University of Music and Performing Arts Graz
Institute of Electronic Music and Acoustics

Project report

Audio over OSC

Wolfgang Jäger

supervision

Winfried Ritsch

January 27, 2010

Audio signals should be distributed to playback systems over network. To save resources like bandwidth and computing time, the audio data should be sent on demand only. Ethernet established itself quite well in the field of controlling embedded devices and computer, therefore it will be used here. Furthermore most of the available network components are able to handle the IP protocol. On the application layer Open Sound Control (OSC) will be used, due to its message based structure. Unfortunately there is no standardized implementation for Audio over OSC yet. So with regard to the possibility of synchronicity and partly discontinuous data streams with constant and predictable latency a protocol to transmit Audio over OSC has to be developed.

Contents

1 Requirements

In our principle setting we have distributed audio systems, where a variable amount act as sound sources and a variable amount act as sound drains. It should be possible to connect any source to any drain, furthermore multiple connections to one drain, respectively multiple connections from one source, should be possible. Accordingly the crosslinking between the audio components is arbitrary.
Looking at an exemplary setting in illustration 1 the grouping of particular drains should follow a logical principle, independent of the hardware circumstances. For example: Drain number four is specified between the upper left computer and the upper right microcontroller, independent of its separated hardware components.
The transmission of the audio data has to be in an uncompressed format, due to low latency requirement up to real-time capability of the prospective protocol.
We want to be able to use given hardware environment, like an Ethernet network, therefore it is necessary to be independent of the physical conditions. Looking at the OSI model this means, that we want to settle our protocol at the top, onto the application layer, where no dependencies to any transmission parameters should appear.
A further limitative feature is the availability of bandwidth at the interaction between the audio systems. Since our approach is for low budget systems, for systems with lower hardware requirements and therefore with lower power consumption, the amount of available network capacity is limited. For that reason it is decisive that the audio data is sent on demand only.

Totalizing the requirements:

- Audio signal intercommunication between distributed audio systems
- Arbitrary connections
- Uncompressed transmission
- Separation of transmission and application layer
- Data on demand only

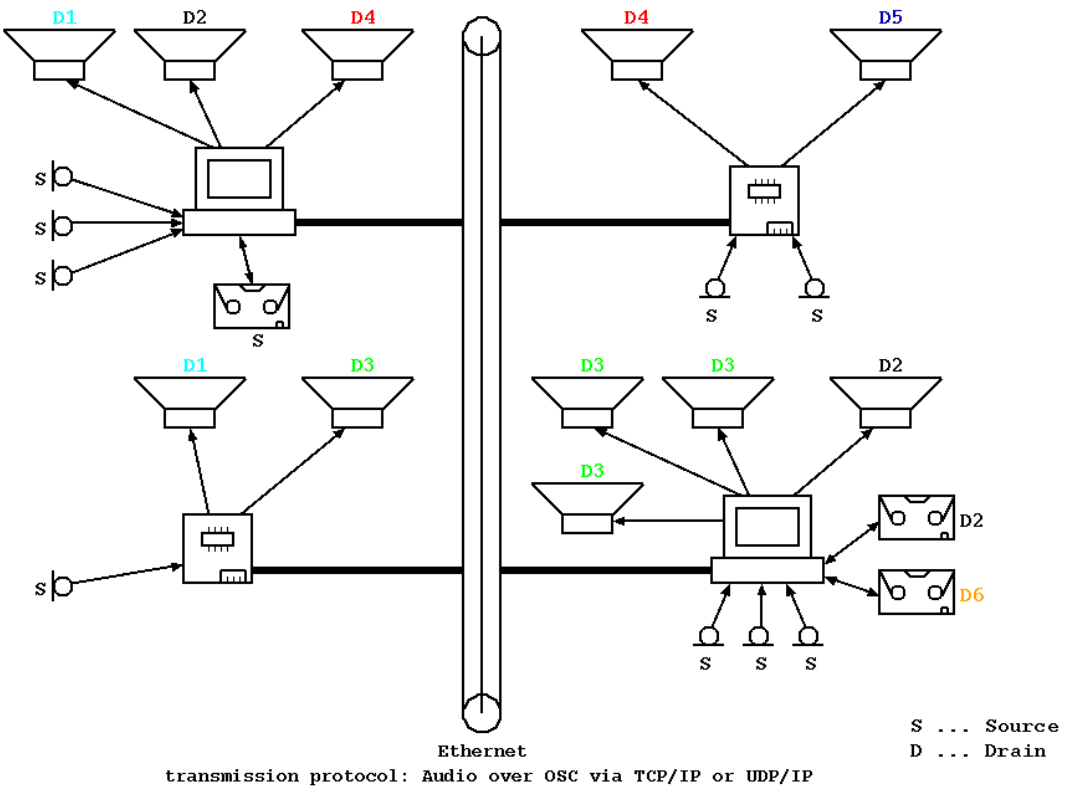

Figure 1: Exemplary setting

2 Audio over Ethernet

The most common way of communication within local networks is Ethernet. Therefore we want to take a look at Ethernet based solutions concerning digital audio. It is possible to roughly classify two sorts of approaches: Stream based and packet based solutions.

Stream based audio solutions represent the data as a continuous sequence of datasets (Illustration 2). According to the requirement of data on demand only a packet based transmission protocol is the solution.

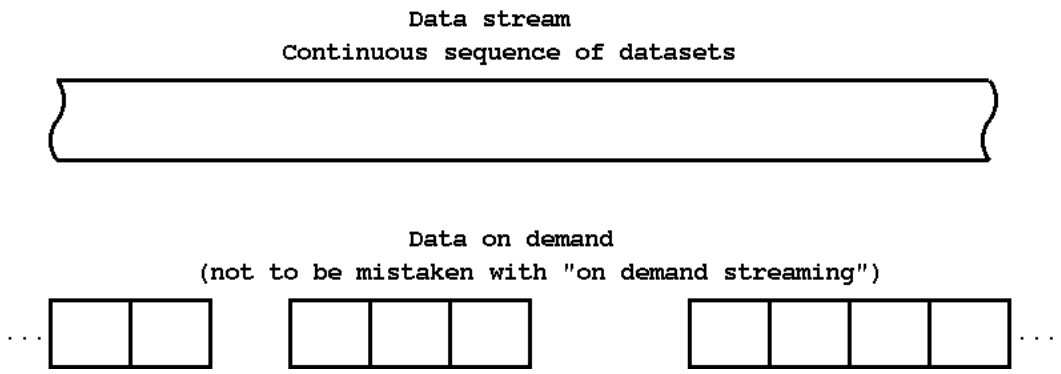

Figure 2: Stream vs. packet based audio solutions

Again there are a couple of different implementations of packet based audio protocols available. Lets take a look:

	Products	Networking Tools
Group 1	Ethersound by Digigram NetCIRA by Fostex REAC by Roland	Network Hub
Group 2	AVB by IEEE AVB Task Group CobraNet by Peak Audio	Network Switch
Group 3	DANTE by Audinate JackTrip by CCRMA OpenSoundControl (OSC)	Router, Gateway

Table 1: Audio over Ethernet solutions on different layers [10]

The first column specifies the membership of the protocols. They can be classified after the layer they are operating at. Group 1 operates at the physical layer of the Ethernet specification, it uses the hardware infrastructure but doesn't implement the frame structure of the Ethernet protocol. Compared to the OSI model it equates the implementation of layer 1 - physical layer. At this low level, no information about the routing within the network can be evaluated. Therefore only signal repeaters, like network hubs are, can be used. Typical representatives of this group are the protocol Ethersound [5] and the protocol REAC [9]. NetCIRA [6] is an implementation of the

Ethersound protocol.
Group 2 is based one layer above group 1. Additionally to the physical Ethernet structure, the Ethernet frame is implemented. Compared to the OSI model this would conform with layer 2 - data link layer. Due to networking tools like network switches, which evaluate the destination MAC address within the Ethernet frame, this group is bound to Local Area Networks. Audio Video Bridging (AVB), being currently under development by a IEEE task group [2] and CobraNet developed by Peak Audio [4] are representatives of this group.
Referring to the our considerations of being independent of the physical conditions in our network, this two groups are of no interest to our approach.
Therefor we have to take a look at group 3 of the table. Group 3 operates at the application layer of the OSI model (layer 7). Protocols like Ethernet, IP, UDP and TCP are subjacent and we are finally independent. This also means that the availability of the data is not bound to any local network. A proprietary representative is DANTE [1]. JackTrip is developed at the University of Stanford [3] and is based on the sound server JACK. One further possibility within this group is represented by OpenSoundControl (OSC) [11].

2.1 Open Sound Control

Open Sound Control (OSC) is a widely accepted standard for message transmission in the field of computer musics. It has been developed at the University of Berkeley, California at the Center for New Music and Audio Technologies (CNMAT).
With its flexible address pattern in URL-style, physically independency and the possibility of high resolution time tags it provides a great opportunity, with the only disadvantage of the absence for packing audio data in a defined modality. Due to the factor that OSC is message based, our requirement according data on demand only is also fulfilled. Hence finding a definition packing audio data inside OSC messages should be the goal of this paper. [11]

3 The "Audio over OSC" protocol

Based on the OSC protocol a transmission protocol will be developed, which allows transporting audio data over the OSC definition. The OSC standard does not require a specific underlying protocol. Anyway in our case this would be UDP/IP or TCP/IP. One step further down in the OSI layer model the Ethernet protocol is specified.
OSC is organised in packets. A packet contains messages and/or bundles. A message includes an address pattern, an OSC Type Tag and the OSC arguments (representing the actual data). Messages may be a part of a bundle, which groups single messages and attaches an OSC Time Tag.

3.1 Addressing

In a first step the hierarchy of the protocol has to be specified. There are drains and those drains are subdivided into channels. There could be an arbitrary amount of drains and each drain could have an arbitrary amount of channels.
The addressing of a device could for example look as follows */AOO/drain/26/channel/3*, where the third channel of drain number 26 is sent, respectively the audio data followed by the address pattern is meant to be for channel three of drain number 26. */AOO* is the protocol specific prefix. The principal structure of the address pattern follows the specification of the OSC protocol.

3.2 Sampling rate

The value of the sampling rate has to be transmitted within every package of the protocol. According to the used hardware, different sampling rates are possible.

3.3 Blocksize

The amount of samples in each package of audio data has to be specified here.

3.4 Overlapping factor

The overlapping factor is by default one. This means, that block after block is sent, without overlapping. An overlapping factor of two indicates, that there is a redundancy of two, half of the block equals half of the block being sent before, the other

half equals the first half of the following block (see also illustration 3). An overlapping factor of 1/2 would indicate, that only every second block is transmitted.

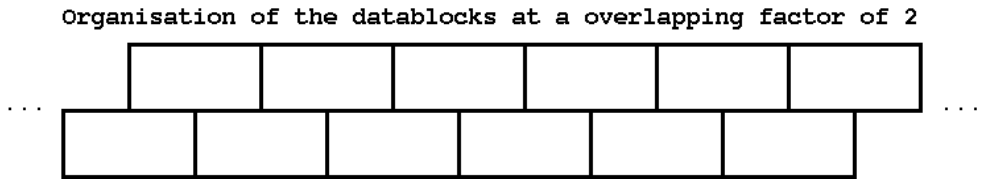

Figure 3: Overlapping of the data blocks

3.5 Resampling factor

The resampling factor is specified for each channel. It is linked to the sampling rate in order to be able to choose the precision of each channel individually (e.g. if the resampling factor is 2, this would indicate, that at an overall sampling rate of $44100Hz$, the sampling rate of the specific channel is $88200Hz$). The value has always to be at the power of two.

3.6 MIME type

As you never know if the field of applications where the protocol is used expands, there should be provided a possibility to classify the type of coding of the audio data. Therefore we may use the so called Multipurpose Internet Mail Extensions (MIME) standard. MIME Types label the data in form of type and subtype, separated by a slash [7]. In our uncompressed format, the MIME type would be for example "audio/pcm", whereas "audio/CELP" classifies CELP coded data.

3.7 Bandwidth considerations

For better evaluation concerning the bandwidth of our defined Audio over OSC (AOO) protocol let us look at some calculations. There are different options for Ethernet transmission rates. For small embedded system boards a transmission rate of $10Mbit/s$ is still widely spread. Faster systems may use $100Mbit/s$ or even $1Gbit/s$.

Transmission rate	Resolution	f_S	Channels
$10Mbit/s$	$16bit$	$44100Hz$	14
$100Mbit/s$	$16bit$	$44100Hz$	148
$1Gbit/s$	$16bit$	$44100Hz$	1521

Table 2: Number of channels for different transmission rates

If we want to use for example a sampling frequency of $44100Hz$ with an resolution of $16bit$ per sample, the amount of sendable channels would be limited to 14 channels. With the overhead of the underlying protocols like Ethernet, TCP/IP, UDP/IP and the overhead of the own OSC protocol, approximately thirteen to fourteen channels would be possible to send.
Illustration 4 shows how the limited bandwidth could be used in the most efficient way. If a relatively fast computer with a possible transmission rate of $100Mbit/s$ serves as sound source and the core of the network is a $100Mbit/s$ switch, each embedded device could be provided with the maximum amount of fourteen channels. If the sound source or the core element in the network would be exchanged against slower devices the fourteen channels would be the maximum for the whole network. This example shows, that a flexible protocol which allows variations in the sampling rate and variations in the resolution per sample, is needed.

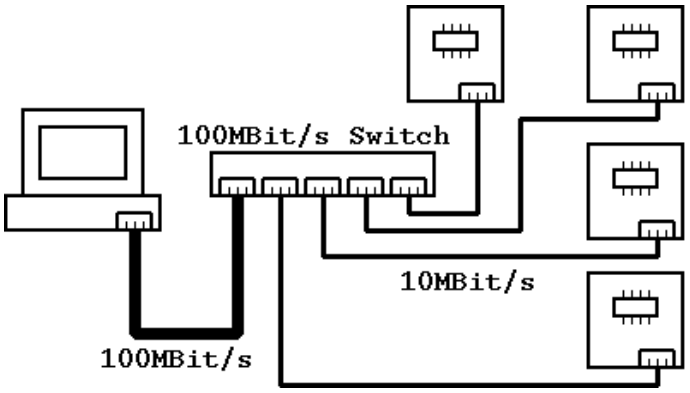

Figure 4: Possible distribution of the transmission rate in a network with embedded devices

3.8 Dynamic range considerations

Talking about the variability of the resolution: If the ambient noise level in a public place equals about $40 - 50dBA$ and we want to operate a loudspeaker in this ambience with a audio signal resolution of $12bit$, the resulting dynamic range of $6.02 \cdot 12bit = 72.24dBA$ would still sum up to an adequate maximal level of about $112dBA$. Therefore the AOO protocol should provide a possibility of resolution conversion.

3.9 Data types

There are some fundamental data types specified in the OSC standard (Table 3).

int32	32-bit big-endian two's complement integer
float32	32-bit big-endian IEEE 754 floating point number
OSC-string	A sequence of non-null ASCII characters followed by a null, followed by 0-3 additional null characters to make the total number of bits a multiple of 32.
OSC-blob	An int32 size count, followed by that many 8-bit bytes of arbitrary binary data, followed by 0-3 additional zero bytes to make the total number of bits a multiple of 32.

Table 3: Fundamental data types [11]

Parameters like sampling rate and blocksize are transmitted as int32 data types, this isn't possible for the audio data, if the transmission should be variable in the resolution and space saving. Therefore the data type "blob" is needed as it is for arbitrary use.
We use this data type in the following form: The first argument is like defined the size count, followed by the actual audio data that is rearranged in dependency to the chosen resolution. If the specified blocksize and resolution don't result in a size, that is a multiple of $32bit$, additional zero padding is performed (for rearranging details look at illustration 11).

3.10 Request

To offer a possibility to ask for the parameters of the drains present in the network, a so called "request" is established. The form of the address pattern for a request is */AOO/request*. Additionally to the already known parameters a name for each drain has to be established, to guarantee a pleasant handling for humans. Normally the request would be sent into the network as a broadcast. Drains which are available should answer to a request and transmit the following information (in the below-mentioned order):

- Drain number
- Sampling rate
- Blocksize
- Overlapping factor
- MIME type
- Name of the drain
- Number of available channels
- Resampling factor of each channel

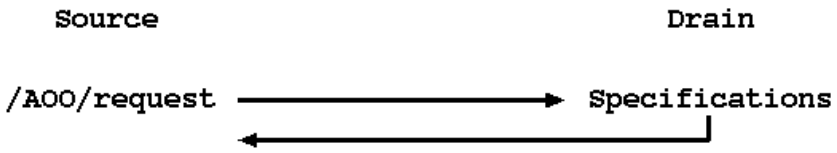

Figure 5: Request carried out by a source

3.11 Mixing

If two or more audio signals are sent to one drain, mixing has to be done. There are two different modes like this could happen. Mode 1 should carry out a arithmetic averaging and mode 2 should provide the possibility of a simple summation of the received audio signals. As audio data is organised in packets, there has to be done

some sort of labelling of together belonging data. This is achieved by the insertion of a so called “identification number (ID)”.

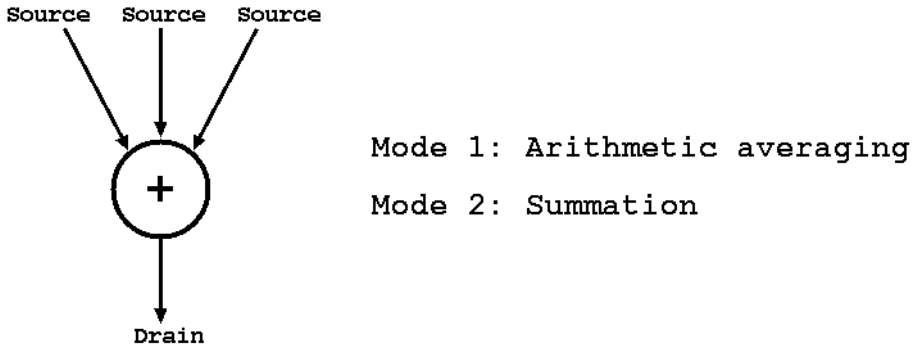

Figure 6: Mixing of multiple channels at one drain

3.12 Timing

Like it is prepared in the OSC standard, a bundle with an OSC Time Tag will be used.

Time Tags are represented by a 64 bit fixed point number. The first 32 bits specify the number of seconds since midnight on January 1, 1900, and the last 32 bits specify fractional parts of a second to a precision of about 230 picoseconds. This equals the representation used by Network Time Protocol (NTP) [11].

Since OSC implements an equal time standard as NTP, but does not provide any mechanism for clock synchronization, the synchronization, which has to be carried out externally in this case, will be performed by NTP. Therefore we have to run a NTP server on the sound devices. Looking at illustration 7 will explain about its functionallity.

The particular machine is connected to different time servers with variational accuracies. The overall accuracy of the servers is marked with the first character within the display. An asterisk identifies the best time reference, a plus sign classifies a good time reference, whereas an minus sign indicates a not that good time tag. The offset to the time reference is displayed in milliseconds and the local time is adapted successively to this reference time (successively, to prevent time jumps, which may lead to data inconsistence). The displayed jitter (again in milliseconds) is this part of the time error, which we are not able to eliminate.

```
     remote           refid      st t when poll reach   delay   offset  jitter
==============================================================================
+129.27.2.3      192.53.103.108   2 u  121  256  377   16.988   -0.766   1.948
*193.171.23.163  .PPS.            1 u   38  512  377   12.766    1.485   0.215
-85.125.223.113  193.171.23.164   2 u  110  256  377   16.057    4.072   0.260
+80.121.209.38   94.249.153.212   3 u  120  256  377   17.871    1.407   0.560
```

Figure 7: NTP server information

As explained on the NTP projects web appearance, an accuracy down to a few milliseconds is possible within a local area network and an accuracy down to a few tens of milliseconds is possible in the global internet [8].
Working with audio signals on different play back devices, a delay of a few milliseconds between signals affects the spatiality, but is not fatal.

By anticipant I want to talk about the more complex case, when Pure Data comes into the game. Pure Data itself works with an internal "logical" time. This logical time is hard synchronized with the sound card of the machine it is working at.
As you can see, we are working with two different time bases here. The main issue is to find the best possible solution, concerning the interaction between those time bases. Therefore an synchronisation between the "logical" time and the "real" time has to be found. Pure Data is working with a buffer, where data is written, that is going to be released through the sound card. This buffer is circular shaped. When we link one particular place of this buffer to the "real" time (for example the first bit or the last bit of the buffer) and always interpret the distance from the buffer index to this particular bit, there would be a synchronisation between "logical" and "real" time. The discussed topic is accompanied by illustration 8.

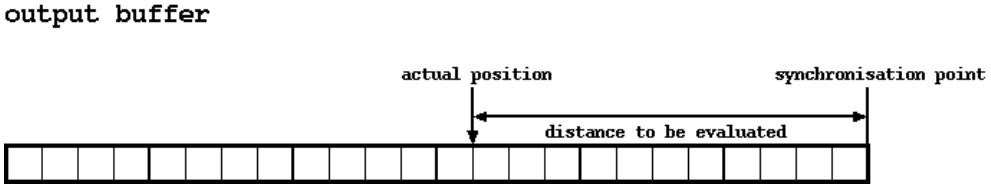

Figure 8: Synchronisation between "real" and "logical" time

In principle the above discussed suggestion should work, with the only problem that there is no implemented function within Pure Data, that delivers the position

of the actual pointer position in the output buffer of the sound card. Therefore a workaround has to be established. This leads to a solution containing an unique Identification Number (ID) and Sequence Number (SQ) for each audio signal and each audio data packet. The Identification Number specifies the packets belonging together and the Sequence Number increases with every packet and gives information if there are missing packets. If two IDs appear at one destination channel, mixing according to chapter 6 has to be done.

3.13 Final structure of the transmission protocol

Concerning signal processing, the following parameters have to be transmitted: The sampling rate of the source (in my case a default blocksize of 44100Hz will be used), the blocksize of the sent packages (I use a default blocksize of 64 samples), the overlapping factor of the packages (default value one), a MIME type classification (audio/pcm), a resampling factor for each channel, a resolution value for each channel, for rearranging the received data by the drain a sequence number (SQ) is specified for each channel and an identification number (ID) is also specified within each channel.
The parameters samplingrate, blocksize, overlapping factor and MIME type are only sent once in each bundle, whereas the identification number, the sequence number, the resampling factor and the resolution are transmitted within the data of every channel. The identification number is defined as first argument, the sequence number is defined as second argument, the resampling factor as third argument and the resolution of the sent audio data is defined as fourth argument within the audio data message. The audio data is sent as an OSC-blob, which allows an arbitrary arrangement of the transmitted bits. Figure 9 explains the protocol in a visual hopefully less confusing way.

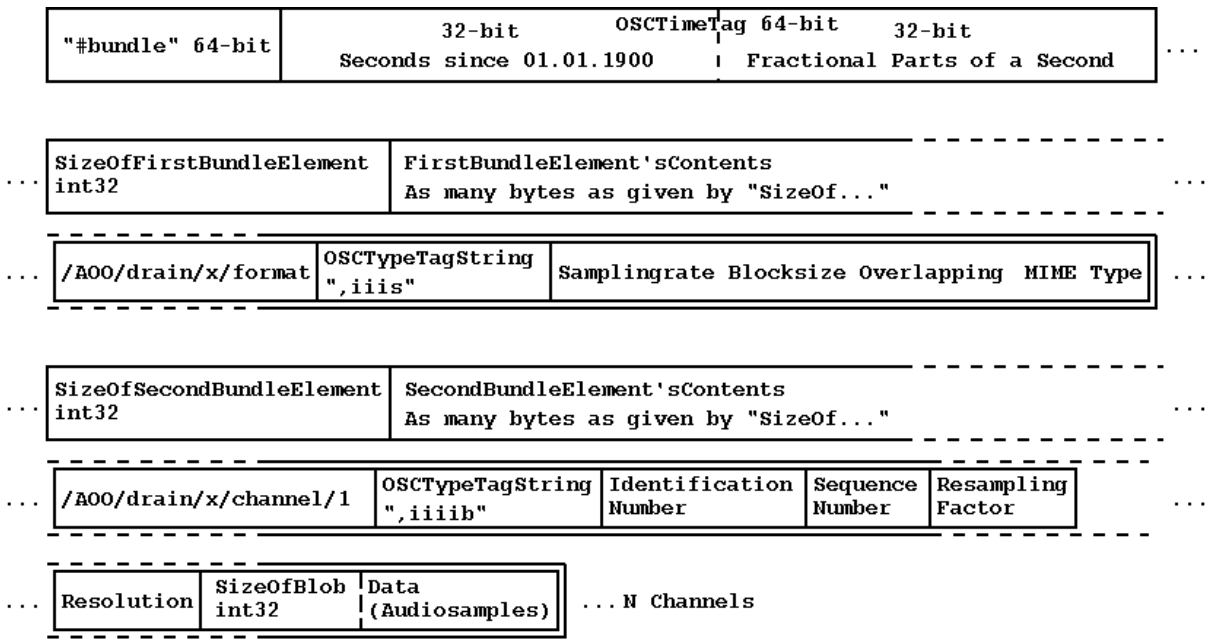

Figure 9: Structure of the protocol Audio over OSC

4 Proof of concept

As the structure of the protocol had been fixed, the next step was to implement a proof of concept environment. Therefore I used Pure Data (pd). I structured the implementation like it would be in a hardware scenario. There are sound sources, and sound drains.

In the middle is a concatenation of special objects, which pack the raw audio data in the desired form. The main object are: pack~, blob, packOSC, udpsend on the senders side and udpreceive, unpackOSC, pipelist, routeOSC, unblob and funpack~ at the receivers side. Some of the objects have been available as externals in pd (pack~, udpsend, udpreceive, routeOSC, pipelist), some have to be modified (packOSC, unpackOSC) and some have to be written (blob, unblob, funpack~).

What has to be said to each object?

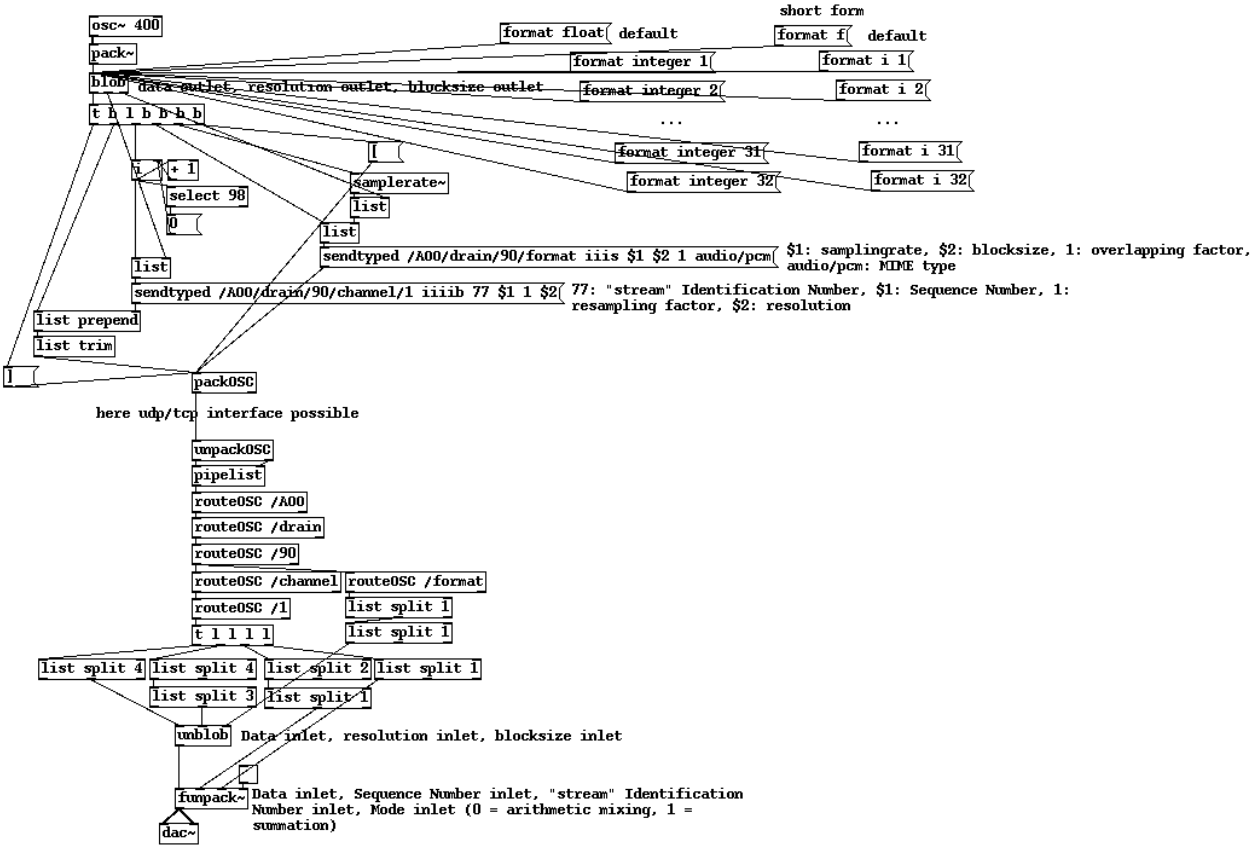

Figure 10: Signal flow diagram

4.1 pack~

packOSC works on the message layer of pd. Therefore the audio data has to be converted from the signal layer to the message layer, which is accomplished by this object. The package is written by Johannes Zmölnig.

4.2 blob

The OSC standard provides an arbitrary data type, called blob. It is an int32 size count, followed by that many 8-bit bytes of arbitrary binary data, followed by 0-3 additional zero bytes to make the total number of bits a multiple of 32 [11].
Unfortunately packOSC hasn't provided a possibility to handle blobs in the first place. Neither is it possible to pack the prepared audio data easily. So the object "blob" has to be designed and a possiblity to handle blobs within packOSC, has to be created. For the latter case I got into correspondence with Martin Peach, who wrote the object packOSC.
In order to save bandwidth, I want to choose the resolution of the audio data and maybe the sampling frequency of single channels freely. So in the messages where the audio data is located, I transmit as first argument a identification number, followed by a sequence number, followed by a resampling factor, followed by the resolution of the audio data. These values have the Type Tag "integer". As last argument the message contains a blob with the actual audio data. The audio data is arranged as space saving as possible within the blob. This is achieved by shifting the bits into the right order. After rearranging the bits and if necessary perform zero padding to have a size that is a multiple of 32 bits, the message is sent to the modified packOSC object, where the obligatory size count is carried out.

4.3 packOSC

The packOSC object forms manageable packages. The contents have to be arranged and prepared for transmission. Lets look at the options in detail. At first a bundle is opened. With the opening of a bundle the so called Time Tag comes along. Time Tags are represented by a 64 bit fixed point number. The first 32 bits specify the number of seconds since midnight on January 1, 1900, and the last 32 bits specify fractional parts of a second to a precision of about 230 picoseconds [11].
After the Time Tag, the size of the first bundles element is transmitted, followed by

the content of the first element of the bundle. The content of a bundle could be again a bundle or as it is in our implementation, it could also be an OSC message. The message contains an address pattern, followed by the Type Tag of the message, followed by the actual data of the message, which is in our case for the first message in a bundle, the format parameters of our audio data. After the first bundle element, x-elements follow. Where "x" is the number of channels, which will be transmitted. Those x-elements also contain a size count and an OSC message, which itself contains an address pattern, a Type Tag and the actual audio data of each channel. Finally the whole bundle is closed and sent to the next object.
Using the object packOSC, the first problem I came in touch with had been, that it wasn't possible to send special Type Tags of the OSC protocol alone in one message. If you specify the message as "sendtyped" it is up to the user to care for the Type Tags. The problem affected the Type Tags T (TRUE), F (FALSE), I (INFINITUM), N (NIL) where no data is allocated in the data argument. After getting in touch with the mailing lists of the pd community, where bugs could be reported, the problem got solved.

4.4 udpsend, udpreceive

In my implementation the transport protocol is UDP. The object udpsend receives the prepared data from the packOSC object and transmits it to the chosen receiver in the network. Therefore an IP address and a port number are required. On the receivers side the object udpreceive is located. For this object the port number, where it has to listen to, has to be specified.

4.5 unpackOSC, pipelist

The first element at the receiver side, after the transport layer is left, is the unpackOSC object. It receives the data from the object udpreceive and decodes its content. The bundle is dissolved and the single messages are released. On a second outlet the time delay of each bundle is evaluated. This outlet serves as informant for the pipelist object, which delays the messages as desired. The delay could be adjusted at the senders side by adding the desired delay to the actual Time Tag.
During the implementation of the transmission tools, it has been up for debate, that the delay between Time Tags may be used for sequencing the received packages.

This would be possible as the principle time delay between two packages would be $(samplingrate/samplesperblock)^{-1}$. Therefore I expanded the object unpackOSC about one further outlet, which reveals the Time Tag of each bundle. However it has to be recognised, that this time delay is inaccurate. This is caused by the fact, that the particular Time tag is produced at the moment when the bundle is opened at the senders side. It is not assured, that the bundle is closed within a fixed time duration, this depends on the actual workload of the machine where the package is generated. Neither is this time synchronized in any way to the "logical time" of the senders audio buffer (for "real time" and "logical time" details look at chapter 3.12).

4.6 routeOSC

The optionally delayed messages without bundles, are left and need to be routed to their destination. This is carried out by the routeOSC object. In dependency of the address pattern the messages are routed to different destinations.

4.7 unblob

Reaching their target, the data of the message is left. Within this data block the space saving arranged audio data is packed. The resolution of the data could be found as fourth integer value in the actual message. According to this information the bits have to be reorganised and brought back into a form that Pure Data could handle.

4.8 funpack~

The last object before the audio data could be played back, is the object funpack~. There would be an already implemented object called unpack~, but as I want to handle some errors, which occur during the transmission, I had to write my own object on the basis of the unpack~ object (which is written by Johannes Zmölnig).
At the beginning of the signal chain, there has been the object pack~. It transformed the data from the signal layer of pd to the message layer. funpack~ is the counterpart of this object. It recycles the data from the message layer to the signal layer, so that it could be processed and replayed. The recycling goes along with buffering.
As it is never assured, that a UDP package arrives at its destination, the received data is stored in a buffer and up to a certain boundary the error could be balanced. If there are too many missing packages (could be caused by a network overload), what

can be deduced by the sequence number (is provided over an inlet), the funpack~ object tries to keep the periodicity and interpolates between the occurring jumps in the signal flow.

5 Writing pd-externals

We have already heard about the pd objects I worked with. We also heard about how they are working roughly. But as it was a major issue of my project, I want to look at the three externals I wrote on my own in detail. At the beginning I want to give a short overview over the principle procedure designing an external. If you want to know more I would recommend the paper "How to write an external for pure-data" written by Johannes Zmölnig [12]. My knowlegde of writing externals is also mainly based on this paper.

pd is written in C and therefore it is possible to write your own external in C and use it in pd. To write externals it is further necessary to compile the written code. This would be possible, for instance with the GNU C compiler - gcc on a linux operation system. The linker I used is the GNU linker - ld. Additionally an interface for programming is necessary, this is given by the include file "m_pd.h".

5.1 blob

5.1.1 Data space

At the beginning it is necessary to generate a class with its data space.

```
static t_class *blob_class;

typedef struct _blob{
t_object x_obj;

t_atom *buffer;
t_int anzahlbits;
}t_blob;
```

`*blob_class` is the pointer to the new class. `t_blob` prepares the data space of the class. The first element of the structure is the variable `x_obj` of the type `x_object`. In this variable properties of the new object are stored. After this obligatory variable,

the variables for our own purposes are declared. For the blob object we need a pointer to a buffer of the type `t_atom` and the value of the resolution of our audio data has to be stored in an `t_int`.

5.1.2 Method space

As we are aware of the data space know, we need methods to work with the data space. In the so called method space, those functions are described in detail. On the right inlet of our pd object, we want to specify the resolution of the audio data. Therefore we need a method, which is called if the resolution is set over the inlet.

```
void blob_resolution(t_blob *x, t_floatarg f) {
        x->anzahlbits = (t_int)f;
}
```

The function `blob_resolution` needs to get a pointer of the type `t_blob`, to access and modify our data space. The argument of the type `t_floatarg` is the transfer value and contains the received data from the left inlet. In our case it specifies the resolution. The resolution is an integer, so we typecast it and assign it to our variable `anzahlbits` of the data space. The function doesn't need to return any value. Therefore the return value is of the type `void`. For our blob-object we need a second method, to react to the incoming audio data (`blob_list`). The data is sent from the pack~ object on the message layer of pd. It arrives in form of a list. Before I discuss the content of this method I want to show how our class is generated and how the instances of our class are initialized within the constructor.

5.1.3 Generation of a new class

To generate a new class it is necessary for pd, to know about the data and method space. The function which is responsible for that has the name `blob_setup` and is called only once by pd at the time of loading.

```
void blob_setup(void)
{
        blob_class = class_new(gensym(``blob''),
                        (t_newmethod)blob_new, 0,
                        sizeof(t_blob),
                        CLASS_DEFAULT, 0);
```

```
        class_addlist(blob_class, blob_list);
        class_addmethod(blob_class, (t_method)blob_resolution,
                gensym(``resolution''), A_DEFFLOAT, 0);
}
```

The instruction `class_new` creates a new class and returns a pointer to this class. The name of the object has to be specified as first transfer value. The second value is the constructor of the class, the third is the destructor (not necessary for our object, hence it's zero), followed by the size that needs to be allocated for our data space. The fifth argument particularizes the graphical type of the class. In our case the object is of the type `CLASS_DEFAULT`. `CLASS_DEFAULT` creates an object with at minimum one inlet. Afterwards up to six numerical or symbolic transfer values of the object can be specified. This itemization has to be zero terminated. `blob` doesn't need any transfer values, so the first argument of our not existing list is the zero termination. After the class is created by `class_new`, the method space has to be added. There is a prepared function for adding a method handling "list"-messages to the class. It is called `class_addlist` and the transfer parameters have to be a pointer to the class where it is added to and the name of the method where it is added. The general solution adding a method to a class is constituted by the function `class_addmethod`. The first transfer parameter is again a pointer to the class where the method is added. What follows is a name for the method, the selector, the type of the arguments which are transferred (up to six different are allowed) and an obligatory zero padding at the end of the transfer parameters. The selector is a symbolic name associated with this method (in our case ``resolution''). And our resolution parameter has the data type ``float'', hence `A_DEFFLOAT`. Adding the methods to our class concludes the generation of our new class.

5.1.4 Constructor

What follows is the constructor. The constructor is called when a new object of our class is created within pd.

```
void *blob_new(void)
{
        t_blob *x = (t_blob *)pd_new(blob_class);
        outlet_new(&x->x_obj, &s_list);
        inlet_new(&x->x_obj, &x->x_obj.ob_pd,
```

```
                gensym(``float''), gensym(``resolution''));
        x->buffer = (t_atom *) 0;
        x->anzahlbits = 0;
        return (void *)x;
}
```

The constructor is always of the type `void *`. In the first line an instance of the class is generated, memory for the data space initialized and a pointer pointing to the data space is returned. Afterwards a new outlet, where the rearranged data is released and a new inlet, where the actual resolution can be set, has to be added. The function `outlet_new` needs a pointer to the variable `x->x_obj` as first argument and the type of the selector as second argument. Are there messages with different types of selectors, a "zero" would be the right choice, instead of `&s_list`. Internally the function returns a pointer to the new outlet and stores it in the variable `x_obj.ob_outlet`. The function `inlet_new` does the same for an inlet. An inlet is generated at the object, where `x->obj` is pointing to. The second argument is the destination of the inlet and points usually to `x->xobj.ob_pd`. The following two arguments are selectors. When there is a message with the selector `‘‘float’’` arriving at the new inlet, this selector is substituted with the selector `‘‘resolution’’` and the associated method is executed. After all inlets and outlets are created, every single variable of the data space is initialized. The constructor returns a `void`-pointer to the initialized data space.

5.1.5 The signal processing function in the method space

Going back to the method space, where we spared the main content of our new pd object. At first let us look at the function head:

```
void blob_list(t_blob *x, t_symbol *s, int argc, t_atom *argv)
{...}
```

We need a pointer of the type `t_blob` to the data space of our actual class, a second pointer to the selector symbol (in this case, as we are always dealing with "lists", the pointer to the selector symbol is always `&s_list`), a third transfer parameter representing the argument count (amount of the atoms in the list) and as last parameter a pointer to this actual atom list. In this function the bit shifting, depending on the resolution factor, is realized. There are three main possibilities we are dealing with. If the resolution factor `x->anzahlbits` is "zero", the received data isn't processed in any way. If the resolution factor is equal to `groeszeint` or greater, the received data

is mapped into an integer value with the maximal word width of `groeszeint`. If the resolution factor is between "zero" and `groeszeint`, the word width of the data is reduced (according to `x->anzahlbits`) by means of bit shifting and rearranged space saving into new words.
Bringing an easily understandable example about the reduction:

There are 64 audio samples with a resolution of $32bit$ in one audio block, this equals $2048bit$ (equals $2048/8 = 256byte$). If the resolution is for example reduced to $11bit$ the remaining bits add up to $704bit$ (equals $88byte$). In this example the data reduction would be 65.625%, certainly accompanied by a great loss of audio quality. Illustration 11 is giving information about, how the bitshifting is realized.

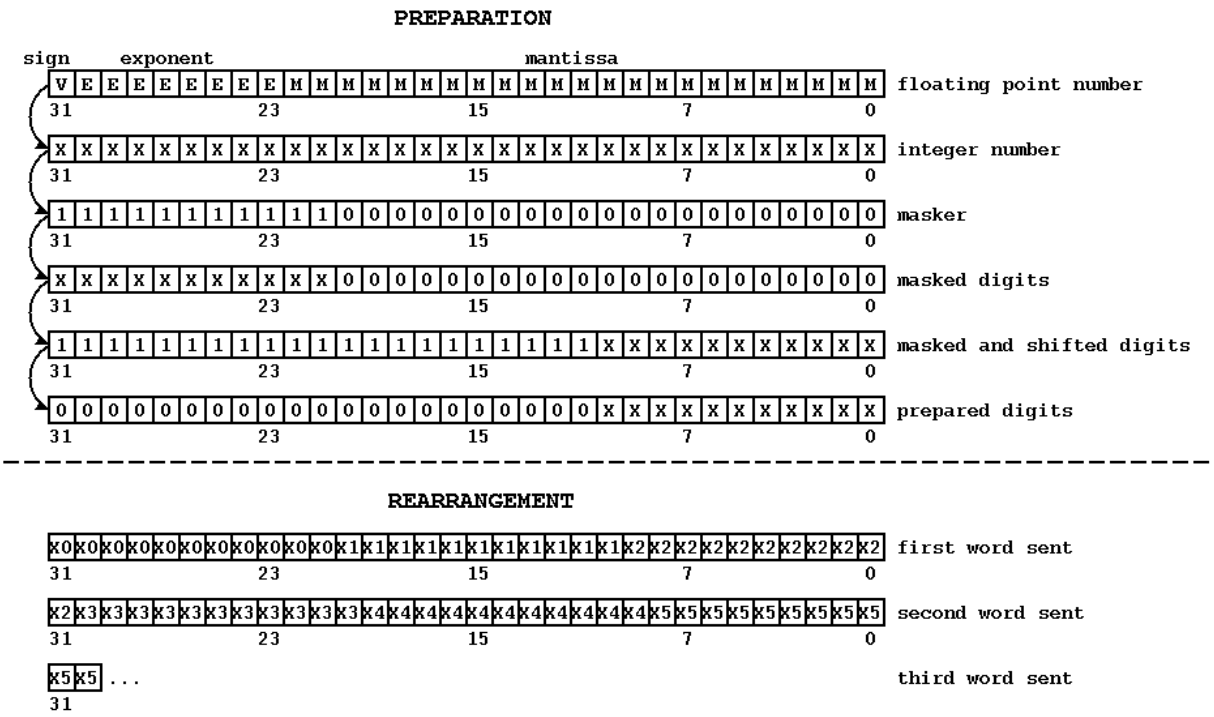

Figure 11: Functionality of blob

At the end of this chapter a short notice about the two memory functions `getbytes` and `freebytes`.

```
{...}
x->buffer=getbytes(sizeofbuffer);
{...}
```

```
freebytes(x->buffer, sizeofbuffer));
{...}
```

The first code line allocates `sizeofbuffer` byte of memory space and returns a pointer to this space. The second code line deallocates `sizeofbuffer` byte of memory space at the address `x->buffer`. To speak from my own experience: Don't forget to free the once allocated memory space. Otherwise the memory you are working with overflows.

5.2 unblob

The counterpart of the object "blob" is the object "unblob". The received data is unpacked by the object "unpackOSC", piped through "pipelist", routed via "routeOSC" and once arrived at the object "unblob" it is decoded and prepared for the following "funpack~" object, where the data is brought back from the message layer to the signal layer of pd. As "unblob" constitutes the counterpart of "blob", it is also working with the same concept.

```
typedef struct _unblob{
t_object x_obj;
t_atom *buffer;
t_int anzahlbits;
}t_unblob;
```

The data space contains the same variables. And a method, reacting to "list" messages is needed.

```
void unblob_setup(void)
{
        {...}
        class_addlist(unblob_class, unblob_list);
}
```

In the constructor we need to create an instance of our class, add one new outlet and initialize the variables from the data space.

```
void *unblob_new(void)
{
        t_unblob *x = (t_unblob *)pd_new(unblob_class);
        outlet_new(&x->x_obj, &s_list);
        x->buffer = (t_atom *) 0;
```

```
        x->anzahlbits = 0;
        return (void *)x;
}
```

The function `unblob_list` comprises the main signal processing part of this object. To choose the right decoding method, it is necessary to evaluate the resolution, which is provided and transmitted from the "blob" object. There are again three main possibilities, of how the data is stored. If `x->anzahlbits` equals "zero" the resolution of the data is a floating point digit and no further restoration has do be made. If `x->anzahlbits` equals `groeszeint`, the resolution of the data is an integer digit, it has to be brought back into the original number range and released as floating point argument. If `x->anzahlbits` is a number between "zero" and `groeszeint` the resolution is exactly as high as the amount of this number. Therefore a preperation of the packed data is necessary as shown in illustration 12.
The bits belonging together have to be masked, shifted into the right position and brought back into the right number space to be released as floating point digits. If bits are distributed over two different words, the restoration process is a little bit more complicated, as the result is dependent on two intermediate results of the masking and shifting process.

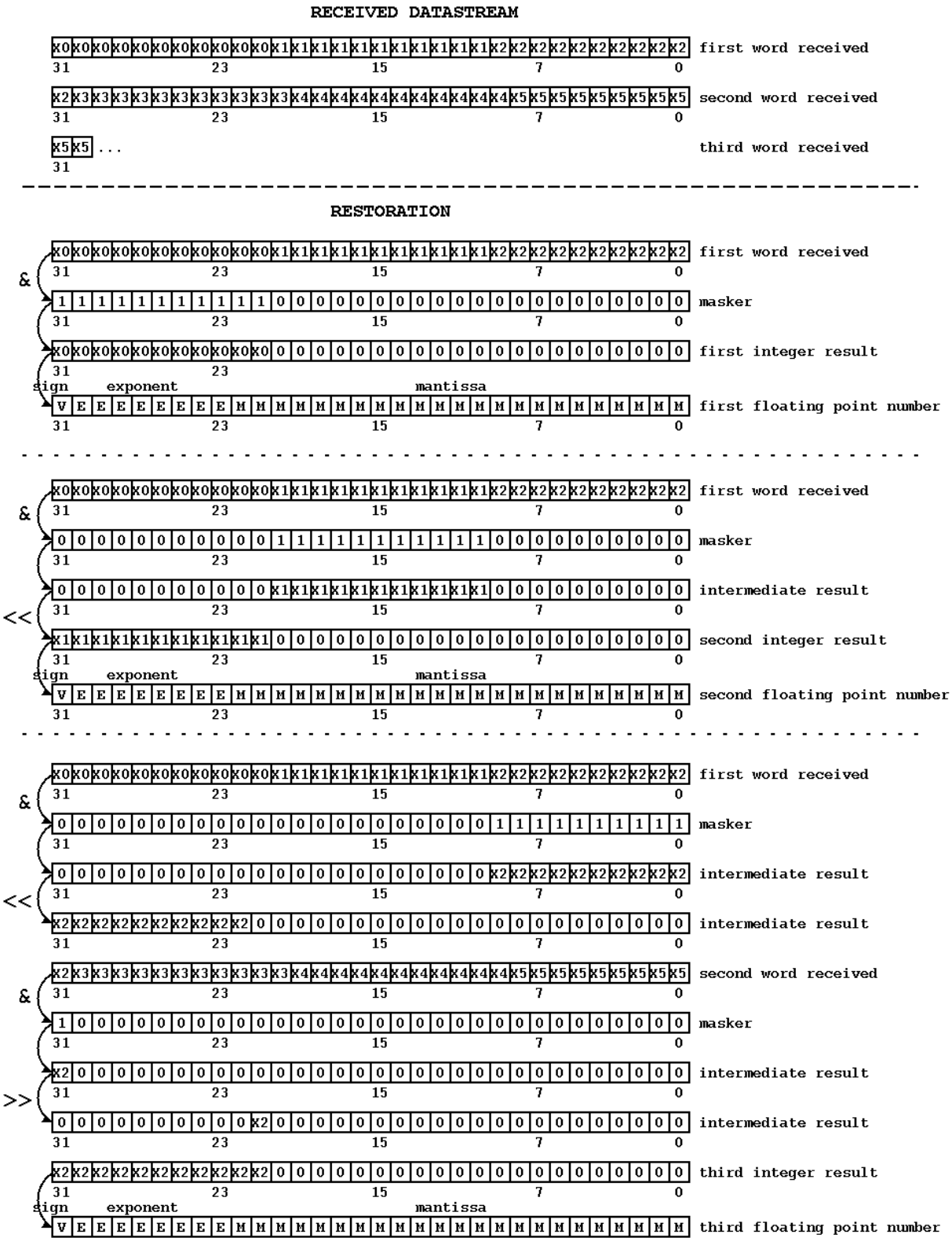

Figure 12: Functionality of unblob

5.3 funpack~

In this chapter I will introduce the object funpack~. The received packages have already been routed and the internal data is prepared. The last step is to bring it back from the message layer to the signal layer and to absorb if possible occurred errors. At the beginning I want to shortly introduce the functions you have to deal with, writing a signal external in pd.

5.3.1 Signal classes

Signal classes are like other pd classes, additionally they have methods dealing with signals. To distinguish easily between a signal and a normal object, the signal objects names are expanded with the character "~". Every signal class needs a method for processing the audio signals. That is caused by the fact, that if pds audio engine is started messages with the selector "dsp" are sent to every signal class. In our case this method looks as follows:

```
static void funpack_tilde_dsp(t_funpack_tilde *x,
        t_signal **sp)
{
        dsp_add(funpack_tilde_perform, 3,
                sp[0]->s_vec, x, sp[0]->s_n);
}
```

Within this method, a so called "perform"-routine, is added to the DSP-tree of pd. There are a couple of transfer values, showing the way to the array where the signal is stored and also the length of those signal arrays have to be delivered. In the "perform"-routine the actual signal processing is carried out.

5.3.2 The functionality of the object

As I proposed earlier in this report funpack~ is an expanded version of unpack~. Data is sent on the message layer predominantly with the "list"-selector and released as audio signal on the signal layer. funpack~ is extended about a bigger ringbuffer with resampling if a certain filling level is understepped or overstepped, fades in/out the different audio signals, interpolates at discontinuities appearing after an dropout and mixes signals in dependency of the mode flag.
Two ringbuffers with the same size are allocated in the memory. If the ringbuffer is

filled under one third of its size, ringbuffer one/two is copied into ringbuffer two/one and every sixth sample is linearly interpolated, to prevent the buffer of an underflow. On the other side, if the buffer is filled over two third of its size, ringbuffer two/one is copied to ringbuffer one/two and every sixth sample of the existing data is deleted, to prevent a buffer overflow.
Illustration 13 and 14 accompany the functionality, whereas illustration 14 is, in order of better understanding, drawn as linear buffer.

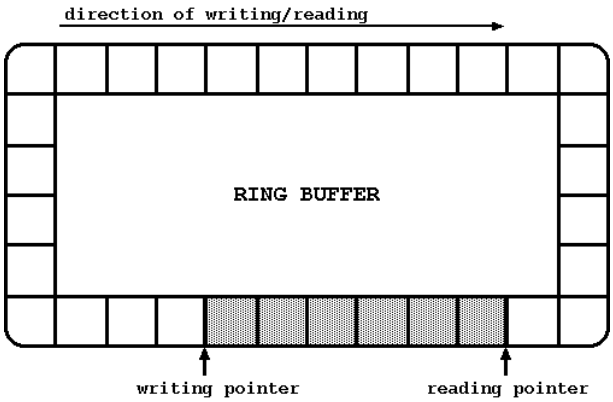

Figure 13: Ringbuffer

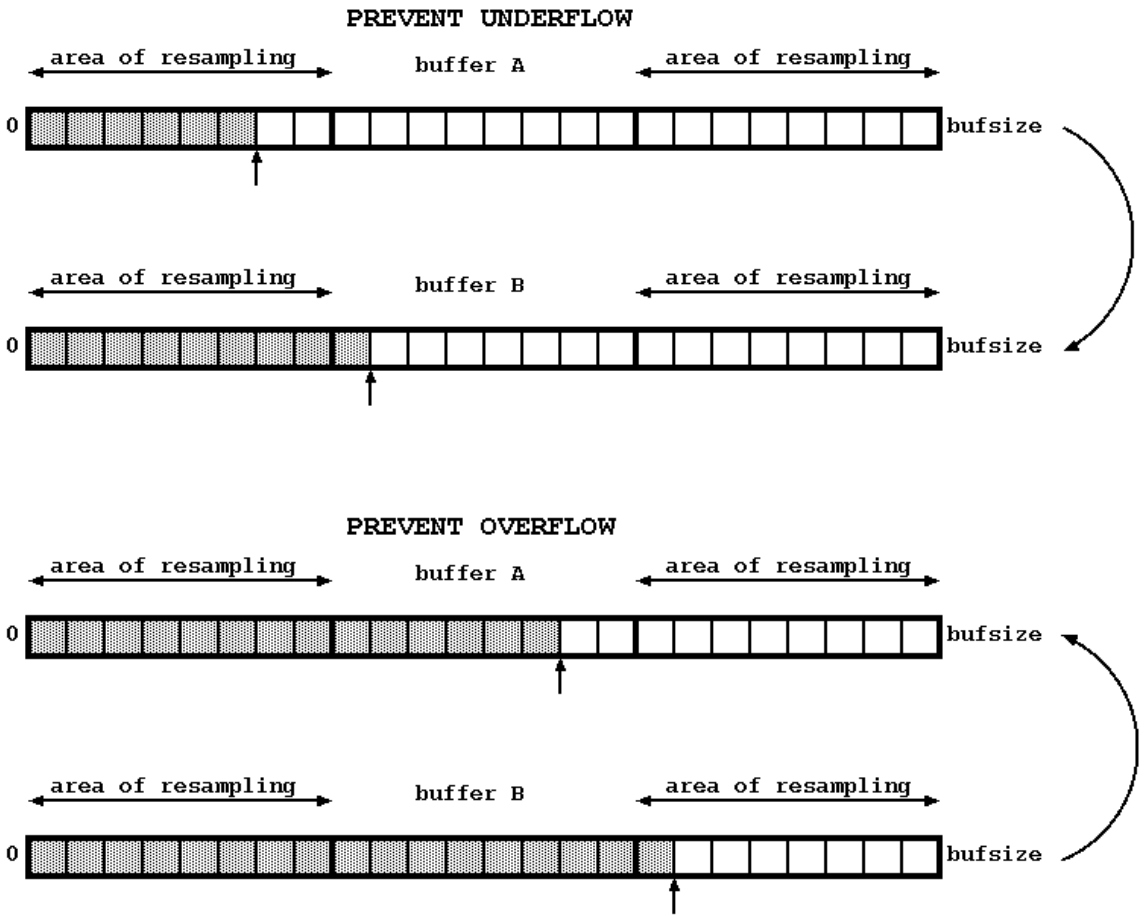

Figure 14: Resampling of the ringbuffer

Resampling the audio data is helpful, but if a longer dropout occurs, it couldn't sustain the signal. Therefore the object detects dropouts in the transmission over the Sequence Number and fades the audio signal if necessary out/in. This could be watched in illustration 15, where a $200Hz$ sinus signal is transmitted and every 25. and 26. package is artificially suppressed.
Figure 16 shows the case, when two channels are mixed at one funpack~ object. If a dropout occurs, like displayed in line two and three of the figure, funpack~ tries to soften the jump, which appears in the time signal after a dropout (displayed in line 1). Line four of the illustration shows the perfect case, like the signal would appear if no dropout occurs. The frequency spectrum of this case is seen in figure 17. The more reddish the spectrum is, the more energy is contained in a certain frequency band.

Lastly we want to look at the case, when three signals with some missing packets are mixed at one drain. Figure 18 shows the spectrum. It can be observed, that some errors are not detected any more, so there are jumps within the signal, which can be identified by the high frequency peaks in the spectrum.

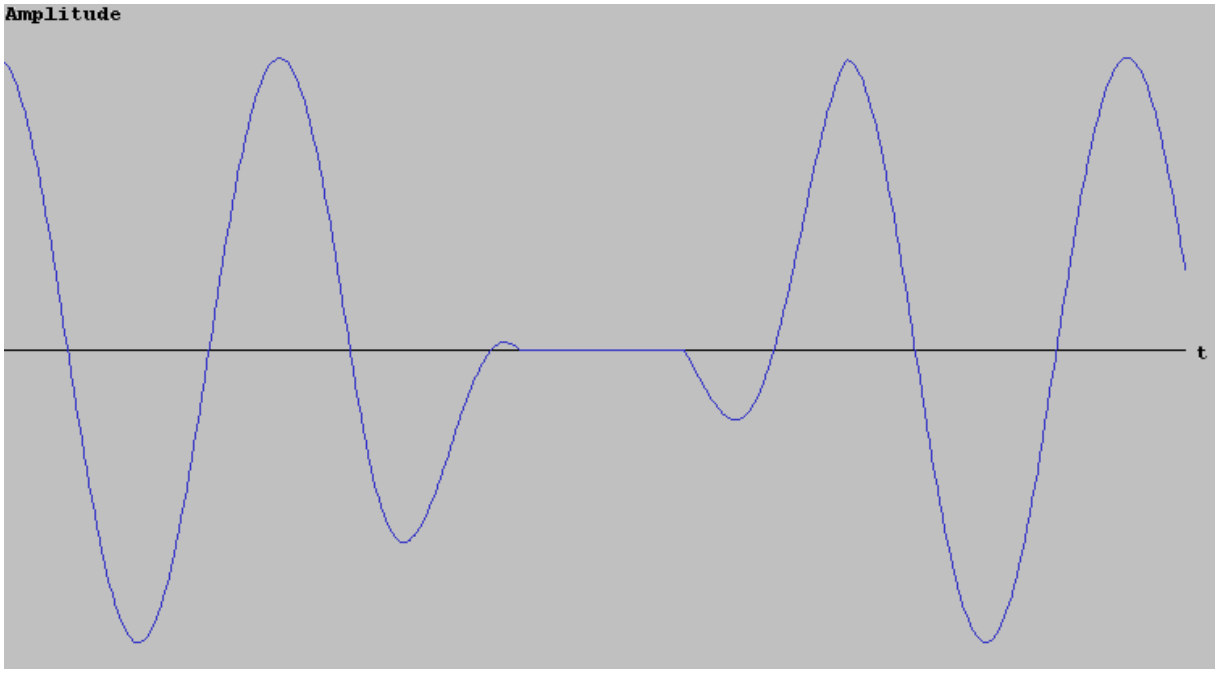

Figure 15: 1 channel time domain

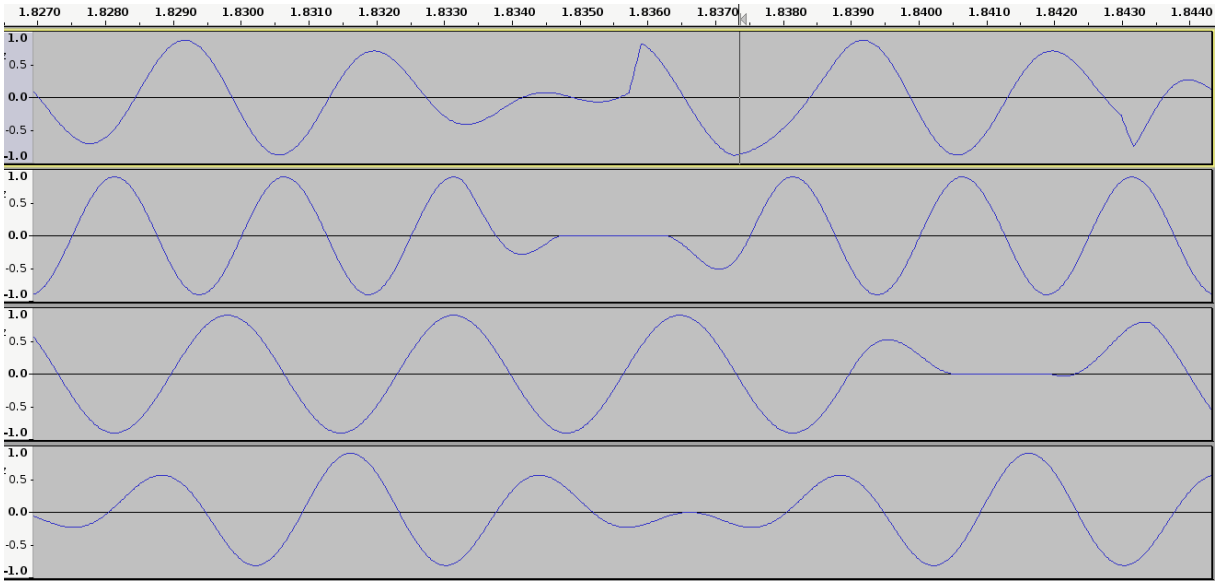

Figure 16: 2 channels time domain

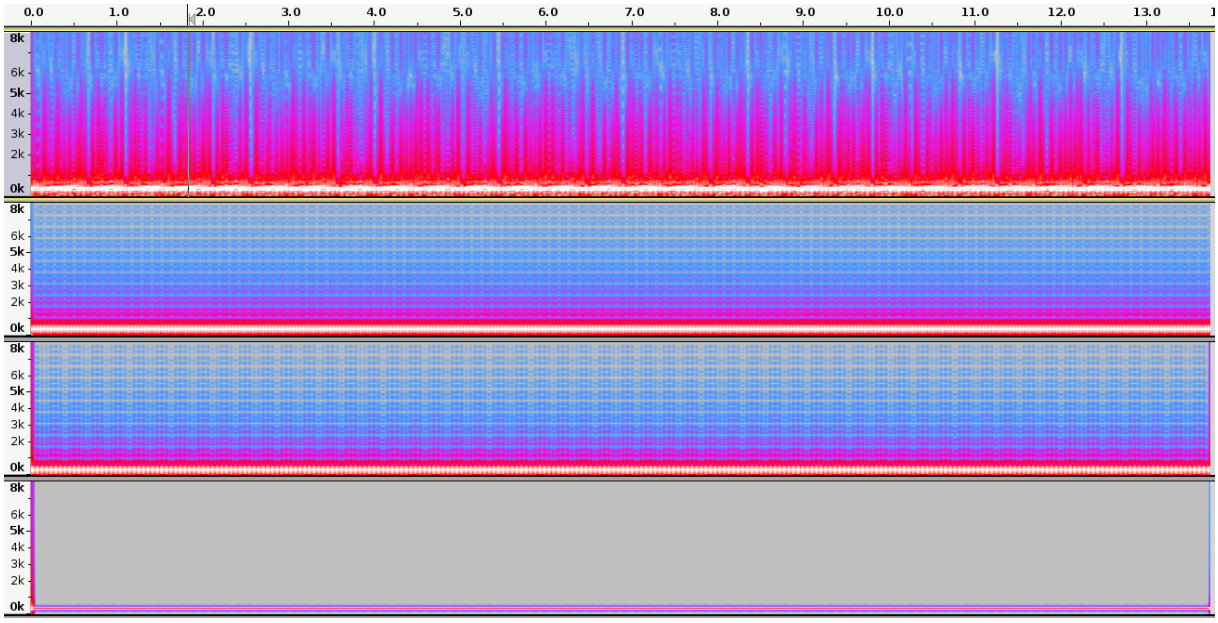

Figure 17: 2 channels frequency domain

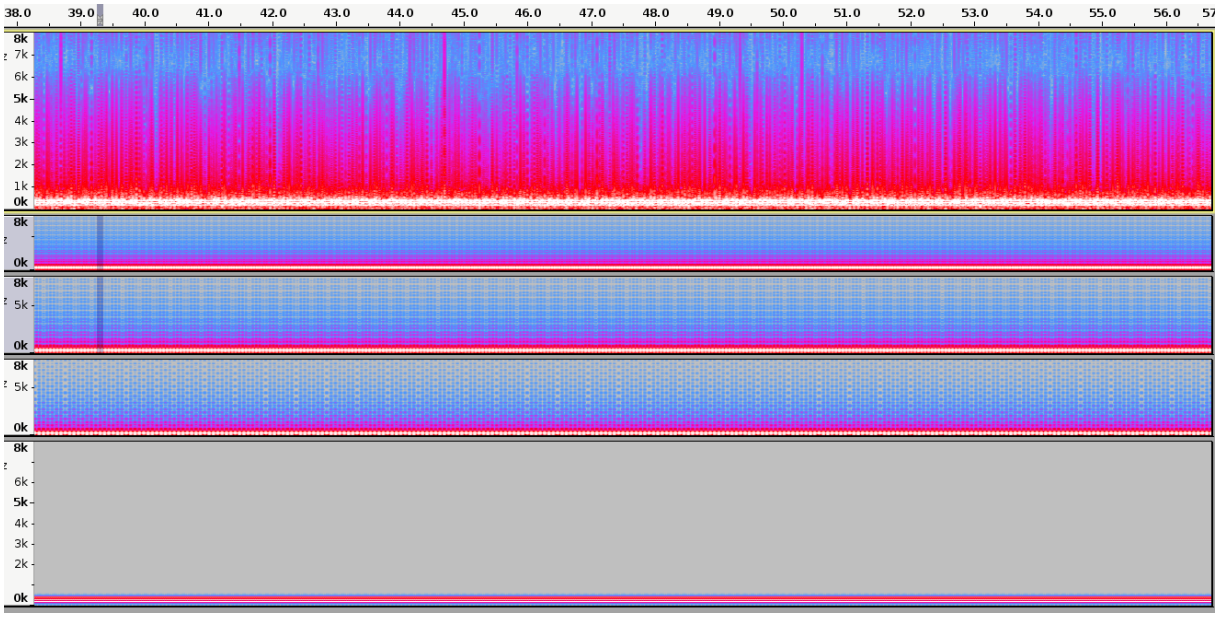

Figure 18: 3 channels frequency domain

6 Test environment, working aids

As test environment served a couple of different open source software audio components. The test patches were written within pd version 0.42.5, where a central component has been the OSC library of Martin Peach. pd used JACK (version 0.3.2) as underlying sound server, from where the data has been sent to Audacity (version 1.3.5) to record and interpret the results. All components have been working at a sampling frequency of $f_S = 44100Hz$. In illustration 19 a screenshot of the "working place" is pictured.

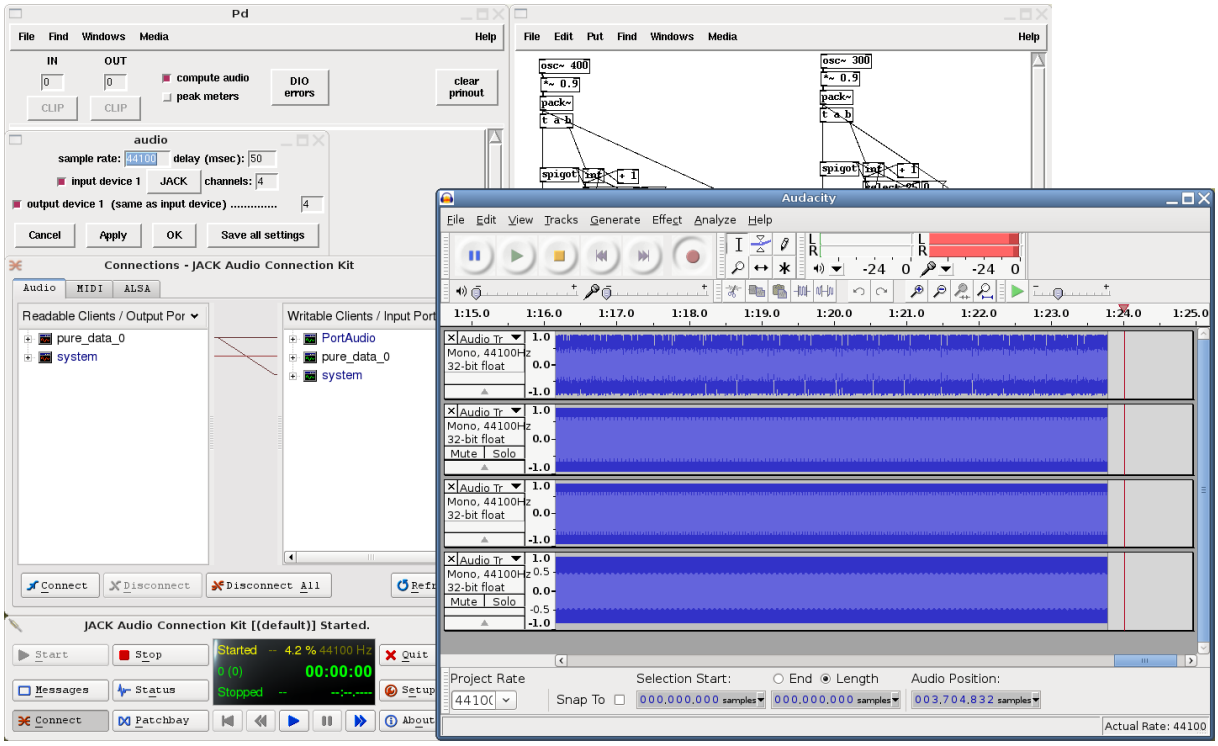

Figure 19: Test environment

7 Future proposal

There are still some problems remaining with the actual implementation, especially with the udp objects. If there is more than one channel, to send over one udpsend/udpreceive pair, data gets lost. To avoid this loss, every channel has to be transmitted over a single udpsend/udpreceive pair, which is unnecesarrily complicated.
Using udpreceive in combination with JACK sound driver freezes the pd GUI, a not really convenient situation.
To get away from the unsatisfactory solution with the sequence numbers, the problems with the time stamp should be solved in a first step. Solving this problem offers the possibility of very accurate sorting of the audio data at the receivers side.

References

[1] Audinate. Dante. http://www.audinate.com/, 2009.

[2] AVnu Alliance. Audio Video Bridging (AVB). http://www.avnu.org/, 2009.

[3] Caceres, J.-P. Jacktrip. https://ccrma.stanford.edu/groups/soundwire/software/jacktrip/, 2008.

[4] Cirrus Logic. Cobranet. http://www.cobranet.info/, 2009.

[5] Digigram. Ethersound. http://www.ethersound.com/, 2008.

[6] Fostex. Netcira. http://www.netcira.com/docs/home/netcira_front.shtml, 2009.

[7] Internet Assigned Numbers Authority. MIME Media Types. http://www.iana.org/assignments/media-types/, 2009.

[8] Mills, D. Network time synchronization research project. http://www.eecis.udel.edu/~mills/ntp.html, 2009.

[9] Roland. REAC. http://www.roland.com/, 2009.

[10] Wikipedia. Audio over ethernet. http://en.wikipedia.org/wiki/Audio_over_ethernet, 2009.

[11] Wright, M. The open sound control 1.0 specification. http://opensoundcontrol.org/spec-1_0, 2002.

[12] Zmoelnig, J. Howto write an external for puredata. http://iem.at/pd/externals-HOWTO/, 2001.