O2: Communication Middleware for Real-Time Distributed Music Applications

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

ABSTRACT

O2 is a communication protocol or "middleware" for real-time music applications. It features automatic discovery to simplify network configuration, clock synchronization, a reliable message option, and named services. A new version of O2 offering new capabilities is described. O2 now supports global discovery and communication, extending the previous version, which was limited to a single local area network. O2 can also deliver messages through shared memory allowing efficient lock-free communication with high-priority audio threads. Multiple styles of communication are facilitated in this new version, which supports *taps* to copy or "spy" on message streams. Taps can be used to implement publish/subscribe directly, and services also have writable properties that are eagerly pushed to peer processes for reading. Typical applications of O2 in creating computer music systems are described.

1. INTRODUCTION

Computer music systems are often "composed" from relatively isolated and independent modules for signal processing, graphical interfaces, control logic, audio effects, etc. In the past, performance requirements often dictated using a monolithic program where all processing could be carefully organized and scheduled. Today, most computers have multi-core processors, which allow multiple programs to run in parallel. This results in less contention for resources and therefore fewer scheduling problems.

Moreover, if tasks such as graphical interfaces, control and synthesis run in multiple programs, or at least in separate threads, more parallelism and therefore more computing power becomes available. Communication between programs, however, can be difficult. Software developers or end users must deal with a variety of communication protocols, each with different data representations, requirements for establishing connections, naming conventions and other details. Different programs support different protocols, including MIDI [1], Open Sound Control (OSC) [2], ReWire (en.wikipedia.org/wiki/ReWire_(software_protocol)), Abelton Link (www.ableton.com/en/link), and others, so additional software is often required to relay information between programs.

Experimental computer music systems often explore new ways to control and generate sound, so existing protocols based on traditional conventions of equal temperament, time signatures and tempo are often too rigid and limited. This might explain the popularity of OSC, which is a very simple and open-ended protocol for sending messages with an arbitrary set of parameters to objects or functions, organized in a hierarchical name space.

The O2 protocol aims to address some of the shortcomings of OSC, making it simpler to create powerful distributed music applications while keeping the open-ended extensible nature of OSC with which many musicians are already familiar. Important features of O2 include *services*, which help to achieve modularity, *discovery*, which simplifies establishing connections, *clock synchronization*, which creates a shared time base among all communicating processes, and *message delivery options*, that include (1) normal best-effort delivery as offered by UDP and (2) reliable delivery as offered by TCP [3].

These features formed the core of O2 version 1, which has been described previously [4]. This paper describes some of the new features in version 2 and their relevance to music systems. Related work is described in the next section, followed by an overview of O2. Section 4 presents new capabilities in version 2, and some implementation details are provided in Section 5. Section 6 describes some typical applications of O2, and Section 7 presents conclusions.

2. RELATED WORK

O2 is inspired by OSC, which is widely used in the community. O2 messages include the key ingredients of OSC messages: A URL-like address string and a set of typed parameters. Despite its success, OSC has a number of limitations, including the need to provide clients with the server IP address (which is often dynamic), and the problem of losing messages sent over Wi-Fi.

Libmapper [5] is a music communication system intended to map inputs from sensors into synthesis control parameters. This rather specific focus makes libmapper less general than other systems but more powerful in

Copyright: © 2022 Roger B. Dannenberg. This is an open-access article distributed under the terms of the [Creative Commons Attribution License 3.0 Unported](), *which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.*

tasks for which it is designed; for example, libmapper offers dynamically configurable connections from sensor parameter to control parameter with various adjustable mapping functions.

LANdini [6] has similar goals to O2 in that it offers discovery on a LAN and reliable transmission. However, LANdini messages flow in 3 "hops" – from application (1) to local server, (2) to remote server, (3) to remote application, and the overall message rate for N devices is proportional to N^2. However, LANdini has inspired some recent new capabilities in O2.

MobMuPlat (www.mobmuplat.com) [7] is software specifically for running Pure Data (Pd) [8] 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.

3. INTRODUCTION TO O2

O2 is described elsewhere [4], but we give a brief introduction here. The remainder of this paper will focus on new developments that extend the first version of O2. The basic function of O2 is to send one-way, peer-to-peer messages, usually to make a remote procedure call. Rather than directing messages to specific ports or host computers, messages are addressed to *services*. Any process can offer one or more services. This allows some flexibility in system design and configuration. Services can be moved from one host to another or from one process to another without changing any clients.

Messages contain a sequence of data items with type information (as in OSC). O2 messages also include a timestamp. Addresses are URL-like strings, similar to OSC addresses, e.g., `/synth/lfo/freq`; however, the first node in an O2 address (here, `synth`) is a service name, and the rest of the address refers to destinations (or operations) within that service.

O2 offers two ways to send a message:

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

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

where `o2_send` sends with a "best effort" or UDP policy, typically for time-sensitive sensor data that will be updated periodically, while `o2_send_cmd` uses a lossless or TCP policy, typically for critical one-time commands.

O2 automatically performs clock synchronization so that messages can be delivered at precise times in the future. This is a technique for avoiding network jitter [9].

O2 also offers OSC compatibility. To act as an OSC client, an application can *delegate* an O2 service to an OSC server. Messages for that service will then be forwarded to the OSC server. O2 can also act as an OSC server that converts incoming OSC messages to O2 messages and forwards them to an O2 service.

4. NEW CAPABILITIES

4.1. Global Discovery and Communication

The pandemic has created new awareness and activity in music networking. O2 version 2 extends the discovery protocol beyond the local area network to the global Internet. This wide-area discovery protocol uses MQTT (mqtt.org), a lightweight Internet-of-Things (IoT) messaging protocol for which there are open servers.

Using MQTT, O2 processes subscribe to a topic that informs them of other O2 processes. To support multiple O2 users on the Internet or even on a single LAN, each group of communicating O2 processes, called an *ensemble*, is given a unique name.

Suppose our ensemble name is `myens`. To become reachable, each process subscribes to the MQTT topic `o2-myens/xxxxxxxx:yyyyyyyy:zzzz`, where `x…:y…:z…` encodes the public IP address (`x…x`), local IP address (`y…y`) and port number (`zzzz`) of the process, all in hexadecimal.

Each process also subscribes to `O2-myens/disc` and sends its service name (every process has an automatically generated O2 service name) to that topic. This notifies every existing process that the new process exists and how to reach it. By examining local and public IP addresses, O2 processes can determine if they have direct connectivity to each other. If so, pairs establish a direct peer-to-peer connection. Processes behind NAT firewalls continue to use MQTT to reach other networks[1]. Figure 1 illustrates connectivity between O2 processes.

To send a message, O2 first looks in a dictionary for who offers the indicated service. If the service is offered by a remote process, O2 checks for an existing connection (either a TCP socket or a UDP address). If found, the message is sent directly. If not, O2 (version 2) publishes the message to the MQTT topic corresponding to the destination process. Network routing is based only on the

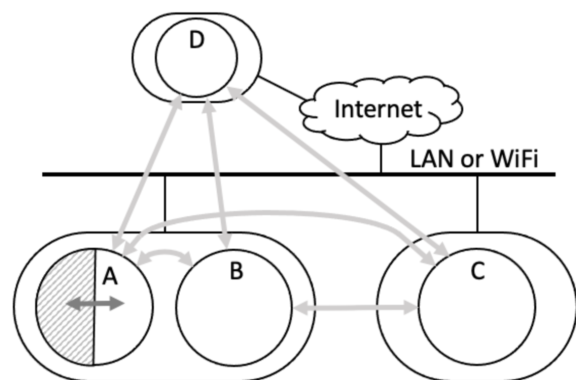

Figure 1. Connectivity possibilities in O2 include processes on the same host (A to B), processes on the same LAN (A or B to C), and processes on different LANs (any to D). Process A uses the shared memory option to exchange messages between two threads.

[1] A future extension might use STUN (en.wikipedia.org/wiki/STUN) to traverse NAT gateways and offer lower latency in some cases.

service name, but upon receipt by the process offering the service, the remainder of the address is used to find a specific handler (a function pointer).

4.2. Shared Memory Communication

One drawback of O2 version 1 for audio applications was that all communication required networking system calls. Typically, network communication is not recommended (and may be prohibited) in audio processing threads. To address this, version 2 has a shared memory interface for very low-latency messaging between threads. This is particularly intended to support sound synthesis. O2 can now provide communication where one would typically require special lock-free mechanisms to communicate with asynchronous audio computation.

The life cycle of a message in this case is as follows: (1) In the message construction process, message memory is allocated from a built-in, real-time allocator that dequeues memory from a free list[2]. If the free list is empty, memory is allocated from a large pre-allocated memory region. (2) The message is delivered by pushing it onto a linked list that is read by the other thread. (3) The receiving thread periodically removes the entire list with an atomic lock-free operation. It then reverses the list to put the messages in time order and delivers each message. (4) After delivery, the message is pushed atomically onto the same free list from which it was allocated. These free lists are shared by all threads.

Threads using shared memory communication run a subset of O2 called O2lite and cannot send network messages. Instead, they communicate only with a full O2 thread acting as *proxy* to forward messages to and from other hosts. To a remote process, all the services appear to belong to the proxy. When an incoming message is received by the proxy, the service name is checked to determine if it should be handled immediately or be forwarded to a shared memory thread. Similarly, outgoing messages from a shared memory thread are posted to the proxy, which forwards them to the real destination.

4.3. Publish/Subscribe

A normal style of message passing in O2 (or OSC) assumes the sender is requesting a remote action. This is essentially the remote procedure call (RPC) approach. Another style is based on the idea that the sender wants to make information available, such as the current value of a sensor, but does not know what processes are interested in that information. This style can be implemented by allowing interested parties to subscribe to some message source. The source must remove subscribers (other services) when they fail or terminate.

To implement this style of communication, O2 has a mechanism called a *tap*. Originally, taps were created to support debugging in a distributed system. When a service *taps* another service, it receives copies of all messages arriving at that service. Taps can be used directly to implement publish/subscribe. The publisher creates a local service, possibly without any message handler. To publish, it sends a message to this service. Subscribers create a local service that *taps* the publisher's service.

O2 maintains taps, removing them when the receiver fails (detected by monitoring the receiver's TCP socket). To avoid the infinite circulation of messages in the event of a circular tap structure, tapped message have a "hop count" that prevents the message from being forwarded more than twice.

4.4. Properties

Yet another style of communication is based on sharing state rather than sending messages. Imagine a service that can be "ready" or "paused." Clients of the service might want to know when the service is "ready." Of course, the service could actively send its state to interested parties through publish/subscribe, but new subscribers might miss the message. At best, all subscribers have to receive and remember state changes.

LANdini has a built-in mechanism for assigning properties to hosts. For example, a laptop orchestra might have a conductor, a number of players, and some non-player processes. The conductor might want to identify who are the players. This is simple if players can have a "player" property. In O2, players would be services, each with a unique name, and originally there was no easy way to find services that are players (except possibly through naming conventions: player1, player2, etc.

Inspired by LANdini, O2 now allows applications to associate a list of property/value pairs (both are strings) with each service. O2 functions allow a process to look up property values or to search for all services with a certain property value, such as "`player:true`."

Properties are implemented as part of the O2 discovery system. Whenever the status of a service changes (it is created or destroyed), update messages are automatically sent to all known O2 processes. Properties are simply an extension of the service status. Whenever anything changes, the complete service status is transmitted to all processes. Frequent changes or large amounts of state can result in a large amount of network traffic, but small amounts of fairly stable state information can be conveniently provided with this mechanism.

5. IMPLEMENTATION

O2 is written in C++, runs on macOS, Linux and Windows, and is freely available as open-source software (github.com/rbdannenberg/o2). It uses Bonjour (known as Avahi on Linux) for **discovery** of processes. As each process is "discovered" by another, a TCP connection is made, and additional information is then exchanged via special O2 messages about services. Every process keeps

---

[2] One free list for each size, with choices in increments of 8 up to 512 bytes, then sizes double up to 16MB.

a complete and up-to-date map from service name to service provider (a process) so that messages can be routed quickly and directly to the intended receiving process.

Clock synchronization is built into O2 and requires one process to be manually designated as the reference clock. Other processes discover the clock service and periodically request the time. Local clocks are updated after filtering out delayed responses and compensating for round-trip network transmission time. Clocks are adjusted smoothly to avoid jumps in the estimates of the global clock time. Since O2 messages are accurately timed, a process can schedule internal events simply by sending messages to itself. By layering clock synchronization over O2 discovery and messaging, and by integrating O2 clocks with timestamps and message scheduling, the implementation is compact, self-contained, and allows an external time reference such as digital audio, in contrast to using a protocol such as NTTP.

Message delivery begins by looking for the first node of the address in a hash table of services. (See Figure 2.) Each service is associated with two lists: (1) *service providers* lists remote processes or local handlers (functions) that implement the service, sorted by their process IP addresses. The highest is designated as the "active" service. (2) *taps* list services that have tapped this service. The message is sent to the active service (if any) and all taps on the service. Normally, the *service provider* directly references an object maintaining a TCP socket, a buffer for incoming messages, and the connection state. Alternatively, the service could be local and described by a tree of hash tables used to decode the address[3].

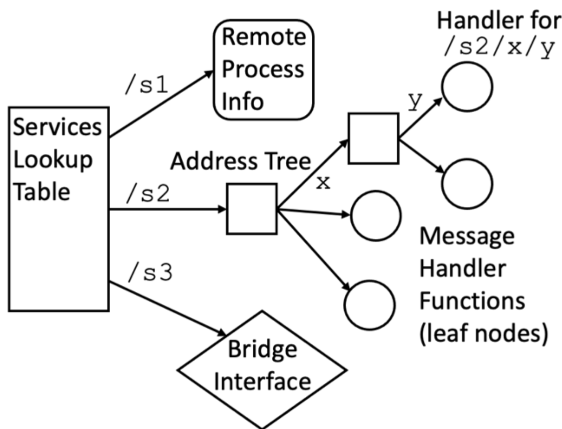

Figure 1. To deliver messages, each process first maps the service name (first node of the address) to either: (top) a remote-process object encapsulating an open TCP socket, UDP address, and connection state; (middle) local service represented as a tree to decode the address, e.g., `/s2/x/y`, mapping it to a handler function pointer or object; (bottom) a Bridge Interface instance, which implements shared memory, MQTT and other transports. This diagram is slightly simplified, as each service actually maps to a list of service providers and taps rather than a single provider.

Yet another option is that the service can be provided by a *bridge*. The bridge abstraction allows O2 to be extended to other transports and protocols. The use of MQTT for global discovery and connectivity is one example, and the shared memory interface is another. The bridge abstraction has also been used to create a minimal protocol, O2lite, which gives O2 messaging capabilities to microcontrollers (over WiFi) and web browsers (over WebSockets) [10]. In fact, O2 includes an integral web browser to serve web applications and an O2lite implementation in Javascript making it easy to connect browsers to O2 ensembles.

5.1. Performance and Evaluation

Most of the runtime in message sending is due to basic network operations and not the additional functions provided by O2. In a simple test, sending short messages round-robin between two processes on the same machine, O2 sends about 35,000 messages per second using either TCP or UDP, compared to about 45,000 per second with a streamlined implementation using UDP directly. When actual networks are involved, costs are higher, and the relative cost of O2 protocols is negligible.

For shared memory operation, we achieve about 600,000 short messages per second, or a 3.3 μs round trip to send a message to another thread and get a reply. Most of the time, however, is due to polling for network activity. If multiple messages are sent within one polling period, the polling cost will be amortized over many messages. Detailed analysis, factoring out the cost of checking for network events, indicates that a one-way shared memory send and receive adds only 0.32 μs of processing time, which is equivalent to 3.1 million messages per second. All tests were performed on a 2.4GHz Intel Core i5 running macOS.

O2 version 2 uses non-blocking calls for send and receive to minimize its timing impact, but only one message is queued to prevent clients from queuing an unbounded number of messages waiting on a too-slow connection. A process that is about to block in a send can detect that condition and avoid blocking altogether.

Of course, any system or protocol is judged not only by performance but by functionality and ease-of-use, which is very hard to measure. We *believe* O2 is a strong candidate for computer music applications because it has evolved from experience with OSC, ZeroMQ [11] and others. We have used it for laptop orchestras, audio servers, interactive music systems and web interfaces, and we hope the community will use it more widely.

6. APPLICATIONS

O2 is intended for a variety of applications. Interconnection and communication become critical as music systems

[3] O2 supports OSC-like address patterns, and the tree structure allows efficient search for matching addresses. However, since OSC patterns are not widely used, a simpler and more efficient "flat" hash table of full address strings is also maintained to allow a full address lookup in a single step. A compilation option allows the developer to disable pattern decoding, which also avoids maintaining the tree structured pattern lookup structure.

become more complex, combining graphical interfaces, sophisticated sound synthesis systems, machine learning modules, wireless sensors, video projection, and even global participation. We believe O2 can be helpful in many applications such as the following.

6.1. Laptop Orchestras

The first application of O2 was in a laptop orchestra performance. Discovery simplified the process of connecting players to a central conductor service, eliminating the need to type in IP addresses and port numbers. The conductor established a shared musical time framework by publishing a mapping from O2 time to beat, e.g., $beat = (real_time - time_offset) \times beats_per_second$. Since clocks were synchronized, music from the ensemble was synchronized so accurately that we had to consider the speed of sound in our setup. Control combined global information from the conductor with local control by each player. Some players used TouchOSC (hexler.net/touchosc) and other controllers connected to their laptops using the OSC compatibility feature of O2. (video: youtu.be/3OYhC3KNt-g)

6.2. Sensors and Synthesis

O2 was designed to be light-weight and to run directly on microcontrollers with Linux, such as the Raspberry Pi (raspberrypi.org). However, microcontrollers abound, and version 2 of O2 includes an O2lite implementation for Arduino-compatible microcontrollers such as the ESP32, which is a low-cost, low-power microcontroller with support for Wi-Fi. With these types of microcontrollers and O2, sensors can be integrated into computer music systems with a minimum of on-stage configuration. Reliable 2-way communication allows sensors to be configured remotely, e.g., to request specific sensor data or change the sample rate without reprogramming. Code samples are included in O2 source code.

6.3. Network Music

Many computer music performers collaborate over networks [12]. Although O2 is not designed to support audio and video streaming, network music often involves collaborative control of music generation systems, graphical displays and sharing gestural controller data. O2 makes it easy to bridge multiple sites, especially computers on home Wi-Fi networks that do not have public IP addresses. Even applications running OSC can be interconnected as follows (see Figure 3): Create and run a local O2 application that acts as an OSC server on (arbitrarily) port 8000, forwarding messages to service `remote`. Create and run a remote O2 application that *delegates* service `remote` to an OSC server at (also arbitrarily) port 8000. Now a local OSC application can send to "localhost" (128.0.0.1), port 8000. The messages will be transferred to the remote machine (anywhere with Internet access) and delivered to the remote OSC server on port 8000. This approach parallels our "Telematic

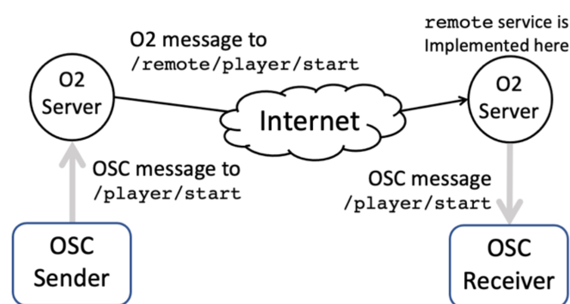

Figure 3. Bridging OSC over the Internet. An OSC Sender sends to a local O2 Server (left), which discovers the `remote` service and forwards OSC messages reliably. The `remote` service converts messages back to OSC and sends them to a local OSC receiver. Thus, OSC is delivered reliably across the Internet 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.

Soundcool" performance [13], which was implemented directly with TCP before O2 was available.

6.4. Interactive Installations

Installations often use multiple computers for audio, video and control processing. Especially when microcontrollers and existing Wi-Fi networks are used, connections can require manual configuration or even reprogramming microcontrollers with fixed IP addresses. With the ability to automatically connect multiple laptops and microcontrollers, O2 simplifies configuration and offers open-ended message formats and addressing to support the needs of the installation. Another interesting option is that artists often want to monitor long-term installations remotely to make sure they are functioning correctly. Using global-scale discovery, publish/subscribe, and other capabilities, the artist can construct remote monitoring software that taps into the installation to obtain status, run tests, or monitor activity. Although this idea has not been used in a real installation, the capabilities are illustrated in a working prototype described elsewhere (see Figure 4 in [4]).

6.5. Audio Synthesis

Audio synthesis typically runs on high-priority threads where locks or blocking calls, which could add unbounded delay, are prohibited. Without locks, software must resort to lock-free data structures so that control software can communicate with audio synthesis software. O2 now offers lock-free communication for time-critical audio processing. This is not unlike the use of OSC by the Supercollider synthesis server [14], only with more flexible addressing, discovery, clock synchronization, etc. See github.com/rbdannenberg/arco for a working example.

6.6. Modular Performance Systems

O2 originated in the context of a modular performance system [15], with modules that include a *conductor* for

coordinating performers, *sequence-players* for performing MIDI sequences, *audio-players* using phase-vocoders to synchronize audio playback to live performers, and *score-players* to display music notation and turn pages automatically. O2 is now integrated with our code, so modules discover other modules and configure themselves, e.g., *players* can operate stand-alone, but when a *conductor* is discovered, the players accept control (start, stop, set position, tempo) from the conductor. Clock synchronization allows modules to run as separate processes but still synchronize precisely, even in the face of network latency.

This approach offers great flexibility. For example, we have a polyphonic score-follower that, when substituted for the usual *conductor,* converts the system into a (not yet released) computer accompaniment system [16]. We believe that future live computer music systems will include intelligent agents as performers, and we hope that configuring these future systems will be as commonplace as today's rock band connecting electric guitars, pedals, mixers, microphones, amplifiers and speakers.

7. CONCLUSIONS

O2 is a powerful communication protocol or "middleware" that aids in the construction of interactive real-time computer music systems. This article introduces new developments now available in version 2, including discovery and communication on a global scale using the same structure of services and messages. O2 does not rely on DNS [3], so it works without domain names and fixed IP addresses, which are often impractical in-home networks, mobile devices and concert venues. At the very local level, messages can now be delivered efficiently between high-priority threads sharing memory using lock-free data structures to support low-latency audio processing.

Also new in O2 are some styles of communication. State information can be shared through properties associated with services. Frequently changing information can be monitored using a publish/subscribe protocol.

We have also described six common patterns of communication in music applications ranging from low-latency audio computation and control to network music on a global scale. A feature of O2 is that a single message format and addressing mechanism can be used to send messages to a thread in the same process or to another host anywhere on the Internet. In fact, the location of services (receivers) is abstracted from the address, so systems become more modular and reconfigurable.

In the future, we expect to make more libraries available implementing O2lite natively so that code written in Python, Java, Javascript and other languages can send messages into an O2-based application. We also expect to create external O2 objects for Max (cycling74.com) and Pd (puredata.info) as an alternative to OSC.

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.

8. REFERENCES

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

[2] M. Wright, A. Freed, A. Momeni. "Open Sound Control: State of the Art 2003." *Proceedings of the International Conference on New Interfaces for Musical Expressio, 2003,* pp. 153-159.

[3] P. L. Dordal, *An Introduction to Computer Networks,* 2016. (available at: www.freetechbooks.com)

[4] R. B. Dannenberg, "O2: A Network Protocol for Music Systems," *Wireless Communications and Mobile Computing*, (2019), Article ID 8424381, 2019.

[5] J. Malloch, S. Sinclair, M. M. Wanderley, "Distributed Tools for Interactive Design of Heterogeneous Signal Networks," *Multimedia Tools and Applications*, *15*(74), 2015, 5683–5707.

[6] J. Narveson, D. Trueman, "LANdini: A Networking Utility for Wireless LAN-based Laptop Ensembles," *Proceedings of the Sound and Music Computing Conference 2013 (SMC 2013)*, 2013, pp. 309–316.

[7] Iglesia, D. "The Mobility is the Message: the Development and Uses of MobMuPlat." *PdCon16~*, 2016.

[8] Brinkmann, P., et al. "Embedding Pure Data with libpd." *Proceedings of the Pure Data Convention*, 2011.

[9] E. Brandt, R. B. Dannenberg, "Time in Distributed Real-Time Systems," *Proceedings of the 1999 International Computer Music Conference*, San Francisco: International Computer Music Association, 1999, pp. 523-526.

[10] Dannenberg, R. B., "Building Interactive Music Systems with O2, Microcontrollers and Web Browsers," *Proceedings of the International Conference on New Interfaces for Musical Expression*, 2022.

[11] P. Hintjens, *ZeroMQ,* O'Reilly Media, Inc., 2013.

[12] C. McKinney*, Collaboration and Embodiment in Networked Music Interfaces for Live Performance*, University of Sussex, Ph.D. Thesis, 2016. (available as: core.ac.uk/download/pdf/60240897.pdf)

[13] S. Scarani, A. Munoz, J. Serquera, J. Sastre and R. B. Dannenberg, "Software for Interactive and Collaborative Creation in the Classroom and Beyond: An Overview of the Soundcool Software," *Computer Music Journal*, 43(4) (Winter), pp. 12-24.

[14] S. Wilson, D. Cottle, N. Collins, *The SuperCollider Book*. The MIT Press, 2011.

[15] R. B. Dannenberg, N. E. Gold, D. Liang, G. Xia, "Methods and Prospects for Human-Computer Music Performance of Popular Music," *Computer Music Journal*, 38(2) (Summer 2014), pp. 36-50.

[16] R. Dannenberg and C. Raphael, "Music Score Alignment and Computer Accompaniment," *Communications of the ACM,* 49(8), 2006, pp. 38-43.