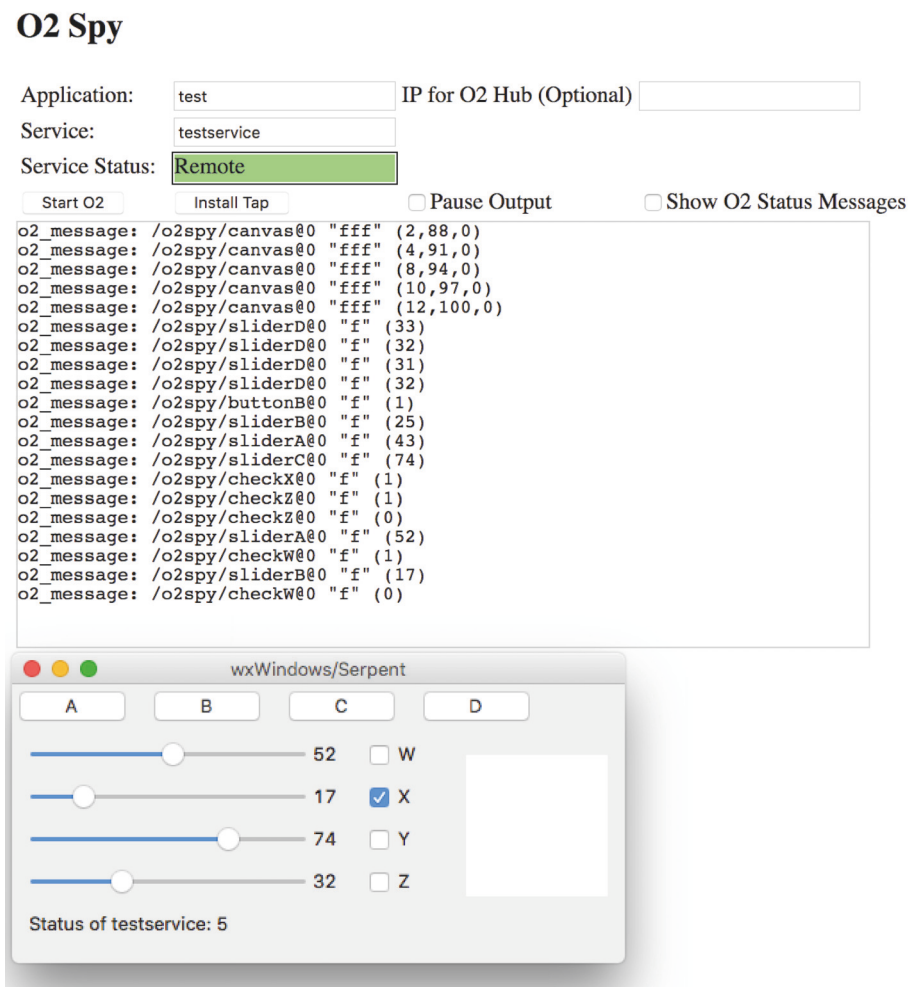

Figure 4: Web pages can access O2 via a WebSockets interface to a local server. Here, a web page is used to tap into messages delivered from a graphical control panel process (at bottom) to a remote receiver process (not shown).

applications where locks are typically forbidden within audio computation threads or callback functions. Applications will see audio computation as an O2 service that can be addressed in the normal way. Messages will be delivered by TCP/IP to the right process, and, from there, messages can be forwarded to the high-priority audio service by appending them to a lock-free queue.

7.4. Language Support. O2 is currently implemented in the C programming language for portability and to simplify linking with programs written in other languages. Serpent [24], a real-time scripting language developed especially for interactive music applications, includes O2 in the standard release. O2 has also been incorporated into Kronos [25] and used to create a network-based audio synthesis server. In this system, real-time audio is streamed via O2 messages [23].

7.5. WebSocket Support. An interesting recent development is a server that enables O2 access from web pages using WebSocket technology. The server, written in Serpent, is a lightweight HTTP server with WebSocket capability. The server is normally run on the local host along with a web browser. Any page loaded into the browser can open `ws://localhost:8080/` to create a WebSocket connection to the local server that also runs O2. A simple protocol is implemented over a WebSocket to enable the web page to join an O2 ensemble, create services, and send and receive O2 messages, including OSC messages. The use of WebSockets adds an extra hop to message delivery but avoids the security and practical problems of writing extensions for a variety of web browsers.

Figure 4 illustrates a web-based application that can monitor remote O2 message delivery using the `o2_tap` function described earlier. The O2 monitor application, implemented in JavaScript and HTML, connects over a WebSocket to the local server that is running as an O2 process.

The WebSocket interface to O2 also creates the possibility for applications written in other languages such as Python, Java, C#, or Ruby to access O2 through existing WebSocket libraries rather than creating a "foreign function interface" library for O2 in each language.

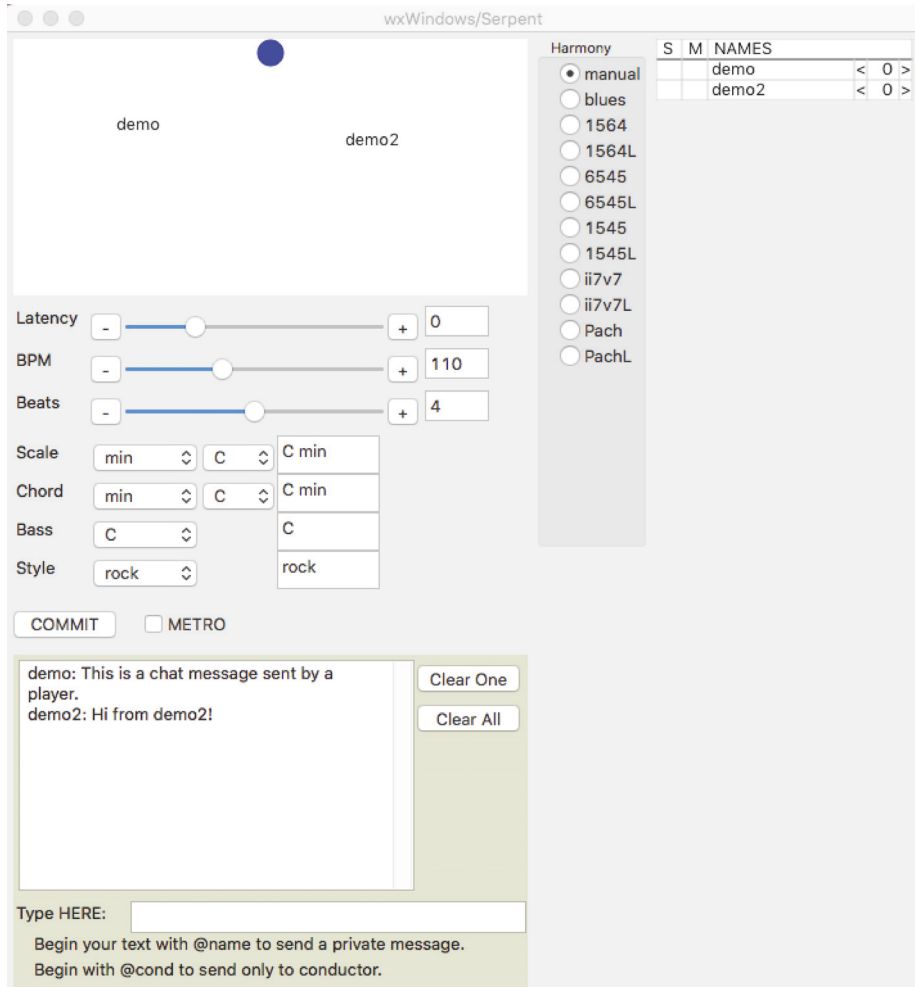

Figure 5: Control panel for the Conductor of the CMU Laptop Orchestra. The top left window is a map of the stage showing the locations of all the players (only 2 are shown here). The circle moves to indicate the beat and tempo. At middle left are controls for tempo, meter, harmony, and style, which are sent to all players when the COMMIT button is pressed. At the bottom is a chat window allowing players to communicate during the performance. The conductor also has a number of standard chord progressions that can be selected below "Harmony" (top, center). When players join the ensemble, their names are displayed in a list. To the left of each name is a solo (S) and mute (M) button; to the right are softer (<) and louder (>) buttons surrounding the current loudness offset (0 in this example).

8. Example

One substantial example of O2 in practice has been recent performances by the CMU Laptop Orchestra. Originally built around TCP/IP and OSC, this networked performance uses a central *conductor* to send tempo, key, meter, and style information to around 20 to 25 client computers (see Figure 5). Each client is a semiautonomous *player* that follows the musical structure and constraints of the conductor but also has real-time controls operated by a human (who is also the creator/programmer of the player). Players fill different musical roles such as bass, melody, arpeggiator, chordal accompanist, or drummer, and controls include mobile devices running TouchOSC (https://hexler.net/software/touchosc) and connected over Open Sound Control. A human "semiconductor" can change tempo, meter, and key and also mute, unmute, and adjust the volume of individual players.

Players initially send their names to the conductor service, which keeps a list of active players by checking their status periodically. To obtain musical synchronization, the conductor expresses beat times as a linear function: the time for beat b, $f(b) = a + b/s$, where a is the (theoretical) time of beat zero, and s is the tempo in beats per second. Only a and s need to be delivered to clients when the tempo changes, and the delivery time is not critical since a and s are not time dependent. All players compute the same value $f(b)$ for each beat, and all players have synchronized clocks, so beats are accurately synchronized. In fact, the main impediment to audio synchronization is variability in the latency of various software synthesizers and audio device drivers. Each player schedules output ahead of time according to an audio-latency compensation parameter that users can adjust, resulting in synchronization to within a few milliseconds. Readers can view performances online at https://youtu.be/icLUJMM-11M

and https://youtu.be/L-Sar4D7lIY. These performances used WiFi to simplify the setup.

9. 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. O2 offers two classes of messages so that "commands" can be delivered reliably, and sensor data can be delivered with minimal latency. In addition, O2 offers a standard clock synchronization and time-stamping system that is suitable for local area networks. We have implemented O2 and shown that its speed is comparable 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, RS-232, or other communication links, and how a WebSockets-to-O2 bridge server can open O2 applications to web browsers.

A number of extensions are possible, and future work includes extensions for audio and video streaming, and dealing with network address translation (NAT). We are also working on "externals" for Pd [26] and Max/MSP [27], which are widely used development platforms in the computer music community. As Zeroconf (Bonjour) becomes standard, we believe we can abandon our self-contained discovery system in favor of a standard one. O2 has been run in networks of 25 hosts, but it would be interesting to measure performance on larger networks, at least up to 100 hosts. Overall, we believe O2 is a good candidate for OSC-like applications and a variety of networked mobile and IoT devices in the future.

Data Availability

The source code for O2 is available for commercial and noncommercial use at https://github.com/rbdannenberg/o2. The source code and executable versions of the Serpent programming language are available for commercial and noncommercial use at https://sourceforge.net/projects/serpent/.

Disclosure

This paper is an extensively revised and extended version of an earlier conference publication.

Conflicts of Interest

The author declares that there are no conflicts of interest regarding the publication of this paper.

Acknowledgments

Thanks are due to Adrian Freed for comments on an earlier paper. Zhang Chi contributed to the initial implementation of O2 and Hongbo Fang implemented a WebSockets protocol in Serpent. O2 has developed and evolved through many interactions with students, visitors, and faculty in the School of Computer Science at Carnegie Mellon University.

References

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

[2] 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)*, pp. 153–159, Montreal, Canada, 2003.

[3] M. Wright, A. Freed, A. Lee, T. Madden, and A. Momeni, "Managing complexity with explicit mapping of gestures to sound control with OSC," in *Proceedings of the International Computer Music Conference*, pp. 314–317, International Computer Music Association, Habana, Cuba, 2001.

[4] E. Lynch and J. Paradiso, "Sensorchimes: musical mapping for sensor networks," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 137–142, Brisbane, Australia, 2016.

[5] R. Fiebrink, P. R. Cook, and D. Trueman, "Play-along mapping of musical controllers," in *Proceedings of the 2009 International Computer Music Conference, ICMC*, pp. 61–64, Canada, 2009.

[6] J. Malloch, S. Sinclair, and M. M. Wanderley, "Distributed tools for interactive design of heterogeneous signal networks," *Multimedia Tools and Applications*, vol. 74, no. 15, pp. 5683–5707, 2015.

[7] G. Essl, "Automated ad hoc networking for mobile and hybrid music performance," in *Proceedings of the International Computer Music Conference 2011*, pp. 399–402, Huddersfield, 2011.

[8] D. Trueman, P. Cook, S. Smallwood, and G. Wang, "PLOrk: the Princeton Laptop Orchestra, year 1," in *Proceedings of the International Computer Music Conference, ICMC 2006*, pp. 443–450, 2006.

[9] R. B. Dannenberg, S. Cavaco, and E. Ang, "The Carnegie Mellon Laptop Orchestra," in *Proceedings of the 2007 International Computer Music Conference*, vol. II, pp. II-340–II-343, The International Computer Music Association, ICMA, San Francisco, USA, 2007.

[10] G. Hajdu, "Embodiment and disembodiment in networked music performance," in *Body, Sound and Space in Music and Beyond: Multimodal Explorations*, C. Wöllner, Ed., pp. 257–278, Routledge, Abingdon-on-Thames, 1st edition, 2017.

[11] R. B. Dannenberg and T. Neuendorffer, "Scaling up live internet performance with the global net orchestra," in *Proceedings of the 11th Sound & Music Computing Joint with the 40th International Computer Music Conference*, pp. 730–736, Athens, Greece, 2014.

[12] S. Gresham-Lancaster, "The aesthetics and history of the hub: the effects of changing technology on network computer music," *Leonardo Music Journal*, vol. 8, pp. 39–44, 1998.

[13] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, no. 03, p. 193, 2005.

[14] E. Brandt and R. B. Dannenberg, "Time in distributed real-time systems," in *Proceedings of the International Computer Music Conference*, 1999.

[15] M. Henning, "The rise and fall of CORBA," *Queue*, vol. 4, no. 5, pp. 28–34, 2006.

[16] E. Guttman, "Autoconfiguration for IP networking: Enabling local communication," *IEEE Internet Computing*, vol. 5, no. 3, pp. 81–86, 2001.

[17] A. Eales and R. Foss, "Service discovery using open sound control," in *Proceedings of the AES 133rd Convention 2012, AES*, pp. 348–354, San Francisco, USA, 2012.

[18] J. Narveson and D. Trueman, "LANdini: a networking utility for wireless LAN-based laptop ensembles," in *Proceedings of the Sound and Music Computing Conference (SMC)*, Stockholm, Sweden, 2013.

[19] F. Cristian, "Probabilistic clock synchronization," *Distributed Computing*, vol. 3, no. 3, pp. 146–158, 1989.

[20] S. Madgwick, T. Mitchell, C. Barreto, and A. Freed, "Simple synchronisation for open sound control," in *Proceedings of the 41st International Computer Music Conference*, pp. 218–225, Denton , TX, USA, 2015.

[21] K. Thomas, "Just Noticeable Difference and Tempo Change," *Journal of Scientific Psychology*, 2007.

[22] A. Friberg and J. Sundberg, "Perception of just-noticeable time displacement of a tone presented in a metrical sequence at different tempos," *STL-QPSR*, vol. 34, no. 2-3, pp. 49–56, 1993.

[23] V. Norilo and R. B. Dannenberg, "KO2 distributed music systems with O2 and Kronos," in *Proceedings of the 15th Sound and Music Computing Conference (SMC2018)*, 2018.

[24] R. B. Dannenberg, "A language for interactive audio applications," in *Proceedings of the 2002 International Computer Music Conference*, pp. 509–515, International Computer Music Association, San Francisco, USA, 2002.

[25] V. Norilo, "Kronos: a declarative metaprogramming language for digital signal processing," *Computer Music Journal*, vol. 39, no. 4, pp. 30–48, 2015.

[26] M. Puckett, "Pure data," in *Proceedings of the International Computer Music Conference*, pp. 224–227, International Computer Music Association, San Francisco, CA, USA, 1996.

[27] M. Puckette, "Max at Seventeen," *Computer Music Journal*, vol. 26, no. 4, pp. 31–43, 2002.