Roger B. Dannenberg
Carnegie Mellon University
School of Computer Science
5000 Forbes Avenue
Pittsburgh, Pennsylvania 15213, USA
rbd@cs.cmu.edu

Communication for Real-Time Music Systems: An Overview of O2

Abstract: Message passing between processes and across networks offers a powerful method to integrate and coordinate various music programs, facilitating software reuse, modularity, and parallel processing. Networking can integrate components that use different languages and hardware. In this article we describe O2, a flexible protocol for communication ranging from the thread level up to the level of global networks. Messages in O2 are similar to those of Open Sound Control, but O2 offers many additional features, including discovery, clock synchronization, a reliable message delivery option, and routing based on services rather than specific network addresses. A bridge mechanism extends the reach of O2 to web browsers, shared memory threads, and small microcontrollers. The design, implementation, and applications of O2 are described.

Dismantling the Monolith

In the early days of interactive music systems, it was common to dedicate an entire computer to a single real-time program so that operations could be carefully scheduled to meet real-time demands. Programs tended to be large and all-encompassing, dealing with a user interface, sensing, control processing, and (when computers became fast enough) audio signal processing. Over time, software components have become increasingly complex, and we cannot expect to find all the functionality we need in a single monolithic program. Reuse, repurposing, and integration of multiple large-scale components now form a practical approach to system building, especially for creators whose first priorities are to make music rather than to consider all details of implementation.

Fortunately, computers have evolved to support this approach. Now, nearly all computers feature many cores, allowing them to run multiple real-time applications in parallel with little interference and far fewer scheduling concerns. Larger memories have also enabled multiple applications to run in parallel without the page swapping that was disastrous for real-time music processing. Multiple coordinated software applications can take advantage of the parallelism available from multicore processors, making more computing power available. Unfortunately, "composing" software systems still requires work to establish communication, coordination, and control of multiple components.

These interconnection problems can be solved in several ways, using any of the following:

1. Any of the specialized standards such as VST, MIDI, SMPTE, Link, and DMX512, which are often associated with off-the-shelf hardware but are not very general;
2. Custom one-off solutions based on TCP/IP, RS232, ZigBee, and other low-level data transports, which are often the simplest solution when only the simplest functionality is required; or
3. Higher-level message-passing systems, including many commercial message-oriented middleware products.

In the experimental music community, Open Sound Control (OSC) is arguably the most successful solution, owing to its flexibility, simplicity, peer-to-peer connections (no third-party intermediary), and available implementations (Wright, Freed, and Momeni 2003). However, OSC lacks many desirable features.

Challenges for Software Communication

An important consideration for communication in music systems is real-time performance. Few networks offer hard real-time behavior where there are absolute guarantees on delivery times, but we can at least design for good expected performance. At least on lightly loaded networks, actual performance can then be quite predictable. Good performance

Computer Music Journal, 45:4, pp. 7–19, Winter 2021
doi:10.1162/COMJ_a_00620

depends on the right communication abstractions. For example, remote procedure calls (RPCs) that invoke an operation and return a result are a nice programming abstraction, but RPCs typically block the sender until the result is available. Because of blocking, this approach is not well suited to real-time systems. Thus, asynchronous one-way messaging, where the sender does not wait for replies, is typically used.

Assuming asynchronous messaging, what do messages look like? Systems have been designed around single-value or attribute/value messages, but network messages can carry 1,000 bytes almost as easily as 1. Moreover, collections of values are often needed to describe operations and events (even a MIDI note-on message contains a channel, a key number, and a velocity), so messages should carry multiple values. The success of OSC, with its multiple-value messages, is a good indicator that this is an important capability.

An important aspect of communication is configuring addresses and connections so that messages can be delivered to the right destination. Computer musicians must often configure raw IP (the "Internet Protocol") addresses manually because addresses are assigned by networks, and addresses can change. Even within a single computer, connections are made to ports, which are typically assigned manually to avoid conflicts. Furthermore, at least TCP connections require the server to exist before making a connection; otherwise, the client's connection request will be dropped. A great deal of effort can be eliminated through "discovery" protocols that automatically configure communications. Automatic discovery also allows services to run together on one computer for development but later run on multiple computers to achieve higher performance.

Another challenge is to achieve both good real-time performance and reliability. In practice, communication is largely based on IP, which, at a low level, is a "best effort" packet delivery system. This simple and direct point-to-point transmission usually offers the lowest latency available. Network messages (packets) can be lost, however, when operating systems or network switches become overloaded, or (in rare circumstances) when data is corrupted in transmission. Therefore, a higher-level protocol (usually TCP) is commonly used to detect errors, retransmit packets, and ultimately deliver data with essentially perfect reliability. For many applications, a combination of best-effort and reliable transmission is necessary.

The full Internet protocol suite, also known as TCP/IP, provides a widely supported and solid foundation for communication, but it is not always fully available, nor is its full set of communication features always called for. To list three examples:

1. Web browsers offer a wealth of cross-platform tools for building interfaces and data displays, but they are limited to HTTP and WebSocket APIs, which can only connect to compatible servers.
2. To achieve low latency, software music synthesizers are often barred from direct network communication and use shared-memory communication instead.
3. Microcontrollers have limited memory and power, and when used only to collect and send sensor data, a powerful middleware package and even a full TCP/IP implementation may be overly complex and power hungry.

Timing in music is essential. Rather than leave timing entirely to applications, an important capability is to synchronize clocks and deliver messages with accurate timing at their destination. Clock synchronization and timed messages allow computations to be synchronized in spite of large amounts of timing jitter caused by network latency.

To address these challenges, O2 was designed as a "communications middleware" for interactive music systems. O2 builds on some existing structures of OSC because they are widely known and successful. In the next section, the fundamentals of O2 are described. The following section, Related Work, describes other communication systems, especially those for musicians. Then in the New Features section we describe several interesting capabilities that have been added to O2 since its original implementation, including publish/subscribe and a bridge abstraction used to extend O2's reach to non-IP systems. The Implementation Details section discusses the implementation and some performance measurements. In the Applications section we describe

how O2 could be, and is, used in practice. Finally, the Future Work and Conclusions sections cover possible extensions and summarize what has been learned from building and using O2.

Introducing O2

O2 is a protocol and an implementation enabling flexible asynchronous communication between processes and across threads, especially for interactive music applications. In this section, principal O2 abstractions are described along with basic operations from the perspective of programmers and developers.

O2 connects processes. An O2 *process* is essentially a single running program. (Threads and interthread communication will be discussed later.) O2 communication is potentially global, so it is important to limit communication to within a selected group called the *ensemble*. An O2 ensemble is a collection of peer processes that are allowed to communicate with one another. Each ensemble has a distinct name, and each O2 process joins one and only one ensemble.

An O2 process can provide one or more *services.* A service is just a name used to route messages. Typically, a service is offered by at most one process, but if multiple processes offer the same service, O2 will pick one process as the active service provider and the others will serve as backups in case the active process terminates or loses its network connections.

To send a message to a service, one needs an *address*. An O2 address is a URL-like text string beginning with the service name. For example, `/synth/lfo/freq` addresses the `synth` service. The suffix nodes `lfo/freq` designate an operation or resource provided or managed by the service. Addresses in O2 are similar to OSC addresses, except that the first node in an O2 address is a service name used to find the process that offers the service, whereas in OSC, there is no service name, and it is the programmer's responsibility to send the message to the correct server.

The message format in O2 is based on OSC messages with minor changes. Every O2 message contains a timestamp that refers to globally synchronized O2 clock time. If the timestamp is greater than the current time, message delivery is delayed until the timestamp. Alternatively, the sender can use a timestamp of zero to indicate "as soon as possible." Like OSC, O2 messages also contain an address, a type string describing the types of the data contained in the message, and a set of values. O2 types include standard OSC types such as integer, float, and string, as well as some new ones, including vectors.

O2 from the Developer's Perspective

After initialization, O2 performs discovery, communication, and timed message delivery in the background. To simplify interaction with the application, all O2 operation is explicitly invoked by the application, which calls `o2_poll()` every 1 to 50 milliseconds, depending on the timing precision required. To receive messages and respond to them, the application creates a service using `o2_service_new` (*servicename*) and installs message handlers using `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, *handler* is the address of the callback function to process matching messages, and *info* is an additional parameter to pass to this handler function. The *coerce* and *parse* parameters enable options for built-in type coercion and message unpacking before invoking the handler. Messages are delivered according to their timestamps using a built-in scheduler.

Messages can be sent from any process, including the one that offers the service (in which case networking is bypassed and the handler is invoked directly.) To send a "best effort" message, one calls `o2_send` (*address*, *time*, *types*, *val1*, *val2*, . . .) with the destination address, type string, and values *val1*, *val2*, etc. To send a message with guaranteed delivery, `o2_send_cmd() is` used instead. This name suggests that the message carries a "command" that must be delivered.

What if a message is sent but there is no active service to handle it? In this case, O2 issues a

warning and drops the message. O2 will also drop timestamped messages (with nonzero timestamps) if the receiver has not established a synchronized clock. In some cases, applications will want to wait for a service to be discovered and synchronized before sending it messages. The function `o2_status` (*servicename*) can tell if a service exists, whether it is local or remote, and whether clock synchronization has been achieved.

To use clock synchronization, at least one process must provide a reference clock to the ensemble. This is done by calling `o2_set_clock()`. Optional parameters allow the clock reference to be provided by the application, such as an audio sample clock instead of the default system clock.

Notice that the application developer does not deal with IP addresses, port numbers, or even host names. O2 offers many more capabilities, described below, but before going into further detail, let us consider some related work.

Related Work

O2 originated as a project to extend OSC with new capabilities. The OSC protocol is intentionally designed to be transport-independent, but that limits it to simple point-to-point communication established by some other means. That usually requires manual configuration of IP addresses and port numbers and forces developers to choose either UDP (best effort) or TCP (reliable), but not both. Clock synchronization, discovery, and other features are missing. At least discovery has been addressed by liboscqs (liboscqs.sourceforge.io), OSCgroups (www.rossbencina.com/code/oscgroups), and osctools (sourceforge.net/projects/osctools). Discovery is also discussed and implemented by Essl (2011); Eales and Foss (2012); and Malloch, Sinclair, and Wanderley (2015).

Libmapper (Malloch, Sinclair, and Wanderley 2015) is designed to map inputs from sensors to synthesis control parameters. The model is akin to connecting systems with patch cords, which is appropriate for connecting sensors, but not for general event-based control. An unusual feature is that libmapper offers various adjustable mapping functions to transform data between the sender and the receiver, which can be particularly useful when working with sensor data.

LANdini (Narveson and Trueman 2013) has similar goals to O2 in that it offers discovery on a LAN and reliable transmission. To simplify the implementation, LANdini messages flow in three "hops," requiring a message from the sender to a local server, from the local server to a remote server, and from there to the destination. Also, the overall message rate for N devices is proportional to N^2 due to traffic used to insure reliable delivery. LANdini has inspired some recent new capabilities in O2, however.

MobMuPlat (www.mobmuplat.com; also cf. Iglesia 2016) can be described as a software framework for running Pure Data (Pd) on mobile devices. In addition to support for graphical interfaces and sensors, MobMuPlat supports a simple discovery and peer-to-peer connection scheme within a LAN, addressing some of the problems that O2 also solves.

Networking approaches outside of the music community are far more numerous. CORBA (Henning 2006) is an example of a distributed object system with many capabilities, but its complexity has discouraged its use. ZeroMQ (Hintjens 2013) is less complex and supports a variety of communication patterns, but it does not offer discovery or messaging over UDP, so it is not suitable for many music applications.

Clock synchronization techniques are well known, but often omitted from music systems because of the extra implementation and configuration required. Flaviu Cristian's (1989) simple method is the basis for synchronization in O2. Madgwick et al. (2015) describe a method that uses broadcast from a reference but assumes bounds on clock drift rates. Brandt and Dannenberg (1999) describe a round-trip method with a proportional-integral controller. OSC bundles have timestamps, but clock synchronization is rarely included in OSC implementations. Florian Goltz (2018) describes Ableton's Link technology, which uses clock synchronization specifically for the task of establishing a shared beat and tempo framework for applications.

New Features

Beyond the basic message-passing functions implemented for the first version of O2 (Dannenberg 2019), significant extensions have been implemented to address various problems or to make O2 even more versatile. Properties provide a way to share information about services. Taps allow communications to be monitored and support the publish/subscribe communication pattern. Bridges allow O2 hosts to be connected to processes through protocols other than IP. Most bridges use a subset of O2 called O2lite, which runs over WebSockets, IP, and shared memory interfaces.

Properties

Inspired by LANdini, O2 *properties* are attribute/value pairs associated with service providers. An important use case is finding players in a laptop orchestra. Each player will be represented by a different O2 service name, so how does a central "conductor" process send messages to all players? Note that in O2, each player could offer a service named `player`, but O2 would direct messages to only one of them. Thus, each player must be reached via its own unique service name. Alternatively, the conductor cannot simply send to every service because not all services are players. The solution is for each player to attach a property, for example, `type:player`, to its service. Then, the conductor can search for services with a `type` attribute equal to `player` to locate players. Properties are stored in strings, and copies of property strings are distributed by the O2 discovery mechanism, so they can be accessed or searched quickly without additional network delay.

Taps and Publish/Subscribe

Messages arriving at a particular service can be forwarded automatically to another service at any process by setting a *tap*, which consists of a process and a service name within that process. The destination service (the "tapper") is associated with a specific process so that if the process is disconnected, the tap can be removed automatically rather than have it possibly redirect to another service provider.

Taps were created to enable message monitoring and diagnosis, but they serve another role that is perhaps more important. A publish/subscribe pattern is one in which the sender publishes using some publishing name, and receivers can subscribe to the name in order to receive messages. This pattern allows the publisher (consider a sensor or global tempo control) to send information without knowledge of who is interested in that information or who will receive the messages. It is also a one-to-many pattern in contrast to the many-to-one pattern implemented by `o2_send()` and services. To implement publish/subscribe, the publisher creates a local service, which need not have any message handlers. Publishing means simply sending to the service. Each subscriber *taps* the service to receive a copy of each published message. This scheme eliminates any need for the application to manage a subscriber list, including the automatic removal of tappers when network connections are lost. (Alert readers will notice that publish/subscribe offers another solution to the problem of connecting a conductor to many players.)

The Bridge Abstraction

To be as versatile as OSC, O2 should be extensible to work with new transports such as Bluetooth or WebSockets. The *bridge* abstraction allows services to be provided across any message-passing transport. A bridge connects a single O2 process, called the host, via a message transport, to a single process, the client. The host advertises services on behalf of the client. O2 messages for those services arrive at the host and are immediately forwarded to the client. The client can send arbitrary O2 messages by sending them across the link to the host, where they are resent to reach their ultimate destination. Essentially, the host is both a proxy service provider for the client and a proxy for sending O2 messages. In this way, O2 can be extended to support any link technology, including RS232, Bluetooth, Zigbee, WebSockets, and shared memory.

O2lite

Most bridges implement a subset of O2 called O2lite. There is even an O2lite bridge for TCP/IP, allowing for lightweight implementations on microcontrollers running Wi-Fi such as ESP32-based devices. In this case, the difference between O2 and O2lite is not the transport, since both use IP, but the fact that O2lite can only send and receive directly from a single O2 host process. Messages to other processes must be relayed through the host. On the other hand, a small and simple implementation that is more suited to microcontrollers can be used, leaving the rest of O2's functionality to laptop or desktop host computers.

Another implementation of O2lite uses WebSockets for the transport. WebSockets allows browsers to connect to an O2 ensemble. Thus, browsers can be used as synthesizers or user interfaces. Mobile devices with accelerometers and touch surfaces can communicate with O2 through their built-in web browsers. A library, `o2ws.js`, exists in Javascript for these applications. Browser applications, typically written in HTML and Javascript, must be downloaded over the HTML protocol. To be self-contained, especially on private LANs often used in performances or art installations, the O2 library implements a simple web service. Users can download web applications and connect to O2 without creating a separate server or website.

A third interesting implementation of O2lite runs over a shared memory bridge. This is one solution to the problem that O2 does not directly support multiple threads. Even if it did, synchronizing threads and invoking network operations could introduce unacceptable latency to real-time threads, especially those used for audio signal processing. The shared memory bridge uses lock-free queues to pass messages between threads. Similar lock-free queues are used to allocate and free memory through a shared heap structure, supporting very-low-latency audio processing. In our tests, the shared memory bridge can send and receive a message using 320 nsec of central processing unit (CPU) time plus a small overhead to poll for messages. An application of this bridge is described later in the Audio Synthesis section.

OSC Compatibility

One transport of great interest is OSC. To interoperate with OSC, O2 uses a bridge-like mechanism that translates to and from OSC messages. To receive OSC, one calls `o2_osc_port_new` (*servicename*, *port*, *tcpflag*) to create an OSC server that receives incoming messages on the given *port* and forwards them to the specified O2 service. For example, if *servicename* is `sensor1`, and the incoming OSC address is `/value`, the message is forwarded to O2 address `/sensor1/value`. To send to an OSC server, one calls `o2_osc_delegate` (*servicename*, *ipaddress*, *port*, *tcpflag*), which creates a new O2 service. Any O2 message to that service is translated to OSC by removing *servicename* from the address and is forwarded to the OSC server specified by IP address, port number, and the protocol indicated by *tcpflag* (TCP or UDP).

Wide Area Networking with O2

O2 uses zero-configuration networking implementations (Bonjour on macOS, Avahi on Linux) for discovery (cf. Guttman 2001). One limitation of Bonjour is its use of broadcast messages, which are restricted to the local area network. This means that distant machines cannot be discovered. Given the interest in network performance and collaboration, O2 implements an extended discovery protocol that works globally. Global discovery builds upon MQTT (mqtt.org), a lightweight Internet-of-Things messaging protocol for which there are open servers that can be located through conventional domain name resolution.

The MQTT protocol offers publish/subscribe services. Processes in O2 construct a topic `o2-`*ensemblename*`/disc`, based on the ensemble name, and publish their local and public IP addresses and port number through MQTT. O2 processes continually listen for MQTT messages that announce new members of the ensemble. When a new process is discovered, a direct TCP connection is attempted to enable further communication.

An often-encountered problem is receiving messages behind Network Address Translation (NAT)

firewalls, which are almost standard for home networks. The NAT mapping translates port numbers and IP addresses as they transit from local home networks to the public Internet. In principle, the only way to receive a message behind NAT is to connect to a server and specify a reply port. The reply port in the outgoing message is replaced with a different number, but NAT updates tables so that any reply message from the Internet can be directed to the true reply port on the home network. Thus, outgoing requests to servers work transparently. On the other hand, a connection originating from the Internet will not reach the desired address or port through the firewall. There are interesting workarounds, including STUN (dl.acm.org/doi/book/10.17487/RFC3489), but O2 currently solves this problem by relaying messages through MQTT using the same server used for discovery. It is only when direct peer-to-peer connections cannot be established that MQTT is used.

Implementation Details

O2 is implemented in C++ but has a purely procedural API accessible from C or any language that can interface with external libraries. O2 currently runs on macOS, Linux, and Windows, and is free and open source (github.com/rbdannenberg/o2). O2 can also be accessed through Pd using a set of four Pd objects ("externals"): `o2ensemble`, `o2send`, `o2receive`, and `o2property`.

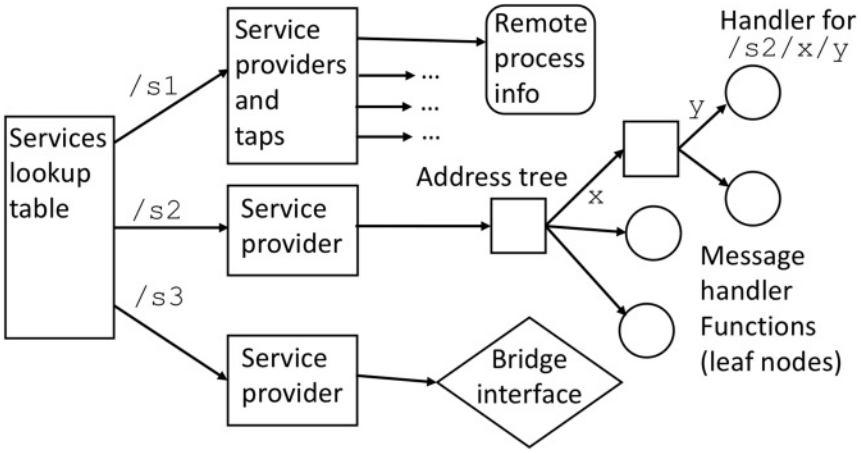
Discovery is based on Bonjour, Avahi, or MQTT, as described earlier. Upon discovery of another process, O2 determines if a connection already exists. If not, a TCP connection is made to the process. To avoid two peers trying to connect to one another simultaneously, permanent connections are only made from lower address and port numbers to higher ones. To connect to a peer with a lower address, a temporary connection is made, and a discovery-like message is sent requesting a connection in the reverse direction.

Once made, TCP connections are bidirectional, and processes can reliably exchange their local services, properties, and UDP ports that are needed for "best effort" messages. All this information is sent using ordinary O2 messages to the specially recognized `o2_` service. As a peer-to-peer network, the number of connections grows as N^2 for N processes, but we assume O2 ensemble size is limited to at most 100 processes. Maintaining 100 network connections per process is small-scale networking in terms of modern servers. Of course, performance will depend upon message rates and network capacity, but O2 has been used successfully over Wi-Fi for a laptop orchestra with 25 processes and many more devices connected via OSC.

Message delivery operates as follows: First, the message address is examined to obtain the destination service name. O2 uses a hash table to map the service to a list of service providers and a list of taps (see Figure 1). The service providers are sorted by decreasing process address, and the first provider is considered the active provider that will receive the message. If the service provider is a remote process, the message is forwarded over TCP or UDP directly to that process for delivery. If the active provider is the local process, there will be a tree of hash tables used to decode the full address and determine a handler function to call. If the active provider is a bridge, OSC server, or MQTT connection, the message is delivered using the corresponding protocol. After delivery to the active provider, a copy of the message is delivered to each tap, if any, replacing the original service name with that of each tap before forwarding.

The implementation of clock synchronization requires one process to provide a clock reference that other processes attempt to follow. Each so-called follower sends periodic requests for the reference time. For each follower, the fastest round-trip time of the five most recent requests is used to estimate any difference between the follower's local clock and the reference clock, and local corrections are made. The implementation in O2 makes smooth clock adjustments by temporarily changing the clock speed, eliminating rhythmic distortions in music that might be caused by suddenly setting the time forward or backward. O2 runs an efficient scheduler to deliver timestamped messages. Applications can use the scheduler by sending themselves timestamped messages to initiate timed events.

Figure 1. To deliver messages, each process first maps the service name (first node of the address) to a list of service providers and taps. A service provider is either: a remote-process object encapsulating an open TCP socket, UDP address, and connection state (top); a local service represented as a tree to decode the address, e.g., /s2/x/y, mapping it to a handler function pointer or object (middle); or a bridge interface (or MQTT or OSC) instance, which implements a specialized transport mechanism (bottom).



The first implementation of O2 used blocking network send calls. Typically, send operations copy messages immediately to buffers and return, so in practice, calls rarely block. However, deadlock can occur when two processes are sending to each other over TCP. Now, O2 uses nonblocking calls, and senders can detect and avoid sending more messages than the network can handle.

Performance and Evaluation

Communication software such as O2 should provide useful functions without introducing high overhead or performance penalties. In practice, sending a network message is already quite slow and expensive, so there is not much one can do that will make network performance significantly better or worse. For example, we measured a median round-trip time of 5.5 msec over a local Wi-Fi network, but within a single computer, the time averaged 28 μsec, indicating that more than 99 percent of the time is due to Wi-Fi. The single computer round-trip time could be reduced to 20 μsec by calling network primitives directly instead of using O2. This indicates a single message send overhead of about 4 μsec, but that number (in this simple test) includes polling all sockets for messages, running the scheduler, and other tasks for each message.

The shared-memory bridge uses the same message decoding implementation as regular O2 messages but avoids networking altogether. By comparing the time to send and deliver a single message to the time to send multiple messages back-to-back, we can factor out the overhead of polling operations and determine that the time to allocate memory, write a message, send it, decode it, call its handler, and finally free the memory is about 320 nsec. This is a likely a best-case scenario because repeatedly performing the same small task will ensure few cache misses. Also, keep in mind that this is a measurement of CPU utilization, whereas the real time delay is limited by the polling period between checks for new messages, typically on the order of 1 msec.

Direct comparisons with OSC (using the liblo implementation) show negligible differences in performance except in one case. O2 uses a single bidirectional TCP connection, whereas OSC uses a one-way connection, so two connections are required for two-way communication. In our tests, OSC over TCP using two connections was exactly half as fast as O2 using a single TCP connection, but we would expect one-way send times to be nearly identical. Again, these times are swamped by network latency when actual networks are involved.

Network configuration with O2 typically takes from one to a few seconds, including clock synchronization. Normally, clock synchronization and Bonjour discovery messages are infrequent, but when O2 is initialized, it actively contacts Bonjour

and O2 processes to make connections quickly. When the reference clock is first discovered, clock synchronization runs on an accelerated schedule to reduce the time to estimate the reference clock time.

Evaluation should also include ease of use, suitability, and generality. These are "soft" attributes that can best be evaluated with time and experience. We believe O2 is a strong candidate for computer music applications because it has grown out of experience with many system implementations, including global network music performances, laptop orchestras, wireless sensors, audio servers, and single applications with communicating processes. We hope the community will find O2 to be as useful as we do.

Applications

To illustrate the potential of O2, we describe several applications with a focus on how O2 can support their construction.

Laptop Orchestras

The first application of O2 was in a performance created by students for laptop orchestra. Discovery allowed for rapid prototyping, testing, and performance setup in which "player" laptops connected to a "conductor" laptop. The conductor established a shared context including tempo, style, and other controls for players to interpret. Musical time was represented as a function of real time, with a few parameters transmitted by O2, namely

$$\text{beat}(t) = \textit{beatoffset} + (t - \textit{timeoffset}) \times \textit{beatspersecond}.$$

Because time t is based on O2 clock synchronization, every laptop was able to compute the current beat with high precision and schedule beat-based rhythmic output accordingly. Timed messages were able to change tempo by sending new parameters at precise times. After compensating for local synthesizer delays, synchronization was limited mainly by the variation in distance from speakers to audience members at different locations.

Control from the conductor was complemented by local control by each player, including the use of TouchOSC (hexler.net), which sends touch screen data from mobile devices over Wi-Fi via OSC. O2's OSC compatibility allowed players to incorporate TouchOSC data easily.

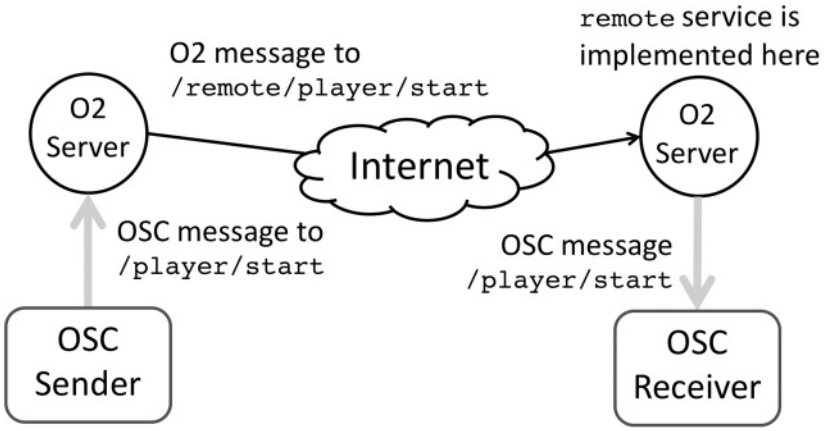
O2 over WebSockets was used to connect to animations written in p5js (http://p5js.org) and running in a browser. Parameters from the conductor including tempo information were shared with the animations to synchronize them to the music. The performance can be viewed at http://youtu.be/3OYhC3KNt-g.

Sensors and Synthesis

The original design of O2 anticipated that even microcontrollers would run a full network stack and act as full O2 processes. This is certainly possible with Raspberry Pi and other small Linux-based systems, but we also felt a need for a lighter-weight and simpler implementation, which led to the development of O2lite. We use O2lite on ESP32-based microcontrollers and the Arduino development environment, which includes an implementation of Bonjour for discovery. Recall that with O2lite, a single O2 process serves as a bridge to the entire network.

An example application is a small self-contained inertial sensor that communicates over Wi-Fi. Without O2, one would hard-code a destination IP address for data into the microcontroller. To use the sensor, one would manually disconnect from the Internet and set a laptop IP address to match the one in the sensor. With O2, the sensor can discover the laptop's dynamic IP address, and the two-way capability of O2 allows the laptop to configure the sensor by setting the sample rate and other parameters. Once data is received at the laptop, software can map the sensor data to sound controls and send them, either via O2 or OSC, to a synthesizer to implement interactive gestural control. An example can be viewed at https://youtu.be/cPQQiYs2xeY.

Figure 2. Bridging OSC over the Internet. An OSC Sender sends to a local O2 Server (left), which discovers the remote service and forwards OSC messages reliably. The remote service converts messages back to OSC and sends them to a local OSC receiver. Thus, OSC is delivered reliably across the Internet without any manual configuration of IP addresses and ports. If OSC processes are behind firewalls, O2 will automatically revert to using an MQTT broker to forward messages.



Network Music

Network music (McKinney 2016) often involves collaborative control of music-generation systems and graphical displays, as well as sharing of sensor data. O2 makes it easy to bridge multiple sites, especially computers on home Wi-Fi networks behind NAT, as discussed above. O2 can even help to bridge existing OSC-based systems by receiving OSC messages in a local O2 process, forwarding the messages to a remote site also running O2, and from there, forwarding from O2 to an OSC system as a final destination (see Figure 2). Because O2 can use reliable transmission across the Internet, not only does this solve problems with configuration and NAT, but it also can eliminate dropped packets, which occur frequently with wide-area networking. This approach parallels our "Telematic Soundcool" performance (Scarani et al. 2019), which was implemented directly with TCP before O2 was available. Even audio streaming is possible with O2 (Norilo and Dannenberg 2018), but it may be simpler to implement audio/video streaming with more-specialized programs and just handle control information with O2.

Interactive Installations

Computer-based art installations, including those that feature multimedia and interaction, often use multiple computers and microcontrollers as sensors, for control and for output. When existing Wi-Fi networks are used, artists and staff are faced with manual configuration tasks or even reprogramming microcontroller programs with hard-coded IP addresses. O2 simplifies connections through discovery while offering flexible open-ended messages to transmit whatever data is needed to support sensing and control. Clock synchronization can be used to coordinate different media controlled by different processes.

Another possibility is remote monitoring and control: Artists cannot always keep an eye on installations in person, so some build monitoring capabilities into their systems in order to oversee the operational status remotely. O2's global-scale discovery, taps, and other facilities simplify making connections from anywhere and for monitoring any message activity within an installation's computing ensemble. Any service can be messaged to invoke operations or query data. O2 has not yet been used for this purpose, but a prototype for remote monitoring has been built and described (Dannenberg 2019); Figure 3 shows an example of how this can be used.

Audio Synthesis

Audio processing in software requires that latencies be on the order of 1 msec or less, which is near the limit of what consumer operating systems can offer. Real-time operating systems and specialized hardware can do better, but software and hardware

Figure 3. The O2 Spy program uses O2's tap facility to "spy" on services. In this demonstration, an interface process manages the Buttons window (lower right). Controls send O2 messages to another process offering the synth service. The O2 Spy program, running in a browser and hosted by a third process, can join the ensemble, display the status of the synth service, and "tap" the messages it receives. Here we see the messages produced by clicking Buttons 1 and 2, adjusting the slider, and clicking Button 3.

choices (including audio interfaces and device drivers) are more limited and often more expensive. Assuming a consumer operating system, effective low-latency audio processing requires developers to refrain from making most system calls, precluding normal memory allocation or locks to safely read and write shared data. Calls within the audio thread to send and receive network messages are out of the question.

Developers can either carefully cope with all these restrictions, or violate them and hope for the best. Alternatively, O2 uses its own lock-free memory allocator, and O2's shared-memory bridge uses lock-free message queues to provide the flexibility of O2 messaging to low-latency audio threads. Audio processing can be tightly coupled to a control thread (an O2 process) using O2 messages delivered through shared memory, or the O2 process can connect to other processes running on the same machine, on a LAN, or even globally. Controlling an audio process in this way is comparable to the use of OSC for control in the SuperCollider synthesis server (Wilson, Cottle, and Collins 2011), but O2 provides scheduling, discovery, and clock synchronization in addition to messaging. A new experimental synthesis engine based on O2 and FAUST (http://faust.grame.fr) has been constructed to explore this direction further.

Modular Performance Systems

O2 was largely inspired by a project to support human–computer music performance by using software modules such as MIDI players, audio players that can time-stretch to synchronize audio

playback, conductors that control the players, and score displays that show music notation to human performers, as well as sensors and hardware or software synthesizers. The need for communication, coordination, and timing among distributed music software components led to implementations with OSC, ZeroMQ (Hintjens 2013), and ultimately O2. With O2, modules operate standing alone, but if a conductor is discovered, the modules automatically connect to it and delegate control over start, stop, "set position," and tempo operations. Clock synchronization is critical for coordinating players, and publish/subscribe is used to distribute conducting commands to all players.

Our vision is that future music performance systems will include intelligent agents as performers, and modular systems will be easy to set up, just as today's performing musicians combine microphones, guitars, effects pedals, mixers, amplifiers, and speakers using off-the-shelf compatible components.

Future Work

O2 is now fully functional, but there are many areas for further development. Additional language support would encourage more use. O2lite can be ported to most languages. For applications to use the full O2 implementation, it should be possible to link to the existing O2 implementation in C++, which we have done for Pd. Debugging distributed systems can be difficult, and some monitoring software has been prototyped but is not ready for practical use. Perhaps O2 could include more internal monitoring of latency, CPU utilization, and other useful information.

Security is a difficult problem, especially for real-time systems. O2 does not encrypt or protect data, so it is quite possible for an attacker to snoop network packets, discover an ensemble name, and inject malicious messages. For this reason, O2 does not enable global discovery by default. Virtual private networks (VPNs) are one way to secure O2 communication, but this is likely to affect latency. Built-in security measures should be based on a clear threat model and are left to future work.

Audio and video streaming have many applications, ranging from modular effect chains within a single computer to network music performances on a global scale. Audio over O2 has been implemented (Norilo and Dannenberg 2018), but it is a complex problem with many conflicting goals. Perhaps wider adoption of O2 can inspire further work in this area.

Conclusions

O2 offers a new level of communication support or middleware, especially for real-time interactive music systems. Conceptually, O2 is similar to OSC in that it sends one-way messages containing a URL-like address that names a parameter or function and a set of typed values. However, O2 also addresses many additional practical needs of application builders. First, O2 delivers messages to services rather than to network addresses, supporting more reconfigurable and distributed systems of peer processes as opposed to simple client/server configurations. Second, O2 includes discovery to automate network configuration, allowing processes to connect even globally without fixed IP addresses, domain names, or port numbers. Third, O2 offers new modes of communication, including one-to-many publish/subscribe messages, properties that are automatically propagated without explicit messages, taps for monitoring message traffic, and shared memory communication for very-low-latency applications including audio signal processing. Fourth, O2 offers a complete solution to distributed clock synchronization and timed delivery of messages, which is important in many music applications.

An important requirement for communication software is to enable connections among diverse systems. O2's C++ API is compatible with C and therefore easily integrated with many other languages. For example, external objects allow access to O2 within Pd. O2 also interoperates with OSC, not only as an OSC server to receive messages but also as an OSC client to send messages, with translation between O2 and OSC message formats. O2lite allows a client to join an O2 ensemble over a point-to-point link. O2lite for WebSockets along with a built-in HTML server enables web browsers to

communicate with O2 processes and bridge to OSC. O2lite in C has been run on ESP32 microcomputers with Wi-Fi to create sensors that automatically connect to O2 networks. O2lite for shared memory is being used to create a new sound-synthesis server.

As computer music software systems continue to grow in capability and complexity, we hope that artists and researchers will build on these systems through creative combination, control, and interconnection using O2. It is poised to solve many interconnection problems.

Acknowledgments

O2 has developed and evolved through many interactions with students, visitors, and faculty in the School of Computer Science at Carnegie Mellon University. Zhang Chi contributed to the initial implementation. Vesa Norilo's early adoption spurred many improvements as well as the exploration of audio over O2.

References

Brandt, E., and R. B. Dannenberg. 1999. "Time in Distributed Real-Time Systems." In *Proceedings of the International Computer Music Conference*, pp. 523–526.

Cristian, F. 1989. "Probabilistic Clock Synchronization." *Distributed Computing* 3(3):146–158. 10.1007/BF01784024

Dannenberg, R. B. 2019. "O2: A Network Protocol for Music Systems." *Wireless Communications and Mobile Computing*, Art. 8424381.

Eales, A., and R. Foss. 2012. "Service Discovery Using Open Sound Control." In *Proceedings of the 133rd AES Convention*, pp. 348–354.

Essl, G. 2011. "Automated Ad Hoc Networking for Mobile and Hybrid Music Performance." In *Proceedings of the International Computer Music Conference*, pp. 399–402.

Goltz, F. 2018. "Ableton Link: A Technology to Synchronize Music Software." In *Proceedings of the Linux Audio Conference*, pp. 39–42.

Guttman, E. 2001. "Autoconfiguration for IP Networking: Enabling Local Communication." *IEEE Internet Computing* 5(3):81–86. 10.1109/4236.935181

Henning, M. 2006. "The Rise and Fall of CORBA." *ACM Queue* 4(5): 29–34. 10.1145/1142031.1142044

Hintjens, P. 2013. *ZeroMQ: Messaging for Many Applications*. Sebastopol, California: O'Reilly Media.

Iglesia, D. 2016. "The Mobility is the Message: The Development and Uses of MobMuPlat." In *Proceedings of the International Pure Data Convention*, pp. 56–61.

Madgwick, S., et al. 2015. "Simple Synchronisation for Open Sound Control." In *Proceedings of the International Computer Music Conference*, pp. 218–225.

Malloch, J., S. Sinclair, and M. Wanderley. 2015. "Distributed Tools for Interactive Design of Heterogeneous Signal Networks." *Multimedia Tools and Applications* 15(74):5683–5707. 10.1007/s11042-014-1878-5

McKinney, C. 2016. *Collaboration and Embodiment in Networked Music Interfaces for Live Performance*, University of Sussex, PhD thesis. Available online at core.ac.uk/download/pdf/60240897.pdf. Last accessed November 2022.

Narveson, J., and D. Trueman. 2013. "LANdini: A Networking Utility for Wireless LAN-Based Laptop Ensembles." In *Proceedings of the Sound and Music Computing Conference 2013*, pp. 309–316.

Norilo, V., and R. B. Dannenberg. 2018. "KO2 Distributed Music Systems with O2 and Kronos." In *Proceedings of the 15th Sound and Music Computing Conference,* pp. 452–456.

Scarani, S., et al. 2019. "Software for Interactive and Collaborative Creation in the Classroom and Beyond: An Overview of the Soundcool Software." *Computer Music Journal* 43(4):12–24. 10.1162/comj_a_00534

Wilson, S., D. Cottle, and N. Collins. 2011. *The SuperCollider Book*. Cambridge, Massachusetts: MIT Press.

Wright, M., A. Freed, and A. Momeni. 2003. "Open Sound Control: State of the Art 2003." In *Proceedings of the International Conference on New Interfaces for Musical Expression*, pp. 153–159.