OSCTHULHU: APPLYING VIDEO GAME STATE-BASED SYNCHRONIZATION TO NETWORK COMPUTER MUSIC

Curtis McKinney
Bournemouth University
Creative Technology Research Group
cmckinney@bournemouth.ac.uk

Chad McKinney
University of Sussex
Department of Informatics
C.Mckinney@sussex.ac.uk

ABSTRACT

In this paper we present a new control-data synchronization system for real-time network music performance named OSCthulhu. This system is inspired by the networking mechanics found in multiplayer video games which represent data as a state that may be synchronized across several clients using a hub-based server. This paper demonstrates how previous musical networking systems predicated upon UDP transmission are unreliable on the open internet. Although UDP is preferable to TCP for transmitting musical gestures, we will show that it is not sufficient for transmitting control data reliably across consumer grade networks.

This paper also exhibits that state-synchronization techniques developed for multiplayer video games are aptly suited for network music environments. To illustrate this, a test was conducted that establishes the difference in divergence between two nodes using OscGroups, a popular networking application, versus two nodes using OSCthulhu over a three minute time-span. The test results conclude that OSCthulhu is 31% less divergent than OscGroups, with an average of 2% divergence. This paper concludes with a review of future work to be conducted.

1. INTRODUCTION

Computer network music has benefitted from three decades of development, including the experiments of the San Francisco Bay Area network band pioneers, the introduction of the OSC protocol [22], and research into streaming and latency issues. Making an infrastructure suitable for network performance in the face of highly distributed participants and online security roadblocks remains a challenging task, and one which this paper confronts. Our solution, OSCthulu, is a client-server architecture which has proven robust in concert performance, and as open-source software may be of benefit to other researchers and performers. This system has been researched and developed by the authors with real-world testing conducted by their network music band Glitch Lich [9].

As noted in *Indigenous to the Net*, the world's first network computer band, the League of Automatic Music Composers, began as an extension of the home brew circuit tinkering that was characteristic of the Bay Area in the mid-1970's. Their computers, MOS Technology KIM-1 models, were modest with only 1 kilobyte of memory and could only be programmed using assembly language. The League of Automatic Music Composers created interactive programs by directly soldering connections between computers and writing programs which would listen and transmit data on these lines. The network was fragile and error prone. It was also difficult to set up as all the connections had to be re-soldered each time the band rehearsed [3].

In what can be seen as a natural evolution of the technology, the spiritual successor to the League of Automatic Music Composers, the Hub, utilized a server-based system. This system provided a standardized interface for connections between members with varying computer models, as well as shared memory for the ensemble. Throughout this time many other approaches to networking were being developed. Previous efforts mentioned focused on the real-time interactions of the performers' computers, but in the 1990's several new methods explored non-real-time connections. Systems such as the ResRocket Surfer and Faust Music Online (FMOL) allowed users to collaborate writing music by providing an online repository [12] [11].

Much research has been done towards investigating the issues that latency presents to instrumentalists when streaming audio as well as strategies to cope with this latency in performance [1]. Often solutions favor research grade connections between sites, providing lower latency, although these types of connections are not widely available to the public [20] [14]. Several alternative approaches have been taken to address network latency such as making the latency a multiple of a preset tempo, as with NINJAM [4]. Local delay offsets are used in the eJAMMING software to produce the same latency of actual audio output between all users, attempting to bring the synchronicity of the users closer [7].

Recent developments in music programming languages such as SuperCollider and ChucK have led to new and exciting systems focused on code sharing in live coding ensembles. Systems such as Co-Audicle and Republic create networks in which performers can share code that will be altered, executed and re-entered into the pool [19] [18]. Live coding is a fundamentally computer-based performance style and for that reason lends itself well to networking. The information being transferred is small, yet

can produce long-lasting results and the time specific parameters for execution are much less restrictive than in a traditional performance. The practice leads to organic interdependencies and produces layers of uncertainty which afford unexplored modes of performance, composition, and listening [6] .

Currently groups like the Hub, PLOrk, Powerbooks Unplugged and many others favor the Open Sound Control (OSC) protocol for networking data between nodes. OSC is a flexible protocol that is geared towards communication between musical systems. OSC provides a dynamic and powerful framework with high resolution data which is usable on local networks and across the internet [22]. In surveying the current solutions we feel that there is room for an alternative solution for networking compositions and performances, which implements techniques established by multiplayer video games.

2. OSCTHULHU 1.0

Work on a new OSC-based platform for networking collaborative electronic music began in June 2010. The project, named OSCthulhu, was inspired by the program OSCgroups created by Ross Bencina, which enables users to share OSC messages with each other over a network [2]. OSCgroups accomplishes this by creating a central rendezvous server that uses Network Address Translation (NAT) hole-punching techniques to enable individual users to bypass firewall and router restrictions normally placed on peer-to-peer communications.

2.1. Nat Hole-Punching and UDP multicasting

Normally, a router will block any message that is received unless a previous message has been sent out by the user to that specific IP address and port. This is done to prevent nefarious traffic from reaching the user's private network and computer. Furthermore, the IP address and port of an application behind a user's router is obfuscated by Network Address Translation (NAT), a system utilized by routers to preserve IP address real estate on the open internet (largely addressed by the upgrade from Ipv4 to Ipv6) [10]. The server works around this by noting the private and public IP Address and port pairs, known as an endpoint, of each user that logs into a group. The server distributes this information to everyone within the group, at which point all of the users then asynchronously send messages to all of the public and private endpoints that it has received from the server. The first messages received at either end will be discarded by their respective routers as they have not been met with a matching outgoing message. However, now the user has punched a hole in their firewall by sending an outgoing message to each of the recorded endpoints. Now when the user receives messages at their endpoint, be it from any of the public or private IP pairs that they have received, it will successfully be accepted as valid traffic by the router, allowing for full bi-directional peer-to-peer communication between the users [8].

Once external communication has been established, the client application on each user's computer opens up an internal UDP port that parses any incoming OSC messages it receives and multicasts that message to everyone in that user's group. Due to the flexible nature of OSC the origin of these messages could be from any application that has OSC capabilities, including programs such as Max/MSP, SuperCollider, or Reaktor [5] [21] [13].

The benefits of this approach are that users can easily and dynamically form groups to share messages between while being in completely different places in the world, without having to note the individual public and private IP endpoints of each person within the group. The strain on the server is also minimal, as it only serves as a rendezvous point for users, and none of the actual OSC messages are passed to the server. The system's multicasting architecture is quite appropriate for network music systems that require musically significant gestures to be shared over a network with the utmost speed and low overhead granted by UDP messaging systems, but do not require the reliability of a slower Transmission Control Protocol (TCP) based system [16].

2.2. Reality of the Internet: Packet Loss

After extensive usage of this system within the context of the authors' network-based computer music band Glitch Lich, several shortfalls of this system became apparent. While OSCgroups is straightforward and robust, the usage of UDP meant that systems that relied upon extensive synchronization between peers would often suffer from potentially fatal errors during performance due to packet loss. Packet loss (when a packet is sent at one endpoint, but not received at the other) is an unfortunate reality of networking that every networked program must address in some way.

Many programs overcome this by utilizing TCP for reliability, which has built-in systems to handle packet loss, retransmitting lost packets after a certain timeout period [15]. However, for systems that require the utmost speed, such as gaming and musical applications, TCP is deemed inappropriate due to its sluggishness. Furthermore, the nature of TCP's retransmission mechanism means that critical real-time gestures in a game or piece of music may be transmitted out of order, negatively impacting the quality of play. Thus, TCP is too slow and unwieldy, and UDP is too unreliable for pieces which rely upon stringent accuracy.

2.3. Looking to Multiplayer Videogames

The similarities between multiplayer gaming and network-based music are rather striking. Both require raw speed to ensure that multiple peers can react to each other's actions as realistically as possible. In both, raw speed is considered more important than absolutely receiving every packet sent. In both, out of order information and information based upon an old state of the system should be avoided if possible. However, both can be fatally affected

by the loss of certain important packets of information. It seems only natural then for the network musician to look to video game networking techniques to learn how they deal with such an important issue.

Tim Sweeny, a game programmer and creator of the Unreal Engine, wrote an in-depth analysis of the history, difficulties, and techniques involved in programming multiplayer games titled "Unreal Networking Architecture". Written in 1999 at the veritable dawn of modern multiplayer First-Person Shooter (FPS) games, the document is rather striking in its presentation of a problem that sounds remarkably similar to the plight of the network musician. Sweeny eloquently states that "Multiplayer gaming is about shared reality: that all of the players feel they are in the same world, seeing from differing viewpoints the same events transpiring within that world [17]."One may easily replace multiplayer gaming with network music in that sentence to describe the promise of network-based collaborative electronic music.

Sweeny describes a system that overcomes the shortfalls of packet loss in UDP by utilizing what he describes as a "Generalised Client-Server Model "(GCSM). In the GCSM whenever a client makes an action the client simultaneously updates its own internal game state (the exact state in which all objects in the world are in at any given time), as well as sends its action as a message to the server. The server then updates its own internal game state to reflect these actions. After a period of time of receiving action messages from clients, referred to as the DeltaTime, the server will update its game state, using predictive analysis to correctly account for lag time in message transmission. The server then issues an update to all of the clients, called a Tick. This game state may differ from that of any of the clients' due to several issues, including packet loss, and the inherent asynchronicity of client actions due to lag. However, as Sweeny put it "The Server is *The Man*"and the server's game state takes precedence over that of the client. Thus, when a client receives an update after DeltaTime, its internal game state is replaced with that of the servers. This solves both of the previous problems stated in the previous section: The client's actions are perceived to be immediate (as it updates its own internal game state immediately), it receives peers' actions in a swift manner due to the usage of UDP, but if a packet is lost in transmission, a system is in place to handle it in a sensible and reliable way.

To the user the only perceived anomalies are when there is a discrepancy in the server update, usually due to packet loss or lag. For example, he may have perceived himself as shooting another player in the head, but the server states that the player is still alive. While this can be vexing, it is preferable to the alternative: the client kills the other player on their computer, but in the other player's game-state they are still alive, effectively creating simultaneous alternate realities and immediately ruining any notion of a shared experience. On the other hand, there is always the reverse scenario in which the client believes they missed, when in fact, the server states they hit their target.

2.4. A Musical Approach to a Gaming Model

OSCthulhu was created as a musical analogue to the GCSM approach. After testing several implementations of the GCSM as described by Sweeny, some tweaking was required to produce a model that was appropriate for usage in the context of a network music environment. The core of OSCthulhu is the way it represents data, which is very similar in approach as the GCSM. Data is represented in the system as a series of networked entities called *SyncObjects*. These SyncObjects contain an arbitrary amount of modifiable values, called *SyncArguments*. SyncArguments may be Strings, Integers, Floats, or Doubles. While in the original GCSM SyncArguments were accompanied by a fixed name, in OSCthulhu they are referred to by index. This change was made to preserve bandwidth.

Another change that was made was the behavior of client actions and ticks. In OSCthulhu, when a client action is received it is immediately multicast to all of the clients instead of the server waiting for DeltaTime and issuing a Tick to update the clients. This was done to make the system as fast as possible, though at the expense of more bandwidth. This is considered acceptable for musical purposes, as the average network music server will deal with significantly less traffic than a gaming server, and thus can afford to be faster at the expense of being less efficient. This also means that there are two ways that a client may be updated in OSCthulhu, either by a *setSyncArg* message, which updates a single SyncArgument, or by a *serverSync* message which wholly replaces the client's state with the servers.

2.5. A difference in styles

One key point to keep in mind when using OSCthulhu is that it is fundamentally a different way of organizing the manner in which a networked composition or software system is constructed. Oftentimes we as composers think of networking as a series of commands: change sections, get louder, stop playing, switch timbres, etc. To network with OSCthulhu, a composer must think of his or her composition instead in terms of a series of objects. These objects may be manipulated in similar fashion to the components of an object-oriented programming language. Objects may be created, destroyed, or have their values altered. So instead of our previous example where we gave commands to modify music, instead a composer would have an object that represents a synthesis unit generator, including variables that represent that unit generator's amplitude and timbre. To create another instance of that unit generator, perhaps with a different set of arguments, one would simply add another instance of that object to OSCthulhu. This approach may require a bit more forethought, but the structure lends itself well to networking musical contexts, especially in remotely rendered synthesis configurations.

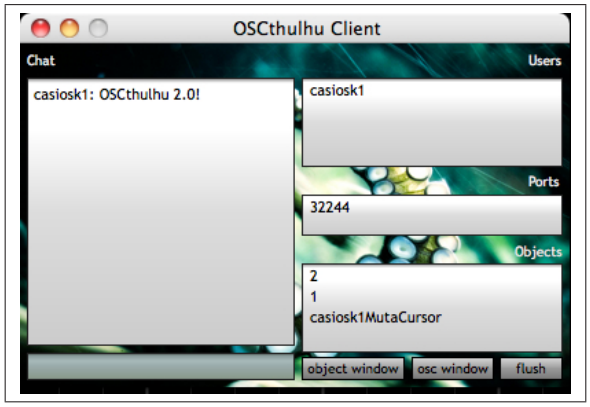

Figure 1. Screen capture of the SyncObject and Chat window in *OSCthulhu 2.0*

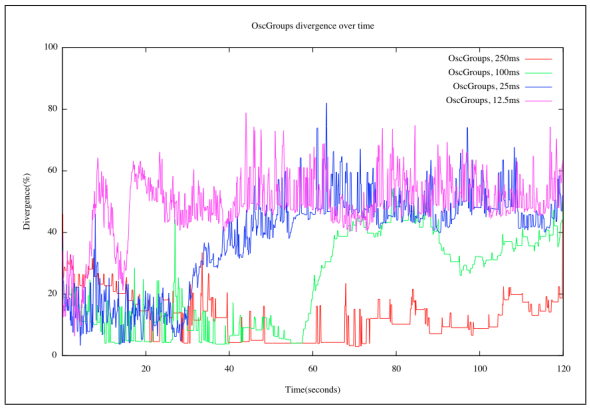

Figure 2. OscGroups divergence over time.

2.6. OSCthulhu and OSCgroups: A comparison

OSCthulhu has not been designed to supersede OSCgroups. Instead, it is meant to be an alternative approach, useful for a set of situations that OSCgroups may be deemed to be less suitable for. The advantages of OSCgroups over OSCthulhu are a simpler interface with less overhead that doesn't restrict the structure in which it is used. Also, due to OSCthulhu using a GCSM a server is required, with all pertinent traffic being directed through that server. Although a central server is required for OSCgroups as well, this server can handle multiple groups simultaneously, and none of the multicasted traffic is forwarded through the server.

However, OSCthulhu is appropriate for projects that require both a close degree of synchronization as well as the speed and simplicity of UDP multicasting. This allows for the construction of new kinds of network-based electronic music systems or pieces. These systems can rely upon shared resources to coordinate a network music performance, with the knowledge that this information will be transmitted in the most musically sensitive way, while always being safely accounted for.

3. DIVERGENCE TEST

3.1. Methodology

A test was conducted to demonstrate this effect. This test consisted of two nodes, one in London, England and the other in Boulder, Colorado, both using standard consumer level broadband networks, sending messages to each other. Standard broadband was chosen for this experiment as OSCthulhu has been designed specifically to facilitate network music performance in real world environments outside the confines of academic institutions with access to research networks. The results gathered in this experiment may differ on these academic research networks and future experiments are planned to investigate the differences this makes.

These two nodes created and altered various data sets on their own systems, while simultaneously sending messages to each other to coordinate those same changes on the other node. There were three different actions a node could make: create an array (with a random number of indices, each containing a random value), alter an index of an array, or delete an array. These actions were chosen randomly, with index alterations occurring twenty times more often than creating or removing an array, to reflect real world scenarios. The test was conducted with four different send rates at which changes would occur and messages would be sent: every 250, 100, 25, and 12.5 milliseconds. These messages were sent over a period of two minutes, using either OscGroups or OSCthulhu on subsequent run-throughs for comparison.

A value called *divergence* was collected every 10 milliseconds for each run-through. This test defined divergence as a measurement of the difference between the two nodes' states at any given moment in time. For example, if at a given moment node one contained four arrays, and node two contained five arrays, but four of those arrays were identical to those contained in node 1, then the systems would be considered 20% divergent. Figure 2 shows the results produced by OscGroups.

3.2. Results

The results show a staggering amount of divergence, with the systems immediately beginning at approximately 20% divergence, and becoming more divergent over time, settling at approximately 50%. This divergence can be accounted for by packet loss, lag time, and the cascading nature of divergence (i.e. if an array is missing on one node, the other node is not aware of this and will continue to attempt to set values in it. They will not realign until the second node serendipitously removes the array). Glitch Lich has personally encountered this divergence in performance, wherein a member at one node is creating sounds with a certain unit generator they have created, but the other nodes do not contain this unit generator, therefore the first node's performance is effectively non-existent.

Figure 3 shows the results for OSCthulhu. The results show a stark difference as the amount of divergence is pre-

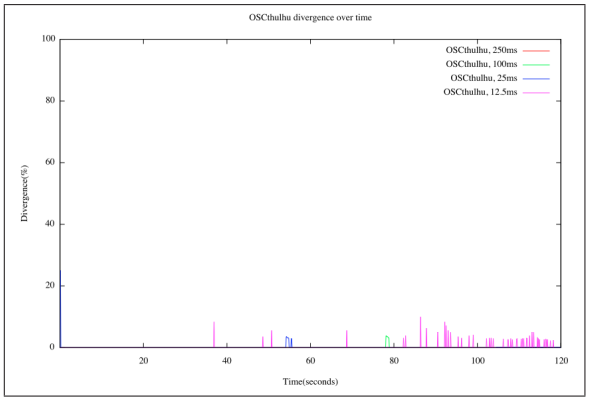

Figure 3. OSCthulhu divergence over time.

Figure 4. Yig in performance at the Network Music Festival in Birmingham, UK, powered by OSCthulhu.

dominantly zero, with spikes up to 5-10%. There are two main reasons for this large difference in divergence between the two systems. Firstly, the effects of packet loss are drastically minimized, as the GCSM server synchronization cycle ensures that every cycle period (1000 milliseconds used for this test) the two nodes locked back in step (unless the synchronization packet itself is lost, which does happen on occasion). This prevents the cascading effects of divergence from taking hold, so differences do not pile upon each other over time. Secondly, differences due to lag time are also minimized, as all the actions are first sent to the central server which then simultaneously broadcasts the effects to both nodes. These nodes then receive the message and act upon it in a very similar time scale.

3.3. Benefits of Convergence

One manner in which networked ensembles may take advantage of this capability is through the usage of what may be called *Remotely Rendered Synthesis*. In many network music bands, including most of the work conducted by The Hub and PLOrk, network messages are transmitted among multiple participants to influence each other's behavior. Each member then uses their own computer to output their own sounds. In comparison, in a Remotely Rendered Synthesis configuration, each of the participants share their sound synthesis descriptors with each other member beforehand(in the case of Glitch Lich, SuperCollider *SynthDefs* are used); then, a *Sound State* is constructed that is mirrored on each participant's own computer. This Sound State is similar to a Game State in Sweeny's GCSM, except that the data being synchronized represents the state of a sonic world instead of a virtual game world. OSCthulhu keeps track of the Sound State present on each user's computer. Whenever a member makes a change to their particular version of the Sound State, this change is replicated on the server, and shared with the whole group. Then, each member's computer outputs audio that contains the full sound present in the piece, including audio that is being produced by other members. This is useful for network music performances wherein all the members are not geographically co-located.

Another benefit of OSCthulhu is that it does not require NAT traversal or UDP hole-punching, due to the multicasted traffic being forwarded through a centralized server. Hole-punching techniques, although mostly successful, have been shown to be ineffective in as much as 20% of routers in general use [8]. Since OSCthulhu utilizes a more traditional client-server model, firewalls and routers recognize the traffic passing through as legitimate outgoing and incoming traffic, similar to what you would see from any web-based server.

OSCthulhu makes designing interactive environments a natural process. Because networking is argument focused instead of message focused, the programmer is freed from worrying about maintaining internal mappings of virtual arguments. The programmer only needs to design their program to react accordingly to state changes and output their local changes to the client. The OSCthulhu client does the heavy lifting of maintaining the argument list, responding to the OSCthulhu server commands directly, and updating in response to synchronization cycles. This provides a single universal interface to program against.

4. FUTURE WORK

As more testing is conducted with OSCthulhu, more features will be added and more bug fixing will be conducted. Future features that are currently being researched include the following: The ability to record and playback states as captured by the OSCthulhu Server. An interface for modifying each individual SyncObject contained on the server may prove useful, especially in situations where an errant object may not be modifiable from the performance GUI. If the OSCthulhu system were to gain more exposure one feature that may be deemed beneficial would be a system for locating other OSCthulhu users on the open internet in a more casual manner. Similar to video game networking, a meta-server could be put in place that all OSCthulhu servers register with. Thus, a client could query the meta-server to gain a list of all the OSCthulhu servers running at any point in time in the world, and could connect to

any open (non-password protected) servers. This would allow for spontaneous network collaborations and improvisations to occur, similar to how users join video game servers for spontaneous multiplayer games. Finally, convenience classes/libraries are planned to be constructed for SuperCollider, C++, Java, Processing, and Max/MSP that will take care of much of the boiler-plate code required to create an application that utilizes OSCthulhu.

5. CONCLUSION

The OSCthulhu synchronization system offers network music composers a new choice for enabling network-based compositions and performances. As the results show from the tests conducted, OSCthulhu can be more effective than OscGroups in certain scenarios for networked computer music. If a dislocated ensemble wish to have the fluidity of UDP based networking while maintaining a sufficient level of reliability on the open internet, especially in cases of shared musical resources, then OSCthulhu proves to be a good choice to meet these demands.

6. REFERENCES

[1] A. Barbosa, "Displaced Soundscapes: A Survey of Network Systems for Music and Sonic Art Creation," *Leonardo Music Journal*, vol. 13, pp. 53–59, 2003.

[2] R. Bencina, "Oscgroups," 2010, available from: http://www.audiomulch.com/~rossb/code/oscgroups/ [Accessed 2 May 2010].

[3] C. Brown and J. Bischoff, "Indigenous to the Net: Early Network Music Bands in the San Francisco Bay Area," August 2002, available from: http://crossfade.walkerart.org/brownbischoff/IndigenoustotheNetPrint.html [Accessed 2 August 2010].

[4] Cockos Incorporated, "Ninjam," 2012, available from:http://www.cockos.com/ninjam/ [Accessed February 6th 2012].

[5] Cycling '74, 2010, available from: http://cycling74.com/ [Accessed 27 May 2010].

[6] A. De Campo and J. Rohrhuber, "Waiting and Uncertainty in Computer Music Networks," in *Proceedings of the 2004 International Computer Music Conference*, 2004.

[7] eJAMMING Audio, 2012, available from:http://ejamming.com/ [Accessed February 6th 212].

[8] B. Ford, P. Srisuresh, and D. Kegel, "Peer-to-peer communication across network address translators," 2005, available from: http://www.brynosaurus.com/pub/net/p2pnat/ [Accessed 20 July 2011].

[9] Glitch Lich, 2012, available from: http://glitchlich.com/ [Accessed 10 January 2012].

[10] S. Hagen, *IPv6 Essentials*, 2nd ed. O'Reilly Media, 2006.

[11] S. Jordà, "Faust music online: An approach to collective composition on the internet," *Leonardo Music Journal*, vol. 9, pp. 5–12, 1999.

[12] ResRocket, 2004, available from: http://www.jamwith.us/about_us/rocket_history.shtml [Accessed February 6th 2012].

[13] L. Sasso, *Native Instruments: Reaktor 3- The Ultimate Hands-on Guide for All Reaktor Fans*. Wizoo, 2002.

[14] F. Schroeder, A. B. Renaud, P. Rebelo, and F. Gualda, "Addressing the Network: Performative Strategies for Playing Apart," in *Proceedings of the 2007 International Computer Music Conference*, 2007, pp. 133–140.

[15] W. R. Stevens, *TCP/IP Illustrated*. Addison-Wesley Professional, 1994.

[16] W. R. Stevens, B. Fenner, and A. M. Rudoff, *UNIX Network Programming: The sockets networking API*. Addison-Wesley Professional, 2004.

[17] T. Sweeney, 1999, available from: http://udn.epicgames.com/Three/NetworkingOverview.html [Accessed 16 May 2010].

[18] G. Wang, A. Misra, and P. R. Cook, "Building collaborative graphical interfaces in the audicle," in *NIME '06: Proceedings of the 2006 conference on New interfaces for musical expression*. Paris, France, France: IRCAM — Centre Pompidou, 2006, pp. 49–52.

[19] A. Ward, J. Rohrhuber, F. Olofsson, A. McLean, D. Griffiths, N. Collins, and A. Alexander, "Live Algorithm Programming and a Temporary Organisation for Its Promotion," *Readme Software Art and Culture*, 2004.

[20] G. Weinberg, "Interconnected Musical Networks: Toward a Theoretical Framework," *Computer Music Journal*, vol. 29(2), pp. 23–29, 2005.

[21] S. Wilson, D. Cottle, and N. Collins, Eds., *The SuperCollider Book*. Cambridge, MA: MIT Press, 2011.

[22] M. Wright, 2002, open sound control 1.0 specification. Available from: http://opensoundcontrol.org/spec-1_0 [Accessed 2 May 2010].