# NETWORK MUSIC WITH MEDUSA: A COMPARISON OF TEMPO ALIGNMENT IN EXISTING MIDI APIS

**Flávio Luiz Schiavoni**
Institute of Mathematics and Statistics
University of São Paulo
fls@ime.usp.br

**Marcelo Queiroz**
Institute of Mathematics and Statistics
University of São Paulo
mqz@ime.usp.br

**Marcelo Wanderley**
IDMIL/CIRMMT
McGill University
marcelo.wanderley@mcgill.ca

## ABSTRACT

In network music, latency is a common issue and can be caused by several factors. In this paper we present MIDI network streaming with Medusa, a distributed music environment. To ease the network connection for the end user, Medusa is implemented using different MIDI APIs: Portmidi, ALSA MIDI and JACK MIDI. We present the influence of the MIDI API choice in the system latency and jitter using the Medusa implementation.

## 1. INTRODUCTION

Network music tools can provide an easy way to facilitate cooperation and collaboration in music. Computer networks can be used to connect applications and devices in live music performances, music recording sessions or rehearsals.

To integrate music tools, different data types can be transmitted, such as audio and MIDI. Despite some criticism [1] about the usage of MIDI in Digital Music Instruments, MIDI continues to be popular because it is a standard protocol present in several applications and music devices.

In this paper we will present the network MIDI distribution using Medusa. Medusa is a distributed music environment that allows users to share audio and MIDI streams through computer networks [2]. [1]

Currently, different MIDI APIs can be used to implement a MIDI application in a Linux system, such as ALSA MIDI, Portmidi and JACK MIDI. The Medusa MIDI implementation recently started running on these APIs in order to simplify integration of different tools over a computer network. Moreover, different APIs can influence system latency. In this paper we will present how these APIs can be used and a pragmatic comparison of these APIs in the Medusa implementation.

There are a few other related tools that allow realtime network music content distribution. Netjack [3] is an internal JACK client in JACK2 that is monitored by JACK and synchronized by JACK sample rate and JACK transport. This tool runs over UDP/IP multicast using JACK MIDI only, and allows redundancy in data transmission to avoid glitches [4].

Other network music tools are similar but their music content is limited to a specific musical data type. QmidiNet [2] is a network music tool that uses UDP/IP multicast to distribute MIDI streams. Jacktrip [5] and SoundJack [6] are network music tools that use UDP/IP and provide audio only music streams.

Another popular music data type used for device communication in music is OSC [7]. OSC does not necessarily pack MIDI or audio data and it is a network ready musical protocol. For this reason, we will not discuss OSC in this paper.

The remainder of this paper is organized as follows: Section 2 discusses the MIDI protocol and presents the MIDI APIs used to develop Medusa. Section 3 presents Medusa and our proposed MIDI stream implementation. Section 4 presents measurements done with the Medusa MIDI implementation over the previously explained MIDI APIs and their results. Section 5 presents our conclusion and future work.

## 2. WHY MIDI?

MIDI is one of the most widely-used standard protocols for interconnecting electronic music devices. Proposed as a unidirectional talker-listener network, MIDI was probably the first music standard protocol that created possibilities for music instrument networking [8].

The MIDI protocol facilitates integration between different music applications and devices in a single computer. Some reasons that motivated the development of MIDI in Medusa are:

- The MIDI protocol enables using digital music interfaces and synthesizers as an alternative or complement to audio transmission channels by reducing network bandwidth usage.
- MIDI messages can control several music devices and equipment like mixers and software plugins.
- MIDI Machine Control (MMC) controls recorders and provides messages that include Play, Fast Forward, Rewind, Stop, Pause, and Record.

[1] The source code of Medusa is available on the project website: http://sourceforge.net/projects/medusa-audionet

[2] http://qmidinet.sourceforge.net/qmidinet-index.html

- MIDI Time Code (MTC) is a time protocol that can be used to sync applications and devices such as loopers and sequencers.
- MIDI Show Control (MSC) is a protocol developed to control equipment in theaters, live performance, multimedia installations and similar environments.

These messages can be used to sync and control MIDI applications and devices that are usually directly connected. The usage of network MIDI streams can expand the control and integration possibilities of the MIDI protocol to a network distributed sound-processing software/hardware environment.

Different MIDI APIs help software developers to integrate MIDI in music applications.

## 2.1 MIDI APIs

Initially Medusa was implemented with JACK MIDI only [2]. Since some applications do not implement a JACK MIDI channel, two other MIDI APIs were integrated in Medusa: ALSA MIDI and Portmidi.

Since a MIDI event is supposed to be delivered immediately, the MIDI protocol does not have time-tagging [9]. On the other hand, the MIDI APIs here discussed have complements to the MIDI protocol that allow synchronizing different applications. How the application deals with event sync is an important feature in order to achieve low latency.

### *2.1.1 ALSA MIDI*

ALSA, the Advanced Linux Sound Architecture, is the part of the Linux kernel that provides support to USB and PCI audio / MIDI devices. One of the basic components of ALSA system is the device driver for sound equipment.

ALSA also provides an API for application development. This API includes all the required features for developing audio and MIDI applications that run over ALSA drivers.

ALSA provides two different MIDI event types for MIDI application development: seq and raw. ALSA seq (sequencer) timestamps MIDI messages and monitors ALSA MIDI software that is bypassed. ALSA raw just operates MIDI drivers without timestamping.

In ALSA, all MIDI applications are mapped as virtual devices and there is no significant difference between physical and virtual devices. Once the application has created a virtual device, this port stream can be routed to other devices through a port connection. Some applications can be used to manage MIDI connections in ALSA, e.g. qjackctl [3] .

In ALSA seq MIDI, an event uses MIDI ticks for timestamping (a discrete time measure related to a track-specific tempo). Other timestamp information is the relative note time in seconds and nanoseconds.

### *2.1.2 Portmidi*

Portmidi is part of Portmedia project, a cross-platform API for music application development. This library supports real-time input and output of MIDI data and runs on Windows, MacOS, and Linux [10].

Using Portmidi API, it is possible to list the MIDI devices, physical or virtual, and present their input and output ports.

Portmidi in Linux runs over ALSA, but differently from ALSA MIDI it creates a data stream directly connected to a MIDI device port. This port connection cannot be changed or routed differently during the application execution.

In Portmidi, a MIDI event has a timestamp related to a Portmidi internal clock time.

### *2.1.3 JACK MIDI*

JACK (JACK Audio Connection Kit) is a real time low-latency sound server that allows the creation of audio and MIDI connections between applications that run on the JACK API [3]. This sound server runs over different operating systems such as Linux, Windows and MacOS.

Like ALSA MIDI, JACK MIDI can be used to connect JACK MIDI capable applications and route MIDI streams between them. However, Jack MIDI does not access ALSA MIDI hardware but only FFADO (firewire) MIDI hardware.

Some software bridges, like a2jmidi / j2amidi [4] , can interface Jack MIDI with ALSA MIDI devices in Linux systems. These applications are an alternative for connecting ALSA MIDI hardware to JACK MIDI capable applications.

Regarding its performance, there is virtually no jitter in JACK MIDI and it is sample accurate. JACK MIDI event process runs with audio sample blocks and for that reason the system latency can be tuned by adjusting JACK sample rate and process block size.

JACK MIDI event has a timestamp concept that is not directly associated with a time clock. It is associated with the sample position in the audio block associated to this MIDI event. For this reason, this concept of MIDI event time depends on JACK audio block size and sample rate.

### *2.1.4 Theoretical comparison*

Despite the fact that these APIs have different approaches and features, we have grouped some of these features for a theoretical comparison. A summary of this comparison is presented in Table 1.

| Feature | ALSA MIDI | Portmidi | JACK |
|---|---|---|---|
| Multi-stream | Yes | No | Yes |
| Cross-platform | No | Yes | Yes |
| Virtual devices | Yes | Yes | Yes |
| Multiples devices | Yes | No | Yes |
| Multiple connections | Yes | No | Yes |
| Hardware devices | ALSA | ALSA | FFADO |

**Table 1**. MIDI APIs theoretical comparison

ALSA and JACK are the default sound server + driver in Linux audio context. Being complementary and not concurrent, the choice between one of these APIs depends on the hardware and software involved. Portmidi in Linux is

[3] http://qjackctl.sourceforge.net/

[4] http://home.gna.org/a2jmidid/

implemented over ALSA MIDI and combines the possibility of a direct device connection and porting the application to other operating systems.

## 3. MEDUSA

Medusa is a music network environment tool developed to simplify multichannel audio and MIDI distribution in computer networks. The Medusa development is divided into two main lines: 1) a common library (libmedusa) that contains the network connections, audio and MIDI pack and transformation, and 2) some specific implementations to connect libmedusa with different sound APIs [11].

The development is organized in a three layer architecture, as depicted in Fig 1.

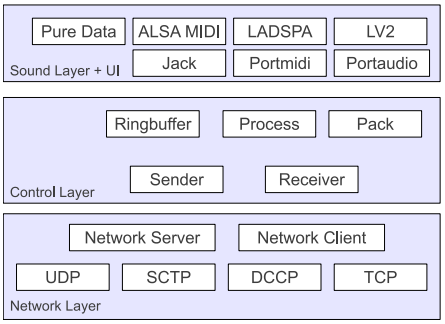


**Figure 1**. Medusa architecture

The **Network layer** is responsible for creating the network connection and data transmission. Rather than using only UDP for communication, Medusa implements different network transport protocols, namely UDP [12], DCCP [13], TCP [14] and SCTP [15]. Thus, the user can choose between a faster and unreliable protocol, namely UDP or DCCP, or a slower and reliable protocol such as TCP or SCTP. We grouped the protocol dependent implementations using two abstractions: a server that sends data to the network and a client that receives data from the network.

The **Control layer** is responsible for data management, data packing / unpacking, and data transformation. Medusa has two main roles in this layer: sender and receiver. Thus, a user can choose to provide a network resource as a sender or to consume a networked resource as a receiver.

Different sounds APIs have different ways to play / capture audio or MIDI data and need special implementations. The **Sound layer** is outside libmedusa and is responsible for connecting Medusa to specific audio / MIDI systems. For example, Pure Data uses its own object graphic interface, and a LADSPA plugin has a particular GUI; the sound layer puts together the sound API and the user application interface.

### 3.1 Medusa data flow

To implement MIDI communication, we developed two different data flows inside Medusa, one for audio and other for MIDI, as presented in Fig. 2. While audio can be processed, fragmented, converted from different sample rate, bit depth and byte order, MIDI data flow packs the data with meta-data to transmit it through the network. The audio data flow converts the audio data to a common format. With this feature, Medusa can interchange audio and MIDI data between applications with different audio configurations or be connected through different sound APIs.

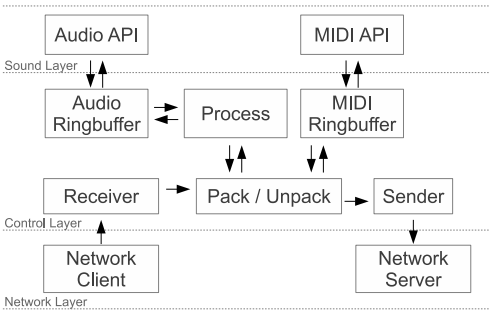


**Figure 2**. Internal Audio and MIDI data flow

### 3.2 Medusa package

The Control layer packs the sound data for transmission. This application package has additional meta-data that helps Medusa management. A field in Medusa package identifies the data type (audio or MIDI) and the data channel number. The Medusa_package header is presented in Fig. 3.

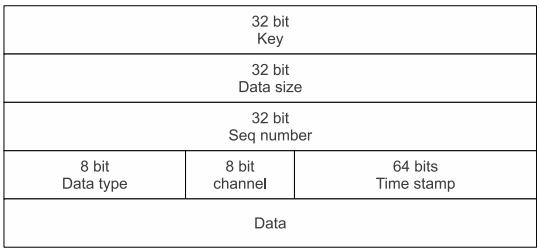


**Figure 3**. Medusa Package

In addition to the data identification and channel addressing, the package contains a sequential number to verify data loss, a timestamp to measure latency and synchronization, and a key to separate Medusa data from some possible network interference.

### 3.3 Medusa loopback channel

Every network socket is full duplex. Once we have separated the sender and receiver roles, this socket feature was used to implement a loopback channel.

The Medusa loopback channel allows senders and receivers to measure network performance during data transmission. When loopback mode is enabled, for every Medusa package sent by the server, the client will reply with a loopback message. The structure of a loopback message, as presented in Fig. 4, adds two new fields to the Medusa package. These fields represent the time when the client received the data package and the time when the client played the data from the package.

With this implementation it is possible to measure a network performance in different stages: a) the sender adds

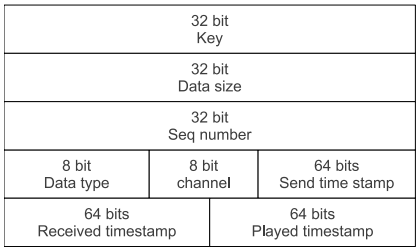


**Figure 4**. Medusa Loopback Package

the sending timestamp when the data is ready to be sent; b) when the receiver acquires the package, it creates a loopback package using the same packet header timestamped with the receiving time; c) when the receiver plays the package data it timestamps the loopback package and sends it back to the sender; d) the sender returns all the loopback information to a callback function including the time it was received.

Thus, a Medusa application can implement a loopback callback function to measure latency in different communication stages.

## 4. PRACTICAL PERFORMANCE MEASUREMENT

Using Medusa loopback channel we can compare time performance variations between Medusa implementations using the different MIDI APIs, thus verifying how the choice of MIDI API influences system latency.

### 4.1 Measurement environment

To measure the system latency a MIDI sequencer was used to produce note-on and note-off commands. We chose Qmidiarp [5] for these tests because it is a MIDI sequencer that runs over ALSA MIDI and JACK MIDI. Thus, we did not need to run an extra software bridge to interface between different APIs.

We set the test loop time to 120 bpm playing two half notes per bar, which means one note-on and one note-off per second. The note duration was set to zero which should result in one note-off event immediately following the note-on. The test was done by executing the loop 1300 times which lasted approximately 11 minutes.

We ran two Medusas in the same machine connected using localhost address. We decided to run these tests on localhost because the sender and receiver can thus use the same clock to timestamp the packages. The first Medusa was sending the MIDI events generated by Qmidiarp and receiving them back from the second instance which was set up as a loopback channel.

Theoretically every note-on should be played one second apart from the preceding and the succeeding note-on, the same being true of each note-off. We will call this theoretical time "expected_t(i)" and the actual time when each event was measured "t(i)". We assumed that the first event would happen at time 0 and so all the other expected note times are integer times relative to the first note. Since the first note can also have some latency, we calculated the minimum difference between all actual note times and corresponding expected note times, as presented in Eqn.1.

$$min\_t = |min(t(n) - expected\_t(n))| \quad (1)$$

We calculated the latency as an average difference between the relative note time and the expected note time adding the minimum latency to all values (Eqn. 2).

$$latency(\Delta t) = \frac{1}{n}\sum_{i=1}^{n}(t(i) - expected\_t(i) + min\_t) \quad (2)$$

The jitter was calculated as the mean latency deviation (Eqn. 3).

$$jitter = \frac{1}{n}\sum_{i=1}^{n}|rt(i) - \Delta t| \quad (3)$$

We also measured the note-off time based on the average difference between note-off and note-on (4).

$$\Delta Note\_off = \frac{1}{n}\sum_{i=1}^{n}(note\_off(i) - note\_on(i)) \quad (4)$$

### 4.2 Performance tests results

We calculated the system latency in 4 different stages: 1) sender transmission time, 2) receiver acquiring time, 3) receiver playing time and 4) sender loopback receiving time, as depicted in Table 2.

| API | Time 1 | Time 2 | Time 3 | Time 4 |
|---|---|---|---|---|
| ALSA MIDI | 0.268 | 0.370 | 0.745 | 3.045 |
| Portmidi | 2.380 | 2.574 | 2.907 | 5.196 |
| JACK MIDI | 3.923 | 4.033 | 14.374 | 14.590 |

**Table 2**. Latency measurements (times in ms)

Since the latency varied during the performance, we also calculated the latency deviation, or jitter, as presented in Table 3.

| API | Time 1 | Time 2 | Time 3 | Time 4 |
|---|---|---|---|---|
| ALSA MIDI | 0.284 | 0.295 | 0.674 | 1.939 |
| Portmidi | 0.609 | 0.936 | 0.937 | 2.016 |
| JACK MIDI | 2.749 | 3.950 | 2.748 | 2.759 |

**Table 3**. Jitter measurements (times in ms)

The average time difference between a note-on and a note-off is presented in Table 4. This table also presents the percentage of note-off events with the same time of the note-on event and latency jitter.

### 4.3 Data analysis

The "Time 1" column in Table 2 and Table 3 presents the latency and jitter in the server. This data represents how accurately the API would play the notes locally. These tables

[5] http://qmidiarp.sourceforge.net/

| API | note-off | jitter | % of same time |
|---|---|---|---|
| ALSA MIDI | 0.008 | 0.009 | 56% |
| Portmidi | 0.033 | 0.046 | 49.5% |
| JACK MIDI | 0.002 | 0.003 | 86% |

**Table 4**. Note-off time and jitter (times in ms)

confirm that JACK is the biggest latency adder for playing each note. The data do not confirm the JACK MIDI theoretical features regarding event synchronization. Analyzing the chosen tool documentation, we observed that Qmidiarp does not use JACK's MIDI event sample alignment feature. We repeated the experiment with other MIDI tools such as j2a, j2amidi_bridge and Hydrogen, but since they also do not implement this sample alignment feature, they all present the same issue.

Some other tools implemented over JACK use it only for audio streams, and use other APIs for MIDI connection; examples of these tools are Pure Data, QTractor, RoseGarden and LMMS. Even if we had chosen one of these tools to obtain a more precise MIDI event time, this accuracy would be lost in the MIDI bridge between ALSA MIDI and JACK MIDI. The only implementation/setup that presented MIDI event sample alignment with JACK was the FFADO driver with a firewire sound interface.

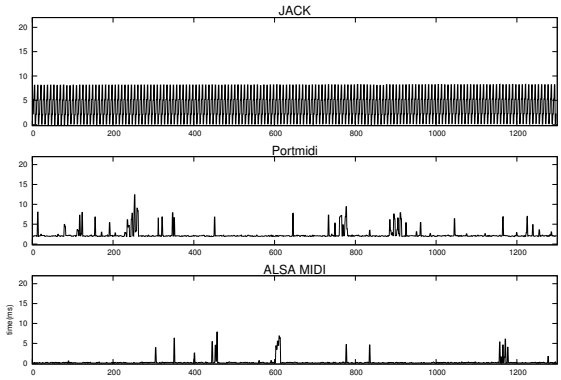


**Figure 5**. MIDI input latency (Time 1)

The result of the input measurement is depicted on Fig. 5. In our experiment we noticed that every MIDI event in JACK was aligned with the first sample of the block, which explains why JACK latency looks like a sawtooth waveform.

In Fig. 5 the worst latency time of a MIDI event is 7.851 ms with ALSA MIDI, 10.140 ms with JACK and 12.390 ms with Portmidi.

Portmidi is just a wrapper for developing portable music applications. In Linux, it runs over ALSA MIDI. Because Portmidi uses ALSA MIDI through a wrapper interface, it is predictable that ALSA MIDI performance would exceed Portmidi. In spite of ALSA's better outcome, its implementation is not portable and can not be used in other operating systems.

The second column, "Time 2", presents the time that Medusa spent to send the event from the server to the receiver. Subtracting Time 1 from "Time 2", Medusa latency is between 0.1 ms and 0.2 ms for every API. Since we are running on the localhost, this time can be understood as the time that the system spends to pack the MIDI data, copy the data to the kernel space, copy it back to user space and finally unpack the data. Medusa transmission time can be considered small if compared with the API latencies.

The third measured latency is the receiver playing time ("Time 3"), presented in Fig.6. Again, JACK performance can be understood as a simplistic implementation of JACK MIDI sync. Moreover, JACK default is to sync MIDI events with audio blocks and because of this the JACK configuration influences MIDI latency. In our tests, JACK was configured with 48Khz sample rate and a block size of 512 samples, meaning that every JACK block is about 10.67ms long. Since Medusa is not implemented as a JACK internal client, all received data has to wait for the next JACK block to be executed. At this stage, ALSA MIDI and Portmidi latency is very close.

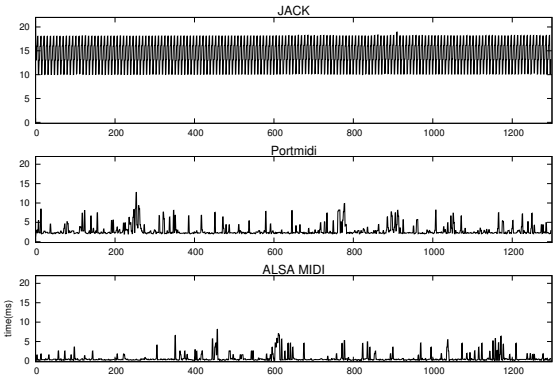


**Figure 6**. MIDI output latency

The last latency measured, "Time 4", is the loopback time. Since the Medusa loopback package is bigger than a Medusa MIDI package, the time to receive a loopback message is bigger than the time to receive a MIDI message. These data show that a round trip time to send both MIDI data and a loopback package is not simply twice the time to send a package. Furthermore, the loopback time in Medusa also includes the time to process the data in the receiver.

The note-off time, presented in Table 4, can be interpreted as how fast an API responds to two MIDI events occurring at the same time. Knowing that the application was programmed to send a note-on and a note-off at the same time, the way the application reads these events defines how synchronized the events really are. Since JACK aligns both events with the same sample, it will always be more precise having the majority of events occurring simultaneously.

## 5. CONCLUSIONS

In this paper we presented the Medusa MIDI implementation using different MIDI APIs. We presented these API's features and some theoretical comparisons between them. We also presented the Medusa architecture and how this tool can be used to obtain feedback about the data transmission. We used the Medusa loopback channel and a

MIDI sequencer to measure the latency in a network session.

Regardless of the best time synchronization advertised by JACK MIDI, in our measurements we discovered that the majority of JACK MIDI applications do not use the event synchronization features present in this API. In our tests, the JACK MIDI sync feature was present only in a firewire MIDI interface driver. Unfortunately, several MIDI interfaces use a USB connection and are available in Linux only by ALSA API. Future work should include proposing a better implementation for JACK MIDI applications concerning the JACK MIDI event sample alignment.

In the latency tests, ALSA MIDI presented the best latency performance. Unfortunately this API exists exclusively for Linux and its implementation cannot be ported to other operating systems.

Portmidi is a portable API and a solution for porting Medusa to other operating systems. Portmidi creates a wrapper to the operating system's native API and helps developers in creating portable music applications. Future work includes testing Portmidi performance in other operating systems.

Our experiments also presented how the different stages of network communication influence latency. Another important result was to verify that round trip time does not reflect directly the time between the data capture in the sender and the data playing in the receiver.

Since each MIDI API has a different approach and use context, the present paper does not intend to judge these APIs but to present some limits to MIDI network streaming using Medusa.

It is also important to affirm that different implementations using these APIs may have different results depending on how the API is implemented or how the application is executed.

In the future, we intend to consider sending MIDI timestamp events in the Medusa package to ensure a better network MIDI synchronization between these different APIs.

## 6. ACKNOWLEDGMENTS

The authors would like to thank the Linux Developers community and all open source software authors. Without their anonymous help this project would not have been possible.

Some friends helped with ideas, feedback and reviews: Wissam Saliba, Pedro Henrique de Faria, Siddhartha Shankar Rana, Egor Sanin, Marcello Giordano and Stephen Sinclair. A special thanks to R. Michael Winters for proofreading.

The authors would like also to thank the support of the funding agencies CNPq (grant no 141730/2010-2), FAPESP - São Paulo Research Foundation (grant no 2008/08623-8) and CAPES (grant no BEX 1194/12-7).

## 7. REFERENCES

[1] F. R. Moore, "The dysfunctions of MIDI," *Computer Music Journal*, vol. 12, no. 1, pp. 19–28, 1988.

[2] F. L. Schiavoni, M. Queiroz, and F. Iazzetta, "Medusa - a distributed sound environment," in *Proceedings of the Linux Audio Conference*, Maynooth, Ireland, 2011, pp. 149–156.

[3] S. Letz, N. Arnaudov, and R. Moret, "What's new in JACK2?" in *Proceedings of the Linux Audio Conference*, LAC, Ed., Parma, Italy, 2009, pp. 1–9.

[4] A. Carôt, T. Hohn, and C. Werner, "Netjack–remote music collaboration with electronic sequencers on the internet," in *Proceedings of the Linux Audio Conference*, Parma, Italy, 2009, pp. 118 – 122.

[5] C. Chafe, S. Wilson, A. Leistikow, D. Chisholm, and G. Scavone, "A simplified approach to high quality music and sound over IP," in *In Proceedings of the COST G-6 Conference on Digital Audio Effects (DAFX-00*, 2000, pp. 159–164.

[6] A. Carôt, A. Renaud, and B. Verbrugghe, "Network Music Performance (NMP) with Soundjack," in *In Proceedings of NIME 2006 Network Performance Workshop*, 2006.

[7] M. Wright, "Open Sound Control: an enabling technology for musical networking," *Organised Sound*, vol. 10, pp. 193–200, 2005.

[8] G. Loy, "Musicians make a standard: The MIDI phenomenon," *Computer Music Journal*, vol. 9, no. 4, pp. pp. 8–26, 1985. [Online]. Available: http://www.jstor.org/stable/3679619

[9] MIDI Manufacturers Association, "White paper: Comparison of MIDI and OSC," http://www.midi.org/aboutmidi/midi-osc.php, Nov. 2008.

[10] R. Bencina and P. Burk, "Portaudio - an open source cross platform audio api," in *Proceedings of the International Computer Music Conference*, 2001.

[11] F. L. Schiavoni and M. Queiroz, "Network distribution in music applications with medusa," in *Proceedings of the Linux Audio Conference*, Stanford, USA, 2012, pp. 9–14.

[12] J. Postel, "User Datagram Protocol," RFC 768 (Standard), Internet Engineering Task Force, Aug. 1980. [Online]. Available: http://www.ietf.org/rfc/rfc768.txt

[13] E. Kohler, M. Handley, and S. Floyd, "Datagram Congestion Control Protocol (DCCP)," RFC 4340 (Proposed Standard), Internet Engineering Task Force, Mar. 2006, updated by RFCs 5595, 5596. [Online]. Available: http://www.ietf.org/rfc/rfc4340.txt

[14] M. Padlipsky, "TCP-on-a-LAN," RFC 872, Internet Engineering Task Force, Sep. 1982. [Online]. Available: http://www.ietf.org/rfc/rfc872.txt

[15] L. Ong and J. Yoakum, "An Introduction to the Stream Control Transmission Protocol (SCTP)," RFC 3286 (Informational), Internet Engineering Task Force, May 2002. [Online]. Available: http://www.ietf.org/rfc/rfc3286.txt