AoO - Protocol -towards message based audio systems

Author: Winfried Ritsch
Author: Christof Ressi
Date: march 2014 - february 2020
Version: 2.0-a1

The deployment of distributed audio systems in the context of computermusic and audio installation is explored within this library, to implement the vision of transition from static streaming audio networks to flexible dynamic audio networks. Audio data is send on demand only. Sharing sources and sinks allows us to create arbitrary audio networks, without boundaries.

This idea of message based audio systems, which has been investigated within several projects from playing Ambisonics spatial audio systems, streaming over Large Area Networks (LAN) and playing within a computermusic ensembles.

In a first implementation Open Sound Control (OSC) is used as the content format proposing a definition of "Audio over OSC" - AoO.

The paradigm, audio has to be synchronous data streams, has led to an audio infrastructure, which not always fit the needs. Audio messages, especially in computermusic can also be short messages like notes, and time bounded content. So each audio message is always a pair with a timestamp, the synchronization of distributed audio sources can be accomplished within the accuracy of network time protocols.

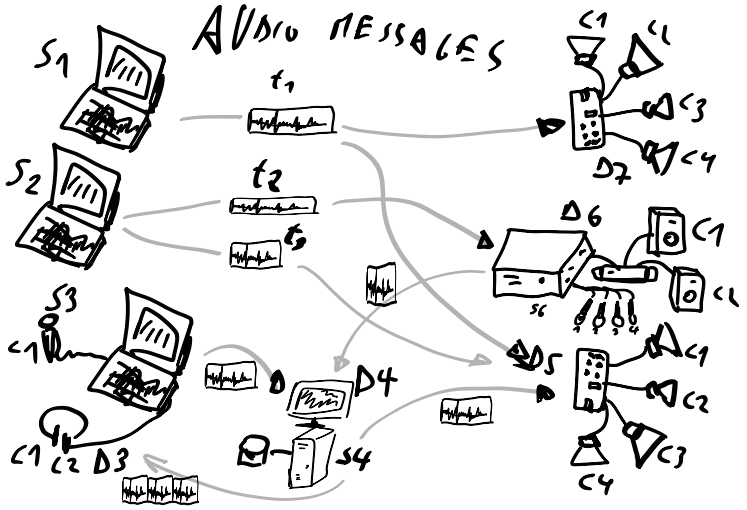

Figure 1: first idea of a message audio system with sources S_n and sinks D_n

Introduction

The first idea of a message based audio system came up with the requirement of playing a multi-speaker environment of distributed networked embedded devices from several computers, avoiding a central mixing desk.

Another demand for a message based audio network came up during the development of a flexible audio network within the ICE-ensemble [ICE]. A variable number of computermusic musicians sending time bounded audio material with their computers to other participants (for monitoring or collecting audio material), would have caused a complex audio-matrix setup of quasi-permanent network connections with all the negotiations and initialization for these streams. Not only because of the limited rehearsal time, this seems to be both too error prone and an overkill in terms of network load.

The structure of a functional audio-network for ICE, especially during improvising sessions, cannot always be foreseen and is therefore hard to implement as a static network. It is therefore important to be able to easily change the audio network during performance, as musicians come and leave (and reboot). On the other hand, the need for low latency, responsiveness and sufficient audio quality has to be respected even during the dynamic change of network connections. No strict requirements on sample-rates, sample-accurate synchronization and the use of unique audio formats should be made in such situations. It should be possible to freely add or remove audio related devices to/from the system without having to go through complicated setup of audio streams and without having to negotiate meta data between the participants. This should simplify the implementation of the particular nodes.

Of course, special care has to be taken when playing together in an ensemble. Factors like network overload, especially peaks, can lead to bad sound and feedbacks. On the other hand, we also find such situations when playing together in the analog world. In any case, the limits have to be explored during rehearsals.

Setting up continuous streams where audio data, including silence, is send continuously to all possible destinations is an overhead, that can easily touch the limits of available network bandwidth. But also can cause wasteful/costly implementations. If we can send audio from different sources to sinks (like speaker systems) only on demand, simplifies the setup. Also, reducing the needs for negotiation for establishing connections simplifies this task, and therefore stabilizes the setup.

The use of messages for the delivery of audio-signals in a network seems to contradict the usual implementation of real-time audio-processing implementations in digital audio workstations, where mostly continuous synchronized audio streams are used. If these audio messages are send repeatedly in such a way that they can be combined together in time, they can been seen as limited audio data streams and supersede continuous audio streams.

Also audio streamed from different sources should be added time synchronous, which means even if the have different transportation times, latencies they should be added at the sinks with their exact source time. This is essential to preserve time information and the jitter should be mostly eliminated to get exact timing between sources.

Summing up these demands, the overall vision is to implement a distributed audio network, where a variable amount of nodes act as sound sources and sound sinks (sinks). It should be possible to send audio messages from any source to any sink, from multiple sources simultaneously to a single sink, respectively broadcasting audio messages from one source to multiple sinks. Accordingly, the cross-linking between the audio components is arbitrary.

There should not be a “Before you stream audio, you first have to negotiate and connect with ...”, Instead, any participant should be able to just send their audio data to others when needed. The receivers should be able to decide how to handle the audio, depending if they can or want to use them.

Following features can be outlined:

- audio signal intercommunication between distributed audio systems
- arbitrary ad-hoc connections
- various audio formats and samplerates
- audio-data on demand only
- time synchronous adding of sources

The most common way of communication within local networks is Ethernet. Therefore “Audio over Ethernet“ has become a widely used technique. However, there is roughly only a single approach: Stream based audio transmission, representing the data as a continuous sequence. For audio messages as on-demand packet based streams[1] we found no usable implementation (2009). This lead to the design and implementation of a new audio transmission protocol for the demands shown before. As a first approach, an implementation in user space (on the application layer) without the need of special OS-drivers was intended. This can also be seen as the idea of “dynamic audio networks”.

AoO protocol

Looking for a modern, commonly used transmission format for messaging systems within the computermusic domain, we found “Open Sound Control” (OSC) [OSC]. With its flexible address pattern in URL-style and its implementation of high resolution time tags, OSC provides everything needed as a communication format [BPOSC]. OSC specifications points out that it does not require specific underlying transport protocol, but often uses Ethernet network. In our case this would be UDP in a first implementation but is not limited to these. TCP/IP as transport protocol can also be used, but would make some features obsolete and some more complicated, like the requirement for negotiations to initialize connections. Wolfgang Jäger implemented “Audio over OSC” (AoO) within a first project at the IEM [AoO] in targeting Version 1.0, which was never accomplished. This was used in tests and ”AUON“ (all under one net), a concert installation for network art[4]

the AoO protocol V2.0

The definition of AoO protocol was made with simplicity in mind, targeting also small devices like microcontrollers. Unlike Version~1, messages are not bundled, and meta-information is split in a format and a data message to reduce size. No `#bundle` means no `timestamp`. Since timestamping in OSC messages is done on send time within a `#bundle`, it does not help on synchronisation and resampling, since the message can be send somewhere in the range of the buffer time of the sender audio application. A new strategy was chosen see Timing section, calculating the resampling faktor to realtime and using this for excact timingm see also section Timing below.

AoO synthax:

```
notify sinks about format changes:

''/AoO/<sink>/format ,iiiisb <src> <salt> <nchannels> <samplerate> <blocksi


deliver audio data, large blocks are split across several frames:
''/AoO/<sink>/data ,iiidiiib <isrc> <salt> <seq> <samplerate> <channel_ons

from sink to source to request the format (e.g. the salt has changed):
''/AoO/<src>/request ,i <sink>''

from sink to source to request dropped packets; the arguments are pairs of
''/AoO/<src>/resend ,iib <sink> <salt> [<seq> <frame>]*''

ping message from sink to source (usually sent once per second):
''/AoO/<src>/ping ,i sink''
```

Parameter used:

```
''src''
  Identification number of the source

''sink''
  Identification number for the sink

''salt''
  Unique random number

''seq''
  sequence of sequent data blocks

''samplerate''
  Different sampling rates of sources are possible, which will be re-sample

''nchannels''
   number of channels to send

''channel_onset''
   first channel number in sink to write ''nchannels''

''blocksize''
   number of samples in a data block

''totalsize''
   total size of package

''nframes''
   number of frames to send
```

```
``frame``
   starting frame in block

``codec``
   which codec is used

``options``
   options for codec

``data``
   data content like defined above
```

Data packages used are uncompressed packets with data types defined by format. However, it's also possible to use blobs with an arbitrary bit-length audio data. This can become handy if bandwidth matters. Sources must be aware, which formats can be handled by the sinks. Using codecs the codec defines the data. At the moment besides raw data only opus [opus] is implemented, since it also supports low latency and to keep it simple, there should not be a need for others.

To provide low latency, time-bounded audio transmissions is sliced into shorter messages and send individually to be reconstructed at the receiver.

theory of operation

There must always be at least one format message before sending data messages to a specific sink, which can request one.

For the addressing the sinks the structure of the resources in the network is used as the base. Each device in the network with an unique network-address (IP-number and Port number) can have one or more sinks with different identification numbers. Each of these sinks can have one or more channels. There can be an arbitrary amount of sinks, and each sink could have an arbitrary amount of channels.

In OSC, there is a type of query operators called address pattern matching. These can be used to address multiple channels or sinks in one message. Since pattern matching can be computational intensive, we propose only to use the "*" wild-char for addressing all channels of a sink or all sinks of a device.

Integer for most parameter was chosen in favor for processors without hardware floating point support. Channel specific data information like the id number of the message stream, the sequence number in the channel message allow more easily to detect lost packages. The resolution of a sample and an individual resampling factor is contained in the channel messages, where the resampling factor enables channels to differ from the samplerate specified in the format message, allowing lower rates for sub channels, control streams or higher rates for specific other needs.

For re-arranging the audio packages there is a need to do some sort of labeling of the messages, since it is not clear if they are intended to overlap or are different material. This is handled via the "identification number" (id) and salt. Identical identification numbers means to recognize the material as one material and they can be cross-faded. So these numbers has to has to unique at least at the sink. Salt means different Audio Messages even on the same id.

addressing scheme

Like described above, to deliver audio messages to a sink, additionally to the sink number and channel number, the address of the device has to be known. A decision was made, that the address is not part of the message, since the sender has to know about the sink on the receiver and the network system has to handle the addressing.

Like stated in in the vision, we do want negotiations and requests, but in situations where IPs are unknown, we needed a mechanism to grasp it. One implementation was announcement message broadcasted by each sink, with the address and a human readable meaningful name. Even more polite we implemented them as invitation messages, which also states: ”ready to receive“.

A second problem arose, since broadcasting to all sinks with the same number, the destination information is not contained in the audio message, we cannot use broadcast to reduce network load and address specific destinations. For this the sink has to know about the sources it will accept. Anyway this worked fine, but made some additional efforts in communication before.

One other problem is if drains or sinks are behind a firewall. So if `A` is behind the firewall, `B` cannot send data to `A` directly. So a receiver can use the back-channel of the receiver, which normally is provided using TCP/IP protocol, but not usinge the UDP protocol we do, but we can grasp it, when a message from the source arrives. But since a normal "NATing" firewall stores session data, there is a chance that it can work when the sink uses the known sources. This has to be explored further and individually before usage and since network setups differ, do not assume it works everywhere.

mixing

In the first implementation we used two different modes: Mode 1 provides the possibility of summation of the received audio signals and Mode 2 should perform an arithmetic averaging of parallel signals. The reason for this is that summing audio signals with maximum amplitudes each causes distortion. Using Mode 2 this cannot happen.

In the Version-2 of AoO only Mode 1 is implemented, since samples are added mostly within a floating point domain, or a with integer with more bits than the samples, and the audio application should take care to reduce the volume as needed. So volume changes are not triggered by additionally sources.

timing and sample-rates

Timing is critical in audio-systems, not only for synchronizing audio, but also to prevent jitter noise. Times in the internet are represented by a 64 bit fixed point number, like timestamps specified by OSC, to a precision of about 230 picoseconds. This conforms to the representation used by the Network Time Protocol NTP [RFC5905].

Also other time-protocols can be used like the Precision Time Protocol PTP, since this is handled by the system, we use the exact time information of the system, so care should be take, that the devices are synchronized over network.

Using fixed buffering mode, the buffer size has to be chosen large enough to prevent dropouts. In the automatic buffer control mode, the sink should use the shortest possible size for buffering. If packets arrive too late, buffering should be dynamically extended and then slowly reduced. This has to be handled by the audio application.

Number of dropouts, ping times and a method for resend is provided to be used for this purpose.

Also using a lot of channels and large block sizes, the can be larger than packetsizes. So a mechanism had been implemented to slice them. The smaller the packets, the more chance they have to travel over many hops, since each router can limit sizes and drop large packets.

Since audio packets can arrive with different sample-rates, re-sampling is executed before the audio data is added to the internal sound stream synchronized with the local audio environment. This provides the opportunity to synchronize audio content respecting the timing differences and time drifts between sources and sinks. This strategy of resampling is shown in a figure *re-sampling*:

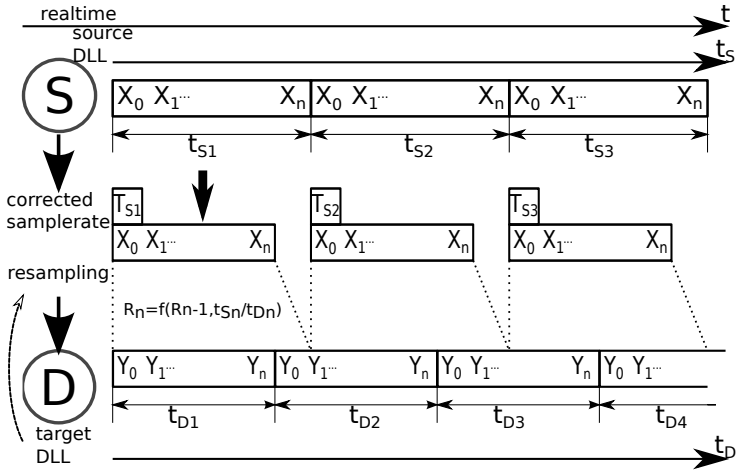

Figure 2: re-sampling rate R_n between source S and sink D is not constant

Looking at synchronization in digital audio system, mostly a common master-clock is used for all devices. Since each device has its own audio environment, which may not support external synchronization sources, the time T_Sn of the local audio environment is used to calculate the corrected samplerate for outgoing audio messages.

Using the incoming corrected samplerates from the remote source, we can compare them with the local time t_Dn and correct the re-sampling factor R_n dynamically for each message. The change of the correction should be small if averaged over a longer time, but can be bad for first audio messages received, since a DLL filter is used, like described in the paper "Using a DLL to filter time" by Fons Adriaensen [FA05] .

Since the local time source of a device can differ from the timing of the audio environment, each device needs a correction factor between this time source and the audio hardware time including the time master device. This factor has to be communicated between the devices, so the re-sampling correction factor can be calculated before the first audio message is sent, guaranteeing a quasi sample-synchronous network-wide system starting with the first message send.

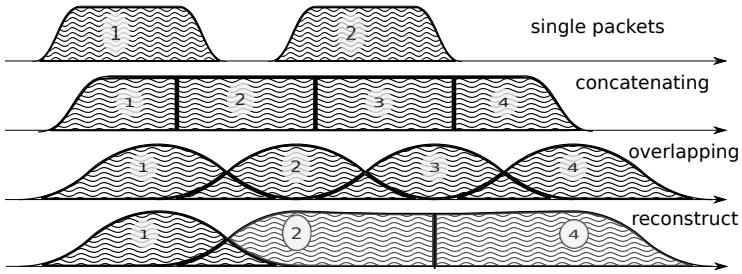

Figure 3: audio messages are arranged as single, combined or overlapped using different salts.

Also the idea that all audio messages, which are originated at the same time are mixed in correctly in the receiving buffer has been dropped from Version 1.0 to Version 2.0. However it can be accomplished using more receiver and using the information by the ping requests to the sources, which deliver the exact network delay and implement delays by the receiving application. This gives more flexibility for different use cases.

Networking

Networking is not part of the library, but the function of the streaming depends on it, so some helpers has been added to accomplish different network tasks.

In the stream project all sources and sinks has been in one big network with known addresses and without firewalls, since every device is its own router and firewall using olrd, the Optimized Link State Routing Protocol (OLSR)[1] is an IP routing protocol optimized for ad hoc networks used by freifunk or funkfeuer [0xFF] with free bandwith usage.

But the project playing together with an ensemble showed, that most user work behind a firewall. One solution is to setup a virtual private network VPN or using port forwarding over secure shell, both needing a server with an offial known IP.

Another solution have been suggested by IOhannes Zmoelnig and needs to be tested, but implementing the invitation message gives us the chance to gather all informations needed for this strageties.

peer

setup

```
client A
    will use listening port 10001
    private IP: 192.168.1.100
    public IP: 192.0.2.0.72
client B
    will use listening port 10002
    private IP: 192.168.7.22
    public IP: 198.51.100.190
```

initiate session:

```
clientA: initiate connection [connect 198.51.100.190 10002 10001( -> [netse
```

```
clientA: send some data
    the data won't arrive on clientB yet
    but the NATting routerA sets up the forwarding rules
clientB: initiate connection [connect 192.0.2.0.72 10001 10002( -> [netsend
clientB: send some data
    data should arrive at clientA (2nd outlet of [netsend])
clientA: send more data
    data should arrive at clientB (2nd outlet of [netsend])
```

A hole-punching server

setup::

serverX reachable via a public IP:port

clientA, clientB live in (separate) private (NATted) networks don't know the public IPs

network connection flow:

```
1 clientA sends <channel-token> <clientA-name> <portA> to serverX
    : some string known to all peers (e.g. "covid19")
    - <clientA-name>: some string identifying clientA
    - <portA>: the port where clientA listens for incoming payload data
2 serverX notes the public IP of clientA and remembers it along with the da
3 clientB sends <channel-token> <clientB-name> <portB> to serverX
4 serverX notes the public IP of clientB and remembers it along with the da
5 serverX sends the public IP:port of clientB to clientA
6 serverX sends the public IP:port of clientA to clientB
    in practice serverX might just "broadcast" the entire stored informatio
7 clientA opens a UDP-connection to the public IP:port of clientB
8 clientB opens a UDP-connection to the public IP:port of clientA
9 tada
```

Note: has to be tested, if failed this documenation part will be removed.

Implementations

As a first proof of concept, AoO was implemented within user space using Pure Data. [Pd96] Also V2.0 has been implemented first with Pd externals, but others will follow since it is done as a C++/C library usable for other computermusic languages, plugins or mircocontrollers.

C++/C library

The main functionality is implemented in this library, which is used by the further implemantation for applications described below.

See source and library documenation for details.

Puredata library

The V1.0 implementation has shown various problems to be solved in future. Using the network library iemnet additional ”externals“ have been written in C to extend the OSC-Protocol, split continuous audio signals into packets and mix OSC audio messages in sinks.

In the new Version-2 the network infrastructure has been implemented within the AoO library to overcome these problems and use new concepts for threading, to avoid blocking the main task.

As a first test environment, a number of different open-source audio hardware implementations, using Debian Linux OS-System, has been used. The new Version was implemented for most OS-System as Pd-Externals in a first place.

The new version can be found in the git library and also should be available via Pd library manager deken:

- see http://git.iem.at/cm/aoo

see documentation in the help and testfiles there or in a reference projects in use cases.

About Document

Thanks all who helped to bring this to live and please test und comment, file issues and pull request to improve it at http://git.iem.at/cm/aoo

authors: Winfried Ritsch, Christof Ressi

date: march 2014 - february 2020

version: 2.0-a1

[1] not to be mistaken with ”streaming on demand” or UDP packets

[4] performed 17.1.2010 in Medienkunstlabor Kunsthaus Graz see http://medienkunstlabor.at/projects/blender/ArtsBirthday17012010/index.html

[OSC] Matt Wright, http://opensoundcontrol.org/spec-1_0 , [Online; accessed 1-Feb-2014], "The open sound control 1.0 specification.", 2002

[BPOSC] Andrew Schmeder and Adrian Freed and David Wessel, "Best Practices for Open Sound Control", "Linux Audio Conference", 01/05/2010, Utrecht, NL

[AoO] Wolgang Jaeger and Winfried Ritsch, "AoO", https://iem.kug.ac.at/en/projects/workspace/2009/audio-over-internet-using-osc.html , [Online; accessed 12-Dez-2011], Graz, 2009

[RFC5905] D. Mills and J. Martin and J. Burbank and W. Kasch, "RFC 5905 (Proposed Standard)", "Network Time Protocol Version 4: Protocol and Algorithms Specification" , published by "Internet Engineering Task Force" IETF, "Request for Comments", number 5905, http://www.ietf.org/rfc/rfc5905.txt , june 2010

[FA05] Fons Adriaensen, "Using a DLL to filter time", 2005, https://kokkinizita.linuxaudio.org/papers/usingdll.pdf

[Pd96] Miller S. Puckette, "Pure Data", in "Proceedings, International Computer Music Conference." p.224–227, San Francisco, 1996

[opus] http://opus-codec.org/

[OLSR] IETF RFC3626 - https://tools.ietf.org/html/rfc3626

[0xFF] Funkfeuer Graz -http://graz.funkfeuer.at/