Network distribution in music applications with Medusa

Flávio Luiz SCHIAVONI and **Marcelo QUEIROZ**
Computer Science Department
University of São Paulo
São Paulo
Brazil,
{fls, mqz}@ime.usp.br

Abstract

This paper introduces an extension of Medusa, a distributed music environment, that allows an easy use of network music communication in common music applications. Medusa was firstly developed as a Jack application and now it is being ported to other audio APIs as an attempt to make network music experimentation more widely accessible. The APIs chosen were LADSPA, LV2 and Pure Data (external). This new project approach required a complete review of the original Medusa architecture, which consisted of a monolithic implementation. As a result of the new modular development, some possibilities of using networked audio streaming via Medusa plugins in well-known audio processing environments will be presented and commented on.

Keywords

Network Music, Pure Data, LADSPA, LV2, Medusa.

1 Introduction

Ever since the availability of network and Internet connections became an undisputed fact, collaborative and cooperative music tools have been developed. The desire of using realtime audio streaming in live musical performances inspired the creation of various tools focused on distributed performance, where synchronous music communication is a priority. Some related network music tools which address the problem of synchronous music communication between networked computers are NetJack [Carôt et al., 2009], SoundJack [Carôt et al., 2006], JackTrip [Cáceres and Chafe, 2009b; Cáceres and Chafe, 2009a], llcon [Fischer, 2006] and LDAS [Sæbø and Svensson, 2006].

The success cases of network music performance concerts could give the wrong impression that the only use for network music is distributed performance, but several other problems in computer music can take advantage of or benefit from network music distribution. For instance, recordings can be made using a pool of networked computers, a resource which might be seen as a scalable distributed sound card; music spatialization could be done using a mesh network topology; digital signal processing power can be vastly improved using clusters of computers; and musical composition may explore fresh new grounds by viewing computer networks as nonstandard acoustical environments for performance and listening.

There are at least two requirements for all these scenarios to be fully explorable by potentially interested users, which are often musicians or aficionados and usually nonprogrammers; first, flexible audio network tools must be made available, and second, easy integration with popular music/sound processing applications must be guaranteed. Despite the fact that most high-level linux music applications nowadays run over Jack and ALSA, we see that time and again end users can't deal with this audio infrastructure lying below the application they are running. Also, the lack of automated setup and graphical user interfaces shun common users from many existing tools, like the network music tools mentioned above.

The work presented in this paper is part of the investigation behind the development of Medusa [Schiavoni et al., 2011], a distributed audio environment. Medusa is a FLOSS project which is primarily focused on usability and transparency, as means to making network music connections easier for end users. The first implementation of Medusa, presented in LAC 2011, was developed in C++ with Qt GUI and Jack as sound

API. The Jack API was extended with a series of network functionalities, such as add/remove remote ports and remote control of the Jack Transport functionality.

Recently, the development of Medusa has been strongly guided by the attempt to deal with the two aforementioned requirements, namely flexibility and integrability. These goals may be reached by extending regular sound processing applications, such as Pure Data, Rosegarden or Ardour, allowing them to function as network music tools. The very basic idea is trying to reach end users wherever they already are.

Most music applications can be extended by plugins: Pure Data can be extended by the creation of C externals (which might be viewed as a plugin), whereas digital audio workspaces such as Ardour, Rosegarden, Qtractor and Traverso can be extended by LADSPA, VST and LV2 plugins. In this paper we will explore the possibility of developing Medusa network music plugins using three popular audio APIs: LADSPA, LV2 and Pure Data API (via C externals). It is worth mentioning that these APIs are all open-source, developed in C and widely used in Linux music environments.

A reimplementation of Medusa has been required in order to grant code reuse in the implementation of these plugins, and also for easy maintenance of the source code. All source code of the Medusa project, including Medusa plugins, are freely available in the project site[1]. This reimplementation of Medusa and the proposed architecture are presented in section 2; section 3 discusses the chosen audio APIs and presents the developed plugins, and section 4 brings some conclusions and a discussion of future works.

2 Medusa

Although some promising preliminary results had been achieved with the first version of Medusa [Schiavoni et al., 2011], the original monolithic implementation raised several difficulties in the implementation of the proposed Medusa plugins, which eventually triggered a fundamental architectural change in the project. First, the implementation language was changed from C++ to ANSI C, which is more compatible with the chosen sound APIs. Second, the monolithic structure has been changed to the development of a core Medusa library (libmedusa.so), comprising control and network functions, which could be re-used by each new plugin implementation. Third, graphical user interfaces, text-based interfaces and plugins now occupy a separate layer, that uses the core Medusa library as an API on its own.

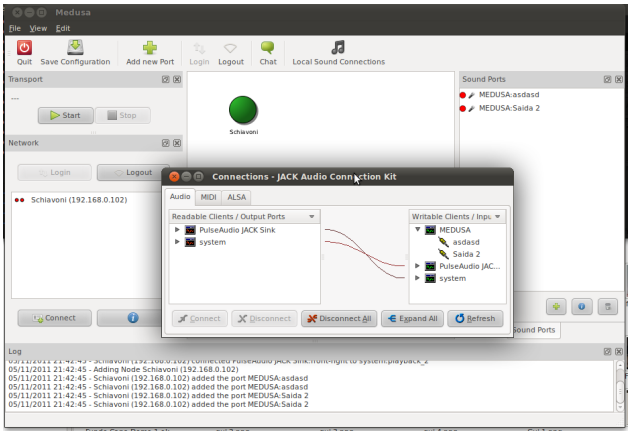

Figure 1: Medusa Jack implementation with Qt GUI

The new Medusa architecture is divided in three layers: Sound, Control and Network. The Control and Network layers are implemented as a unique library and are used by all Medusa implementations. The Sound layer corresponds to specific applications, like the proposed plugins, that are built using each plugin API and the Medusa library. This architecture facilitates code maintenance and bug correction.

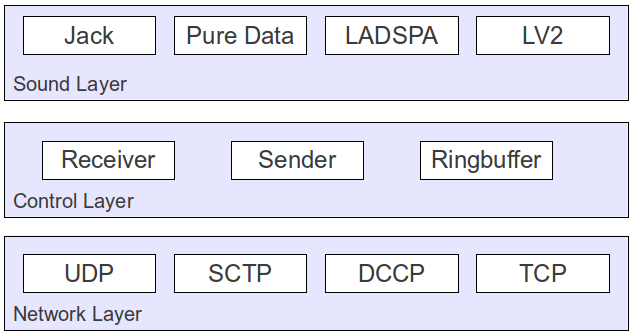

Figure 2: Architectural view of the implementation

[1] http://sourceforge.net/projects/medusa-audionet/

2.1 Network layer

The network layer is responsible for managing connections between network clients and servers. This layer's implementation was made based on a fixed set of network transport protocols, which are normally provided as part of the operating system kernel, since its development and deployment requires superuser privileges. Medusa currently allows the user to choose among 4 transport protocols: UDP, TCP, SCTP e DCCP:

UDP [Postel, 1980]: User Datagram Protocol is the classical unreliable (but faster) transport protocol.

TCP [Padlipsky, 1982]: Transmission Control Protocol is a reliable transport protocol, which ensures absence of packet losses.

SCTP [Ong and Yoakum, 2002]: Stream Control Transmission Protocol is a connection-oriented transport protocol that provides a reliable full-duplex association. This protocol was not originally meant as a replacement for TCP, but was developed for carrying voice over IP (VoIP).

DCCP [Kohler et al., 2006; Floyd et al., 2006]: Datagram Congestion Control Protocol is a transport protocol that combines TCP-friendly congestion control with unreliable datagram semantics for applications that transfer fairly large amounts of data [Lai and Kohler, 2005].

This multi-protocol network communication layer is intended to offer alternatives for users that may need different specific features in data transfer according to application context. For instance, an interactive musical performance with strong rhythmic interactions may require the smallest possible latency while tolerating audio glitches due to packet losses, and a remote recording session may tolerate high latency and jitter, but still require that every packet be delivered. With a few alternatives available, the user may choose which protocol is more appropriate to its own musical use.

2.2 Control layer

The next layer in the Medusa API is the Control layer. The Control layer essentially creates senders and receivers and provides them to the Sound layer. Instead of requiring that all protocols in the Network layer to have full-duplex communication, the Control layer always separates the roles of sending and receiving network data.

Figure 3: Sender / Receiver communication

In order to create a sender it is necessary to inform the network protocol, the network port and the number of channels that will be sent. The sender creates the network server and one ring buffer for each channel, and provides functions for the sound API to have easy access to these buffers. Each ring buffer receives sound content from applications (usually in DSP chunks), out of which the sender prepares network chunks (usually of a different size) for the Network Layer.

The receiver is created in a way similar to the sender, but besides the basic parameters the corresponding server IP address is also required. The receiver creates a network client and the required number of ring buffers (one per channel). Unlike the sender, the ring buffer of the receiver will be fed by a network client and consumed by the sound API.

2.3 Sound layer

The outermost Medusa layer is the Sound layer, which deals not only with audio streams but also with MIDI streams. The Sound layer lies between the Medusa API and several sound application APIs, and represents a collection of Medusa front-ends or interfaces, since each integration of Medusa with a particular sound application may have its own user interface, defined by each sound application API.

The integration of Sound and Network layers uses the Control layer through its sender / receiver plugins. Sender and receiver roles defined by the Control layer are converted into Sound layer plugins that simply exchange audio streams between machines.

Each plugin has a host application and a particular way to communicate with it. The API defines how a plugin is initialized, how it processes data and how it is presented. Each plugin implemen-

tation generates an independent software package that can be individually used and distributed. The separation of these implementations avoids the mixing of different plugin libraries, which is better for code maintenance, chasing bugs and so on.

3 Implementations

3.1 LADSPA

LADSPA [Furse, 2000] stands for "Linux Audio Developer Simple Plugin API", and it is the most common audio plugin API for Linux applications. Many Linux programs, such as Audacity, Ardour, Rosegarden, QTractor and Jack Rack, support LADSPA plugins. The LADSPA API is captured within a header file and is very easy to use. Some examples are provided with an SDK and there is a lot of open source code with great documentation available.

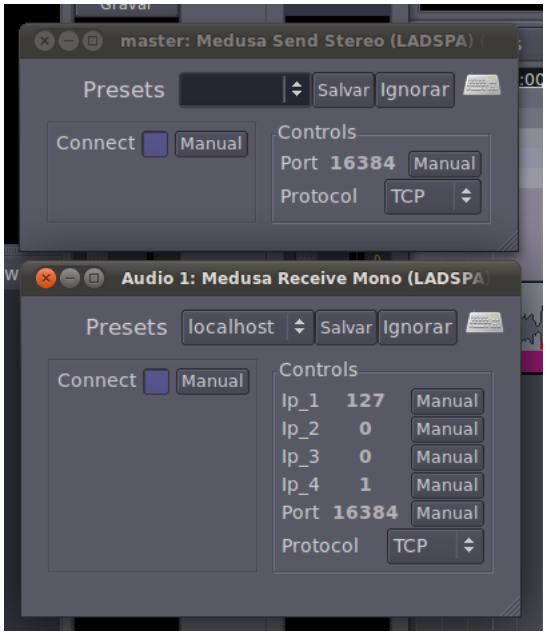

Figure 4: Medusa LADSPA implementation

Figure 4 presents Medusa LADSPA sender and receiver interfaces. The controls of a LADSPA plugin are represented exclusively by 32 bit floating point numbers, and it was awkward to use these to represent IP addresses. LADSPA GUIs are defined in RDF files that cannot be changed on-the-fly, which makes them very cumbersome as interfaces for network audio connections. LADSPA doesn't support MIDI and the documentation suggests the use of Rosegarden DSSI to deal with it.

3.2 LV2

LV2 (LADSPA version 2) [Steve Harris, 2008] is acknowledged as the official LADSPA successor. An LV2 plugin is a bundle that includes the plugin itself, an RDF descriptor in Turtle and any other resource needed. The LV2 bundle may include a GTK GUI, thus offering developers a way to create better user interfaces. As LADSPA, LV2 is an easy-to-use library, and plenty of documentation and examples are provided by the developers.

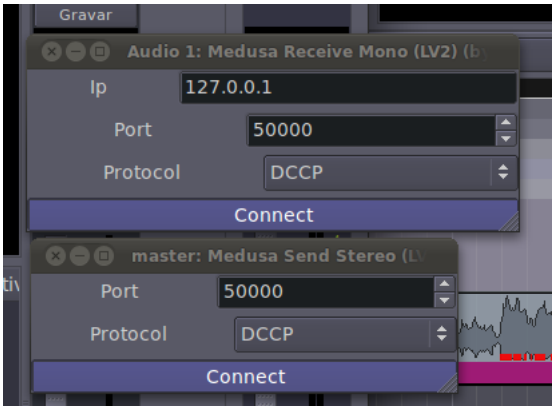

Figure 5: Medusa LV2 implementation

The possibility of creating GTK interfaces allowed a more natural integration of Medusa as an LV2 plugin. The IP mask entry allows easy input for the user, and on-the-fly changes to the interface may be used by Medusa in the future to allow for finding users and sound resources and keeping this information up-to-date on the interface.

3.3 Pure Data external

Pure Data [Puckette, 1996] (aka Pd) is a largely used realtime programming environment for audio, video, and graphical processing, which can be extended by the use of externals written in C/C++ language [Zmölnig, 2001]. Pure Data does have some network externals available, but none offers all transport protocols provided by Medusa. A Pd external may also be implemented with a tcl/tk GUI, which can be useful to improve usability and also to implement Medusa service control in the future. In addition, Pd offers a good environment to measure latency, packet loss and jitter in Medusa Network layer implementation.

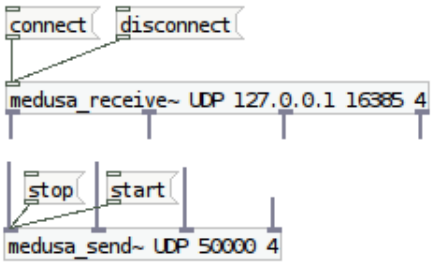

Figure 6: Medusa Pure Data external implementation

The Medusa Pd external implementation is actually a collection of externals (a library) that includes the sender and receiver objects (see figure 6) and a network meter to measure latency and packet loss. Some examples of use were also developed and are available on the project website.

4 Conclusions and Future Works

The possibility of exchanging sound data between applications using the Jack sound server, for instance, has brought new perspectives in audio software development and usage. Extending this possibility to allow for network data exchange from within popular user applications may facilitate the emergence of new ways to compose, record or play music. It may be early to conclude that network plugins for audio software will really expand the musical use of computer networks, but it is not early to observe in users and musicians new expectations about what might be done in network music, and also the desire to try out new ways to do old things, maybe more easily and more transparently than before.

The effort that went into changing Medusa architecture and reimplementeing it is totally justified, in the sense that now the effort of implementing other network music plugins (i.e. Medusa plugins for other sound applications) involve a lot of code reuse and thus have been made much lighter.

On the other hand, the first version of Medusa had an additional structure called Control Service, which was responsible for helping users to connect to network music resources in a transparent way, in other words, without having to specify IP, network ports or audio configuration, using broadcast messages to publish networked audio resources.

With that in mind, the level of usability originally pretended by this project will only be reached when Medusa's control service is implemented in the presented Medusa plugins, which is the very next step in our work. This control service is going to be responsible for improving transparency and usability, by including a discovery service, and a name server to publish networked music resources, thus allowing users to refer to other users and audio streams, instead of IP addresses and network ports. Probably the Medusa LADSPA plugin will have to be abandoned at this point because the control service is impossible to implement without on-the-fly GUI modification (although the current version of the plugin, without control service, should be kept).

Up to this point the plugin development has focused on audio streaming, but some other features must soon be addressed, such as treating MIDI streams and designing better GUIs. There are also other sound APIs that are been investigated and maybe soon will be incorporated in Medusa Sound layer.

Acknowledgements

Thanks to uncountable developers of LADSPA, LV2 and Pure Data externals and their amazing open source code. Without their anonymous help this project would not have been possible.

Thanks also go to André Jucovsky Bianchi, Beraldo Leal, Santiago Davila, Antônio Goulart, Giuliano Obici, Danilo Leite and the Computer Music Group of IME/USP for their interest, feedback and support.

This work has been supported by the funding agencies CNPq (grant 141730/2010-2) and FAPESP - São Paulo Research Foundation (grant 2008/08623-8).

References

Juan-Pablo Cáceres and Chris Chafe. 2009a. Jacktrip: Under the hood of an engine for network audio. In *Proceedings of International Computer Music Conference*, page 509–512, San Francisco, California: International Computer Music Association.

Juan-Pablo Cáceres and Chris Chafe. 2009b. Jacktrip/Soundwire meets server farm. In *In Proceedings of the SMC 2009 - 6th Sound*

and Music Computing Conference, pages 95–98, Porto, Portugal.

A. Carôt, U. Kramer, and G. Schuller. 2006. Network music performance (NMP) in narrow band networks. In *Proceedings of the 120th AES Convention*, Paris, France.

A. Carôt, T. Hohn, and C. Werner. 2009. Netjack–remote music collaboration with electronic sequencers on the internet. In *In Proceedings of the Linux Audio Conference*, page 118, Parma, Italy.

Volker Fischer. 2006. Internet jam session software. `http://llcon.sourceforge.net/`.

S. Floyd, E. Kohler, and J. Padhye. 2006. Profile for Datagram Congestion Control Protocol (DCCP) Congestion Control ID 3: TCP-Friendly Rate Control (TFRC). RFC 4342 (Proposed Standard), March. Updated by RFC 5348.

Richard Furse. 2000. Linux audio developer's simple plugin api (ladspa). `http://www.ladspa.org/`.

E. Kohler, M. Handley, and S. Floyd. 2006. Datagram Congestion Control Protocol (DCCP). RFC 4340 (Proposed Standard), March. Updated by RFCs 5595, 5596.

Junwen Lai and Eddie Kohler. 2005. A congestion-controlled unreliable datagram api. http://www.icir.org/kohler/dccp/ nsdiabstract.pdf.

L. Ong and J. Yoakum. 2002. An Introduction to the Stream Control Transmission Protocol (SCTP). RFC 3286 (Informational), May.

M.A. Padlipsky. 1982. TCP-on-a-LAN. RFC 872, September.

J. Postel. 1980. User Datagram Protocol. RFC 768 (Standard), August.

Miller Puckette. 1996. Pure data: another integrated computer music environment. In *in Proceedings, International Computer Music Conference*, pages 37–41.

Asbjørn Sæbø and U. Peter Svensson. 2006. A low-latency full-duplex audio over IP streamer. In *Proceedings of the Linux Audio Conference*, pages 25–31, Karlsruhe, Germany.

Flávio Luiz Schiavoni, Marcelo Queiroz, and Fernando Iazzetta. 2011. Medusa - a distributed sound environment. In *Proceedings of the Linux Audio Conference*, pages 149–156, Maynooth, Ireland.

David Robillard Steve Harris. 2008. Lv2 track. `http://lv2plug.in/trac/`.

Johannes M Zmölnig. 2001. Howto write an external for puredata. `http://pdstatic.iem.at/externals-HOWTO/`.