Bachelor Thesis
Virtual Distributed Concerts

15. August 2004

Simon Schampijer[*]

[*]<Simon.Schampijer@web.de>

This Document has been made with LaTeX. You can get the source and the pdf document at:

    <url>

The Author would like to thank W. Schmidt for his paper LaTeX2e-Kurzbeschreibung on which this LaTeX paper is based.

Contents

List of Figures

1 Presentation of the Thesis

I was working on the project described in this paper during my scholarship absolved at IRCAM[1], Paris. The Bachelor Thesis "Distributed virtual concerts" is part of my studies of Internet Science and Technology at the Fachhochschule Kiel[2].

1.1 Presentation of the team

The IRCAM, Institute de Recherche et Coordination Acoustique/Musique, was founded in the 1970s by Pierre Boulez. The Institute is one of the most important European institutes for both science about music and sound and avant garde electro-acoustical art music. IRCAM is doing research in acoustics, signal processing and computer science applied to music. The free software team of IRCAM founded in 2002 has been working on several research and development projects. IRCAM free software are today:

1. *jMax*, a visual programming environment for real time musical interaction
2. *OpenMusic*, a visual programming environment for music composition
3. *SDIF*, library and tools to access SDIF files (Sound Description Interchange Format)

`http://freesoftware.ircam.fr/`

Current projects of the free software team are centered on distributed architectures, more precisely on multimedia distributed applications over Internet and on grid computing. They are carried on in collaboration with CEDRIC, CNAM's[3] computer science research center.

2 The Distributed virtual concert

2.1 Introduction

The aim of the distributed virtual concert is to distribute each actor of a concert (musicians, sound engineer as well as public) on the Internet. All persons participating in the performance are geographically distributed and communicate by real time audio streaming over Internet. The picture 1 will show us the general architecture of the distributed virtual concert.

2.2 What is this project for?

The technologies that are developed in this project (real time audio streaming over Internet, distributed real time control, specialization) can be applied to other domains than musical applications, in particular distributed virtual or

[1]contact: François Déchelle,(`dechelle@ircam.fr`)
[2]contact:Dr.-Ing. Helmut Dispert(`Helmut.Dispert@FH-Kiel.de`)
[3]Conservatoire National des Arts et Métiers

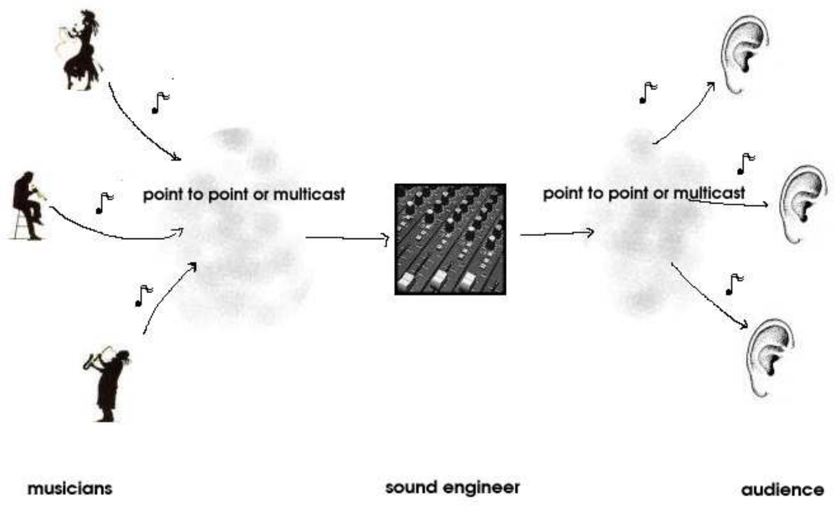


Figure 1: The Distributed virtual concert

augmented reality applications. Furthermore, this project offers an original framework to experiment the new functionalities offered by high speed networks: native multicast routing and quality of service.

2.3 State of the Art

Several experiments of real time multimedia performances on the Internet have already been achieved. We present a brief overview of the known experiments.

2.3.1 MIDI Implementation

Most of the real time multimedia performances use the MIDI standard (Musical Instrument Digital Interface).In the implementation of Eliens latency between the production and the consumption of sounds is not constant which means that variations are perceived while hearing. ...

2.3.2 Transport of PCM audio streams in real time

> On September 23, 2000, a jazz group performed in a concert hall at McGill University in Montreal and the recording engineers mixing the 12 channels of uncompressed PCM audio during the performance were not in a booth at the back of the hall, but rather in a theater at the University of Southern California in Los Angeles[1].

In this architecture the musicians played together in the same place and the on the fly created PCM audio streams of each musician were transported over the Internet and mixed at a physically distributed place. The musicians were located in the same place, causing no problem of interaction between each other. For our project we would like to let the musicians interact across the Internet.

2.3.3 Distributed Musicians synchronized with a fixed delay

On October 19, 2003, two members of a jazz quartet performed in a hall at IRCAM in Paris and the other two members performed in a hall at CNAM in Paris. Each source transmit uncompressed high quality audio data to the other players and get back a mixed, all the sources, feedback. To reach a global synchronized audience of the performance the sources are synchronized with a fixed delay of one bar during the concert between the direct sound of the musicians and the global feedback. This technique allows the musicians to play a real time and collective musical piece. A video conference system was installed to permit the musicians to see each other. Technical Issues:

1. *Multicast*,one Sender to n-receivers connections
2. *RTP*,transport protocol is the Real time Protocol RTP
3. *jMax*,the synchronization of the audio data is made in a jMax object(nJam)

The first concert performed with the streaming engine as part of jMax failed. The problem was the fact of using multicast tunneling at both sides IRCAM and CNAM to access the multicast enabled Internet. Multicast tunneling is the technique where multicast packets are encapsulated in unicast packets over links that are not multicast enabled. So the packets had to pass the unicast routers which means that the packets were treated as normal packets without any priority. The traffic inside IRCAM and CNAM during the concert was very high which caused a unforeseen delay for the packets. Real-time performance needs absolutely native multicast to avoid such problems.

2.4 Latency and known Problems

As we said in the global introduction before, the aim of this project is to provide means to musicians to play across the Internet in real-time. First we have to think about the meaning of real-time and what this means to our application. To explain this we will have a look at an all day situation. Person A decides to say "Hello"to Person B. A observe how B reacts and modify his behavior accordingly. In this case the reaction needs to be very quick, not to bore the person face to face and sometimes a slow reaction is adequate. But in every case we would like to have a feedback by which we can observe the results of our actions. So we can say that real-time is related to the application.

So how fast would response time need to be, to provide musicians with human interactive experience?
Nelson Posse Lago and Fabio Kon argued in their paper "The Quest for Low Latency" that *high latencies (around 20-30ms) are probably perfectly acceptable for typical musical applications*[3]. Anyway we have to do some experiments in the field of musical interaction. Timing is tightly tied to unmeasurable aspects

of music such as "feel". Experiments could be as already mentioned by Nelson Posse Lago and Fabio Kon to let musicians play together and to subject one of the instrumentalists to different feedback latencies to asses their effect over the performer[3]. As a first experiment we could discuss the result of a demonstration concert organized the 10 Mai 2004 at IRCAM. The jMax object nJam discussed in the section above was used during this experiment. We had two musicians playing at distributed rooms at IRCAM both using an instance of jMax-nJam. The synchronization mechanism added consensually delays in each musician's side fills, making each musician hears himself with a constant latency. During this concert the participants performed with a delay of one measure.

At the beginning the musicians had difficulties to play together. They felt not so comfortable with that interaction. But after a while of "training" they both agreed that a time of adaption was necessary to interact in a musical way. This experiment succeeded because we had a fixed delay, "Jitter", on the other hand, can be a bigger problem. I argue that a musical performance with a varying latency wouldn't have been possible. The results achieved during this experiment could not give any answers if high latencies are acceptable for typical musical applications but could give a first impression of the importance of feeling in musical interaction and that a musician is able to adapt to a fixed delay. If we assume that a musical interaction between musicians accept a delay of 20-30s the next question is directly leaded:

Can today's computer networks achieve that target response time?
To give a short introduction to this question I would like to give a briefly overview of limits of the physical universe taken of the work of Stuart Chreshire "Latency and the Quest for Interactivity"[4]. The speed of light in vacuum is 300 000 and the speed of light in typical glass fiber is roughly 33% less, about 200 000 km/sec. How long would it take to send a signal to someone on the far side of the planet and get a response back?

- Speed of light in vacuum = 300 000km/sec
- Speed of light in glass fiber = 200 000km/sec
- Circumference of earth = 40 000km
- Total round-trip delay to the far side of the planet and back = 40 000 / 200 000 = 200ms

This means that physically an interaction in real-time with someone on the other side of the planet under perfect conditions isn't possible. *But is it possible if we interact with someone much closer?*

- Distance from Paris to Bordeaux = about 500km
- Total Round-trip delay to Bordeaux = 1000 / 200 000 = 5ms

This result means that an interaction would be physically possible. Now we have a look at the possibilities today's high speed networks offers. IRCAM is connected to the high speed network Renater[4] which is a network for the domains research, technology, universities and culture in France. The Band with of this network is about 2.4 GBit/s. So we were curious about the round-trip delay to Bordeaux. As Destination we choose a server of labri[5] in Bordeaux which is also connected to the renater network. To Measure this we were using the ping and the traceroute utility. We did the following from a terminal inside IRCAM:

```
ping -c 1 cbi.labri.fr
  PING natchaug.labri.fr (147.210.9.225) 56(84) bytes of data.
  64 bytes from natchaug.labri.fr (147.210.9.225): icmp_seq=1 ttl=51 time=9.94 ms

  --- natchaug.labri.fr ping statistics ---
  1 packets transmitted, 1 received, 0% packet loss, time 0ms
  rtt min/avg/max/mdev = 9.940/9.940/9.940/0.000 ms
```

The result was a round-trip-time(rtt) of 9.94ms from IRCAM(Paris) to LaBRI(Bordeaux).

```
traceroute cbi.labri.fr
traceroute to natchaug.labri.fr (147.210.9.225), 30 hops max, 38 byte packets
 1  129.102.24.254 (129.102.24.254)  0.474 ms  0.391 ms  0.356 ms
 2  ircam.gw.net.ircam.fr (129.102.254.1)  0.301 ms  0.194 ms  0.171 ms
 3  ircam.r-ext.net.ircam.fr (129.102.254.245)  0.383 ms  0.283 ms  0.261 ms
 4  ircam-f1-1.cssi.renater.fr (193.51.182.10)  1.208 ms  1.129 ms  0.888 ms
 5  jussieu-g0-0-10.cssi.renater.fr (193.51.180.202)  1.425 ms  1.218 ms  1.560 ms
 6  nri-b-pos12-0.cssi.renater.fr (193.51.180.158)  8.990 ms  9.201 ms  8.959 ms
 7  poitiers-pos5-0.cssi.renater.fr (193.51.179.134)  6.171 ms  6.326 ms  6.149 ms
 8  bordeaux-pos1-0.cssi.renater.fr (193.51.179.254)  9.792 ms  8.546 ms  8.632 ms
 9  esra-bordeaux.cssi.renater.fr (193.51.183.45)  9.135 ms  10.554 ms  9.170 ms
10  195.83.241.70 (195.83.241.70)  9.124 ms  9.387 ms  9.169 ms
11  hca0.u-bordeaux.fr (147.210.246.205)  9.828 ms  9.662 ms  9.798 ms
```

We also did a traceroute on Grame[6] in Lyon which is also connected to the Renater network and is a potential partner of IRCAM to perform virtual distributed concerts.

```
traceroute www.grame.fr
traceroute to rd.grame.fr (194.5.49.4), 30 hops max, 38 byte packets
 1  129.102.24.254 (129.102.24.254)  0.548 ms  0.386 ms  0.356 ms
 2  ircam.gw.net.ircam.fr (129.102.254.1)  0.311 ms  0.178 ms  0.174 ms
 3  ircam.r-ext.net.ircam.fr (129.102.254.245)  0.335 ms  0.267 ms  0.265 ms
 4  ircam-f1-1.cssi.renater.fr (193.51.182.10)  1.056 ms  0.915 ms  0.898 ms
 5  jussieu-g0-0-10.cssi.renater.fr (193.51.180.202)  2.457 ms  1.635 ms  1.634 ms
 6  nri-b-pos12-0.cssi.renater.fr (193.51.180.158)  2.263 ms  2.073 ms  1.992 ms
 7  lyon-pos9-0.cssi.renater.fr (193.51.179.130)  9.935 ms  9.737 ms  9.818 ms
 8  in2p3-lyon.cssi.renater.fr (193.51.181.6)  9.748 ms  9.899 ms  9.519 ms
 9  Lyon-TIF.in2p3.fr (134.158.224.6)  10.044 ms  10.142 ms  10.097 ms
```

Here we see that the packet passes 2 routers inside IRCAM before being on the Renater network. If we suppose that latencies around 20-30ms are acceptable for typical musical applications and that Renater networks offers us a rtt near 10ms from Paris to Bordeaux or Lyon then we should be able to perform Virtual distributed concerts inside the Renater network. But we shouldn't forget that we discussed the latency added by the network and that this doesn't include the latency that could be added by the software.

[4]Le Réseau National de Télécommunications pour la Technologie, l'Enseignement et la Recherche

[5]Laboratoire Bordelais de Recherche en Informatique

[6]Centre National de Création et Musicale

2.5 The project and the Thesis

This paper presents a RTP protocol based application for the distributed virtual concert. The application uses the live.com library to implement the RTP/RTCP protocol functionality and is a Jack Audio Connection Kit client able to play and transform the audio data generated by applications supported by JACK or music from real musicians captured as a sound card input. The player (application or musician) is connected to JACK and sends its high quality audio stream transmitted by the live.com library through the network.

2.6 Aimed Milestones in the project

1. *Properties of RTP* : getting familiar with RTP and other protocols for multimedia streaming

2. *Overview of existing RTP libraries* : open source libraries and their development state. Which open source libraries supports the RTP, RTCP and RTSP protocols. How high is the project activity, code quality and organization and what is about the accessibility by CVS.

3. *JACK client streaming audio using RTP* : implemented with the chosen multimedia library

4. *JACK client receiving audio using RTP*

5. *Support of multicast* : an overview, possibilities offered to our implementation

6. *Implementation of a synchronization mechanism* : compensation of latency added by the network

7. *Including an algorithm compensating clock skew* : compensate for the clock skew difference

8. *Testing* : latency added by network and software, musical interaction of the musicians

3 Internet Protocols for Multimedia

3.1 RTP

Real Time Transfer Protocol (RTP) is a transport protocol that provides end-to-end network transport functions for applications transmitting data with real-time properties, such as interactive audio and video. The following services are offered by RTP :

- detect packet loss and reconstruct packet sequence : the 16 bits sequence number increment by one for each RTP packet sent.
- format identification : The payload type identifies the format of the framed data. This permit the application receiving the RTP packets to know how to play them. A set of default RTP payload types are defined in RFC 3551.
- source identifying: the SSRC (a 32 bit header field) identifies the synchronization source which permits an application to identify streams from different sources.
- time stamping: the 32 bit time stamp is used to know when a packet has to be played and to synchronize the streams from different sources in the application

As network services unicast or multicast can be used. Typically applications run RTP on top of UDP to make use of its multiplexing checksum service. Both protocols contribute parts of the transport protocol functionality. RTP don't run on top of TCP because TCP does acknowledgment and retransmission of lost packets which isn't gained by multicast and the time of delivering a packet is more important than packet loss. RTP itself does not provide any mechanism to ensure timely delivery or provide other quality-of-service guarantees.It does not guarantee quality of service for real-time services, which means that RTP does not address resource reservation.

3.2 RTCP

To monitor the quality of service and to convoy information about the participants in an on-going session the RTP control protocol (RTCP) is used. RTCP is based on the periodic transmission of control packets to all participants in the session, using the same distribution mechanism as the data packets. Your application will have a UDP/RTP and a UDP/RTCP port.
RTCP Packet Format taken from RFC 3550:

- SR: sender report, for transmission and reception statistics from participants that are active senders
- RR: receiver report, for reception statistics from participants that are not active senders and in combination with SR for active senders reporting on more than 31 sources

- SDES: source description items, including CNAME
- BYE: indicates end of participation
- APP: application-specific functions

We can measure the round-trip delay if we combine the NTP timestamp of a sender report and a receiver report.

3.3 RTSP

The Real Time Streaming Protocol RTSP is an application level protocol for controlling (start, stop, pause and position streams) a single or several time-synchronized streams of continuous media as audio and video. RTSP acts like a "network remote control" for multimedia servers. RTSP may use UDP or TCP (or other) as a transport protocol. RTSP is text based using the ISO 10646 character set in UTF-8 encoding. The most recent RTSP specification is published in RFC 2326

3.3.1 Communication between a Client and a Server

For a better explanation we will have a look at a streaming scenario. We were using the openRTSP client and a modified version of the testOnDemandRTSPServer of the live.com library. This will be an example for a media on demand unicast server. After starting the server chatRTSPServer music.mp3 we had the following stream :

```
rtsp://[ip-address]:[port]/[streamname]\newline
```

So we fired up the client like this :

```
openRTSP -V -e 10 rtsp://[ip-address]:[port]/[streamname].
```

The RTSPClient starts the conversation with the RTSPServer with a DESCRIBE request :

```
Sending request: DESCRIBE rtsp://129.102.24.7:7070/audiotest
RTSP/1.0
CSeq: 1
Accept: application/sdp
User-Agent: ./openRTSP (LIVE.COM Streaming Media v2004.03.04)
```

The RTSPServer receives the request and describe the stream with a SDP-description :

```
Received DESCRIBE response: RTSP/1.0 200 OK
CSeq: 1
Content-Base: rtsp://129.102.24.7:7070/audiotest/
Content-Type: application/sdp
Content-Length: 250
```

```
Need to read 250 extra bytes

Read 250 extra bytes: v=0

o=- 1080661979566079 1 IN P4 129.102.24.7
s=Session streamed by "testOnDemandRTSPServer"
i=audiotest
a=tool:LIVE.COM Streaming Media v2004.03.04
a=type:broadcast
a=control:*
t=0 0
m=audio 0 RTP/AVP 14
c=IN IP4 0.0.0.0
a=control:track1

Opened URL "rtsp://129.102.24.7:7070/audiotest",
returning a SDP description:
v=0
o=- 1080661979566079 1 IN P4 129.102.24.7
s=Session streamed by "testOnDemandRTSPServer"
i=audiotest
a=tool:LIVE.COM Streaming Media v2004.03.04
a=type:broadcast
a=control:*
t=0 0
m=audio 0 RTP/AVP 14
c=IN IP4 0.0.0.0
a=control:track1
```

The Client create a subsession with a rtp port and rtcp port and send a SETUP request :

```
Created receiver for "audio/MPA" subsession
(client ports 33548-33549)

Sending request: SETUP
rtsp://129.102.24.7:7070/audiotest/track1 RTSP/1.0
CSeq: 2
Transport: RTP/AVP;unicast;client_port=33548-33549
User-Agent: ./openRTSP (LIVE.COM Streaming Media v2004.03.04)
```

The Server send a positive SETUP response :

```
Received SETUP response: RTSP/1.0 200 OK
CSeq: 2
Transport: RTP/AVP;unicast;destination=129.102.24.7;client
_port=33548-33549;server_port=33550-33551

Session: 3
```

Setup of the subsession and requesting for PLAY :

```
Setup "audio/MPA" subsession (client ports 33548-33549)
Created output file: "audio-MPA-1"

Sending request:
PLAY rtsp://129.102.24.7:7070/audiotest RTSP/1.0
CSeq: 3
Session: 3
Range: npt=0-
User-Agent: ./openRTSP (LIVE.COM Streaming Media v2004.03.04)
```

Positive PLAY response received by the Client :

```
Received PLAY response: RTSP/1.0 200 OK
CSeq: 3

Session: 3
```

The Client starts to play the session receives data and after 10 minutes he sends a TEARDOWN request :

```
Started playing session
Receiving streamed data (for up to 10.000000 seconds)...
Data packets have begun arriving 1080662768695|

Sending request:
TEARDOWN rtsp://129.102.24.7:7070/audiotest RTSP/1.0
CSeq: 4

Session: 3
User-Agent: ./openRTSP (LIVE.COM Streaming Media v2004.03.04)
```

Client has got received the positive TEARDOWN response of the Server :

```
Received TEARDOWN response: RTSP/1.0 200 OK

CSeq: 4
```

In this scenario the media and the description is stored on one server. We do a DESCRIBE command on the RTSP server get the description of the media and set up the stream by the given description. We could imagine an architecture of several media servers containing different media data. For example a media server A containing audio data and a media server B containing the video media. To get the informations about the streams we could also have a web server containing the media description. This description contains in this example a "rtspurl" for the audio server and one for the video server and the relevant informations.

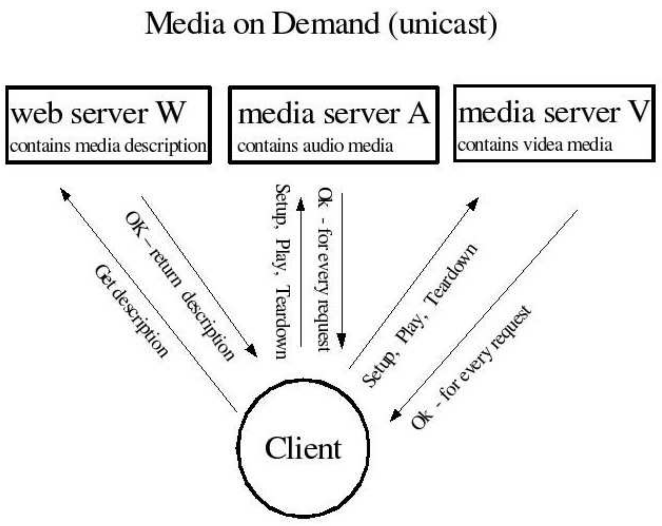


Figure 2: Media on Demand (unicast)

Above we described a RTSP unicast scenario. Media could also be presented using multicast. The media server choose the multicast address and port. The client after getting the "rtspurl" from a web server via, for example HTTP, is doing a DESCRIBE command on the server and get back the description of the session. The description contains the multicast address where to access the stream the ttl and the other stream specific informations.If this operation succeed the client is asking the server to SETUP the stream. The PLAY command will start the stream.

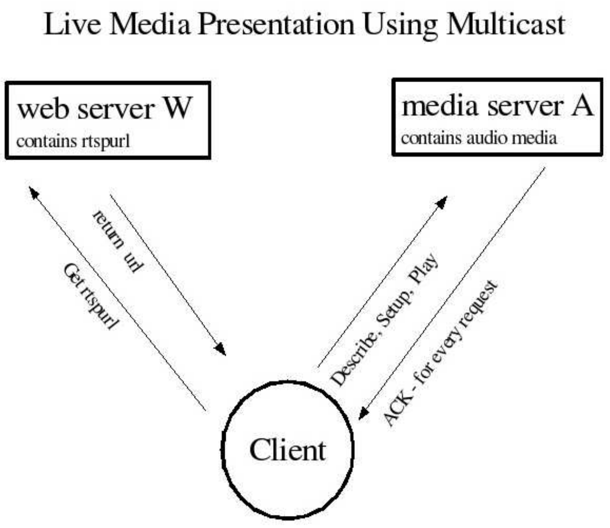


Figure 3: Live Media Presentation Using Multicast

The next picture shows an RTSP implementation in which a rtsp server streams to a multicast group. For example a participant of a conference would like to play back a tape stored on a RTSP media server to the other participants. This participant send a DESCRIBE command to the server and get a description of the media back. He does a SETUP of the stream with destination multicast address of the group. He starts the stream with the PLAY command and the members of that multicast group are able to listen to the tape.

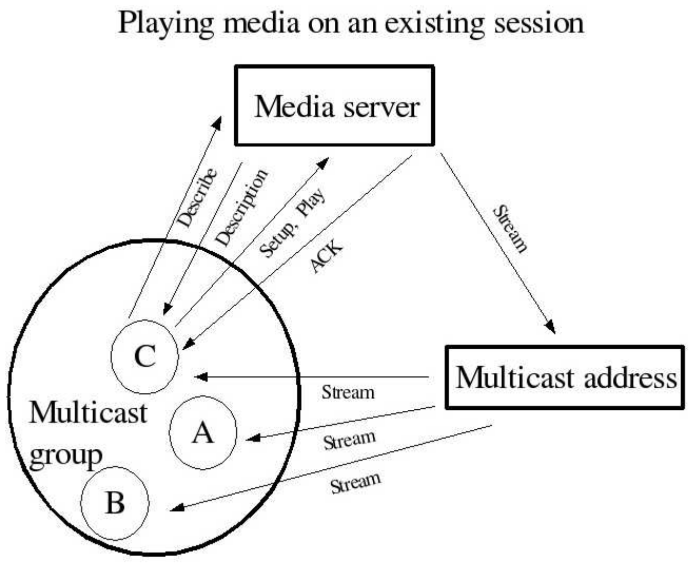


Figure 4: Playing media on an existing session

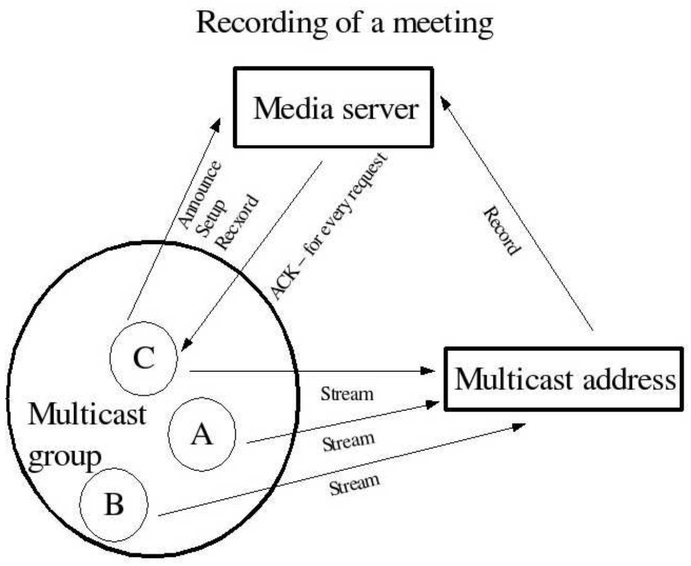


Figure 5: Recording of a meeting

In this implementation [figure 5] a participant asks the media server to record a multimedia meeting. The client uses the ANNOUNCE method to provide meta-information about the media to the server. For example if we got an audio and a video stream on different multicast addresses the client has to ask the server to SETUP two recordings. After setting up the RECORD command will start the recording.

3.4 SDP - Session Description Protocol

The Session Description Protocol [SDP] is used to advertise multimedia conferences and communicate conference addresses and conference tool specific information. It is also used for general real-time multimedia session description purposes. SDP is carried in the message body of SIP and RTSP messages. SDP is text based using the ISO 10646 character set in UTF-8 encoding. For better understanding we will have a look at an protocol extract send by a RTSP-Server implementation and received by a RTSP-Client. The Data streamed was a mp3-file.

```
The Description is in the form <type>=<value>.

v=0
(protocol version)
o=- 1080056361036718 1 IN IP4 129.102.24.7
(<username> it is '-' because the host does not support
                             the concept of user ids
<session id>is a numeric string <version>)
<network type> IN stands for Internet
   <address type> IP4 for IP protocol 4
   <address> from which the stream was created
s=Session streamed by ''testOnDemandRTSPServer'' (session name)
i=audiotest (session information)
a=tool:LIVE.COM Streaming Media v2004.03.04
<name and version of tool>
a=type:broadcast
<conference type>
a=control:*
t=0 0
<start time>  <stop time>
m=audio 0 RTP/AVP 14<media>
<port>/<number of ports>
<transport>
here : the IETF's Realtime Transport Protocol using the
       Audio/Video profile carried over UDP.
  <fmt list> media formats, payload type
c=IN IP4 0.0.0.0
(connection information - not required if included in all media)
a=control:track1
(zero or more media attribute lines)
```

SDP is standardized in rfc2327 and updated in rfc3266.

3.5 SAP

SAP is used by multicast session managers to distribute a multicast session description to a large group of recipients. A SAP announcer periodically multicasts an announcement packet to a well known multicast address and port as described below. A SAP listener learns of the multicast scopes it is within and listens on the well known SAP address and port for those scopes. SAP packets are UDP/IP packets with a SAP header with the IP address of the original source and the body contains the SDP session description. The following multicast addresses of the SDP/SAP block gives you access to SAP Announcements:

- for SAPv1 Announcements listen to 224.2.127.254 port 9875
- for SAP Dynamic Assignments 224.2.128.0-224.2.255.255

Difference to the SIP protocol:
The SIP protocol(Session Initiation Protocol) is used to invite an individual user to take part in a point-to-point or unicast session.

4 RTP library live.com

4.1 What library to take

When starting the project we made the decision to use the RTP protocol to transmit the audio data. We hoped to find a library which supports the protocols RTP, RTCP and RTSP to add this functionality to our application. We decided to start with a research on the available RTP libraries. In our research we paid attention to facts like project activity proved by implementations of the libraries in applications, traffic in the mailing lists and the latest release. Also important was code quality, the organization of it and accessibility via CVS. We made the decision to implement an open library so an open License for the library was quite logic. The research is divided into the two sections multimedia libraries and applications for the multimedia to get a first overview what was already developed as multimedia application[3].

4.2 Result of the research

In this paper we will only discuss the result of this research and will show the preferences of the chosen library (live.com). If you want further informations please have a look on the paper *Applications/Libraries for Multimedia* of Simon Schampijer. We have decided to take the live.com library to implement our application. Determinant was the fact that live.com is used in MPlayer and VideoLAN. This means trust in this library by well managed projects and this could be interesting for a video implementation. The Code organization is quite good and there is a good support by Ross Finlayson (mailing list), head of this project. All in all we think that this library could be a good base for our project[3]. The live.com library has also been packaged for several distribution Linux. Packages where found for these distributions.

[Fedora 1]

`http://atrpms.net/dist/fc1/live/`

[Fedora 2]

`http://atrpms.net/dist/fc2/live/`

[Red Hat 7.3, 8.0, 9.0]

`http://rpmseek.com/rpm-pl/live.html?hl=de&cs=live:PN:0:0:0:0`

[Mandrake 9.2, 10.0]

`http://rpmseek.com/rpm-pl/live.html?hl=de&cs=live:PN:0:0:0:0`

A packaging means that the software has already been tested, is easier to install and it is easier to make updates and to remove software.

Table 1: This code forms a set of C++ libraries for Multimedia streaming, using open standard Protocols.

LIVE.COM	
`Application/Library`	Library
`Web-Address`	http://www.live.com/liveMedia/
`Mailing-List`	live-devel@ns.live.com
`CVS`	No
`Project activity 1`	high
`Programming Language`	C++
`Code Size in lines`	Src = 34518, test = 3961, examples = 236
`Compilation chain`	self-made
`Code organization 2`	very good
`Packaged`	.tar.gz
`Code quality 3`	Lines Analyzed 34518, hits 450
`Latest release`	live.2004.03.11.tar.gz 11-Mar-2004 01:52 335K
`Protocol features`	RTP/RTCP, RTSP, SIP
`Is used in this Applications`	Mplayer, Vlc use the LIVE.COM Streaming MediaLibraries to implement RTSP/RTP
`Platform`	Unix (Linux, Mac OS X), Windows, QNX and other POSIX Systems
`License`	GNU LESSER GENERAL PUBLIC LICENSE Version 2.1

This library is used in Mplayer and Vlc to implement RTSP/RTP. This is very interesting because this prooves trust of other developers in this library and it's a good sign for being updated. Code organization is quite good, license is an open one and there is enough traffic in the mailing list.

1. evaluated by the traffic in the mailing list

2. well disposed (subdirectories)

3. tested with *Flawfinder 1.24*, nearly all the tested libraries gave the same result

5 Streaming of a sinus wave using the RTP protocol

Stream of a sinus transported via the RTP protocol supported by the live.com library. Implemented unicast on demand RTSP server.

5.1 Implementation

To test the live.com library our team decided to develop a test program that demonstrates how to stream a sinus via unicast RTP with built-in RTSP server. The RTSP server acts on demand which means he waits for a client controlling him. The application creates a RTSPServer object with a TCP port in our case the TCP port has got number 7070 and set up the stream. A stream is implemented using a "ServerMediaSession" object plus one or more "ServerMediaSubsession". A RTSP server can have several streams called ServerMediaSession in the live.com library. A ServerMediaSession has got a stream name in our case "sinus" and description of the stream like "This session is streamed by Alice". One Session can have several sub sessions. For example a MPEG-1 or 2 audio+video stream has got a audio and a video sub stream. After adding the session to the server we construct a "rtspurl" of our IP-Address the TCP port and the name of the stream. A address has got following form :

```
rtsp://129.102.24.7:7070/Sinus
```

In our Sub session L16ServerMediaSubsession derived from the FramedSource of live.com we have to set up our source (our sinus wave table, what we want to put into the RTP packets) and our sink (reads from the source and and transmit RTP packets). The Sub session is called L16ServerMediaSubsession because our "mime type" of the stream will be "L16" (linear 16bit) described in RFC 2586 and we will have a sampling frequency of 44100 and 2 channels. The result of these parameters will be a static RTP payload type of 10. These settings will be set in our RTPSink. Applications using live.com are event-driven, using an event loop TaskScheduler::doEventLoop()that works basically as follows:

```
    while (1) {
        find a task that needs to be done (by looking on the
         delay queue,and the list of network read handlers);
        perform this task;
    }
```

Whenever a module (sink or source) wants to get more data it calls "FramedSource::getNextFrame()" on the module that's to its immediate left. This is our source module. In the "dogetNextFrame()" method of our source class we read from our wave table and write it to the RTP packets. Once this is done we set the "fDurationInMicroseconds" which tells the RTP transmitting code how long to delay - after sending a packet - before asking for more data. To generate our sinus we will do a table-lookup synthesis. In our case the table length and the sampling frequency are fixed, the frequency of the sound depends on the value of the increment. The relationship between the given frequency and an increment is given by the following equation.

$$increment = \frac{L * frequency}{samplingFrequency} \tag{1}$$

If we suppose a table length L of 1000 and a sampling frequency of 40000 (even if this isn't common) while the specified frequency of the oscillator is 2000 Hz, then the increment would be 50. Equation for frequency :

$$frequency = \frac{increment * samplingFrequency}{L} \tag{2}$$

5.2 Test

We fire up a terminal and run our program:

```
>./SinusStreamer

"Sinus" stream, from a Wavetable""
Play this stream using the URL "rtsp://129.102.24.7:7070/Sinus"
```

If we have a look at the output of ”netstat -a -p — grep [pid-of the server]” (run in another terminal) we will see that we got a listening TCP/RTSP port. As we described above our RTSP server is a on demand server which listen on that port for a command of a RTSP client.

```
tcp 0 0 *:7070 *:*  LISTEN  17203/SinusStreamer
```

We fire up a third terminal and run the openrtsp program of live.com. This is a RTSP client which we will give the rtspURL of our server as argument.

```
>./openRTSP "rtsp://129.102.24.7:7070/Sinus"
Opened URL "rtsp://129.102.24.7:7070/Sinus",
returning a SDP description:

v=0
o=- 1091811586268887 1 IN IP4 129.102.24.7
s=Session streamed by "SinusStreamer"
i=Sinus
a=tool:LIVE.COM Streaming Media v2004.04.23
a=type:broadcast
a=control:*
t=0 0
m=audio 0 RTP/AVP 10
c=IN IP4 0.0.0.0
a=control:track1

Created receiver for "audio/L16" subsession
(client ports 34950-34951)
```

```
Setup "audio/L16" subsession (client ports 34950-34951)
Created output file: "audio-L16-1"
Started playing session
Receiving streamed data
(signal with "kill -HUP 17967" or "kill -USR1 17967" to terminate)...
```

The RTSP client demand a sdp description of the stream. With this sdp description the client creates a Media Session. It's the same Principe as we discussed already in the server architecture. The session is checked for sub sessions. With the found informations the client creates a rtpsource (an instance to treat incoming RTP packets) and a sink object(a destination where the received data can be written to). In the openrtsp implementation the sink object is a FileSink the data is written to a file. If we repeat the "netstat -a -p — grep [pid-of the server]" command now we will see that the server has created two UDP ports. One for RTP and the other for RTCP. These two ports are to transmit RTP packets between client and server. This connection is unicast. The TCP port 7070 still listen if another client will make a demand on the server. The second TCP/RTSP port opened by the server is two communicate with the first client.

```
tcp 0 0 *:7070                *:*                    LISTEN      17203/SinusStreamer
tcp 0 0 licorne.ircam.fr:7070  licorne.ircam.fr:44719  ESTABLISHED 17203/SinusStreamer
udp 0 0 *:34942               *:*                                17203/SinusStreamer
udp 0 0 *:34943               *:*                                17203/SinusStreamer
```

We suppose that a second client trying to connect to the server will evoke the same effect. So we will fire up another terminal and lunch a second client.

```
tcp 0 0 *:7070                *:*                    LISTEN      18202/SinusStreamer
tcp 0 0 licorne.ircam.fr:7070  licorne.ircam.fr:45241  ESTABLISHED 18202/SinusStreamer
tcp 0 0 licorne.ircam.fr:7070  licorne.ircam.fr:45240  ESTABLISHED 18202/SinusStreamer
udp 0 0 *:34954               *:*                                18202/SinusStreamer
udp 0 0 *:34955               *:*                                18202/SinusStreamer
udp 0 0 *:34960               *:*                                18202/SinusStreamer
udp 0 0 *:34961               *:*                                18202/SinusStreamer
```

As we can see in the result the RTSP server has got two unicast connections. Each connection has got a UDP/RTP and UDP/RTCP port for data transport and the control of it. Each connection has got also a TCP/RTSP port for the client to send controlling commands for the stream to the server. And the server got also the listen TCP/RTSP port if another client will establish a connection with the server. The RTSP client has got two UDP ports - one for RTP one for RTCP and a TCP/RTSP port which has got the unique port number with the TCP/RTSP port of the server. Otherwise the server couldn't identify commands of different streams.

```
tcp 0 0 licorne.ircam.fr:45240 licorne.ircam.fr:7070   ESTABLISHED 18203/openRTSP
udp 0 0 *:34952               *:*                                18203/openRTSP
udp 0 0 *:34953               *:*                                18203/openRTSP
```

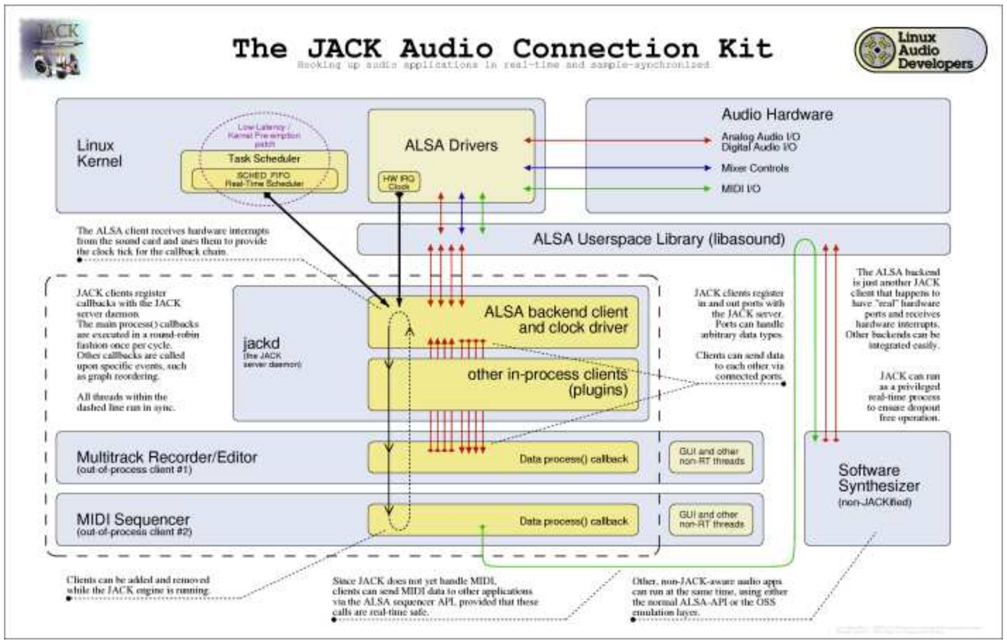
We think this first test shows very well the functionality of a RTSP server and a RTSP client. Also it describes very well the on demand unicast service of the server. With new every client connecting to the server the server opens a new RTP and RTCP port. We got a first impression of what sdp description means and the use of it. This implementation had also the side effect of getting familiar with the live.com library.

6 Jack Audio Connection Kit

JACK is free software and the JACK server uses the GPL license. You can download JACK at

`<http://jackit.sourceforge.net>`

JACK is a low-latency audio server, written primarily for the GNU/Linux operating system. It can connect a number of different applications to an audio device, as well as allowing them to share audio between themselves. Its clients can run in their own processes (ie. as normal applications), or they can run within the JACK server (ie. as a "plugin")[4]. JACK was designed from the ground up for professional audio work, and its design focuses on two key areas: synchronous execution of all clients and low latency operation. As an audio server JACK supports to interface with following sound drivers : alsa, dummy, oss or portaudio.


The JACK Audio Connection Kit
Hooking up audio applications in real-time and sample-synchronised
Linux Audio Developers
Linux Kernel
Low-Latency / Kernel Pre-emption patch
Task Scheduler
SCHED_FIFO Real-Time Scheduler
ALSA Drivers
HW IRQ Clock
Audio Hardware
Analog Audio I/O
Digital Audio I/O
Mixer Controls
MIDI I/O
ALSA Userspace Library (libasound)
jackd
(the JACK server daemon)
ALSA backend client and clock driver
other in-process clients (plugins)
Multitrack Recorder/Editor
(out-of-process client #1)
MIDI Sequencer
(out-of-process client #2)
Data process() callback
Data process() callback
GUI and other non-RT threads
GUI and other non-RT threads
Software Synthesizer
(non-JACKified)

The ALSA client receives hardware interrupts from the sound card and uses them to provide the clock tick for the callback chain.

JACK clients register callbacks with the JACK server daemon.
The main process() callbacks are executed in a round-robin fashion once per cycle.
Other callbacks are called upon specific events, such as graph reordering.

All threads within the dashed line run in sync.

JACK clients register in and out ports with the JACK server.
Ports can handle arbitrary data types.

Clients can send data to each other via connected ports.

The ALSA backend is just another JACK client that happens to have "real" hardware ports and receives hardware interrupts.
Other backends can be integrated easily.

JACK can run as a privileged real-time process to ensure dropout free operation.

Clients can be added and removed while the JACK engine is running.

Since JACK does not yet handle MIDI, clients can send MIDI data to other applications via the ALSA sequencer API, provided that these calls are real-time safe.

Other, non-JACK-aware audio apps can run at the same time, using either the normal ALSA-API or the OSS emulation layer.

Figure 6: Jack schematic Diagram

You can launch JACK by setting different parameter options. The first set of parameters are specific to run the JACK server and the second set are run time options for how JACK interfaces with the sound driver. Here the easiest way for running the JACK server (jackd) from the command line:

- *jackd -d alsa -d hw:0*, This runs JACK with the alsa driver and the first sound card in your system.

The default settings of JACK to interface with the sound driver are:

- a sample rate of 48000hz (number of samples per second)
- a buffer size of 1024 frames per second (determines the latency between when the sound is received by JACK and when it is send to the sound card)
- a period size of 2 (buffer rate which JACK will stream audio at)

If you aren't used to launch programs from the command line you can use the graphical user interface to control JACK called Qjackctl. For more information about setting the parameters and how to launch JACK please visit the excellent documentation at the site mentioned above. Let's have a look at the jack server with several clients connected. To give you a visual introduction we made a screen shot of the Qjackctl jack control application. In this example the jack

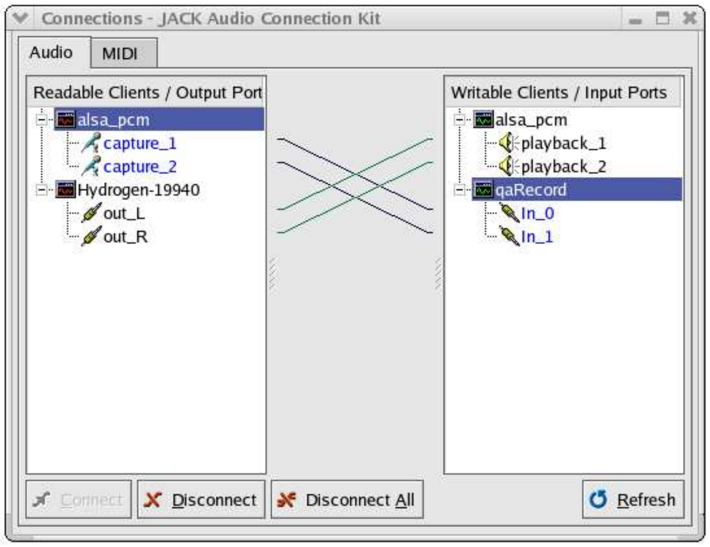


Figure 7: JACK clients

server is running (with the default settings as mentioned above) and we got two readable and two writable clients connected. The first readable client is the alsa_pcm capture device. This is the input of your sound card. Imagine you have got a guitar plugged into your sound card and would like to record your performance. This is why we connected the output ports of that client to the input ports of the writable client of qaRecord a simple sound recorder for ALSA and JACK. The sound is going out of the readable client into the writable client. This is the main concept of jack. Simple isn't it? Same for Hydrogen a pattern based drum machine which is a readable client - we can read the generated sound of that music application. We connect the output ports of that client with the input ports of the writable client alsa_pcm playback device which is the output of your sound card.

7 JACK - LIVE

7.1 JackLive a jack client RTSP server

7.1.1 Implementation

The next step was to develop a jack client. So we started to have a look at the great tutorial "Writing Audio Applications with JACK" of James Shuttleworth. These are the basic steps of creating a jack client:

create a new client (connection to the jack daemon) and try to become a client of the jack server,

set callback tell the JACK server to call "process()" when there is work to be done, JACK works with callbacks - requests us to do something by calling a given function

create ports create ports of the client - input or output (if your client is readable or writable)

get buffer get the memory area associated with the port - for an output port this is an area that can be written to, for an input port an area containing the data from the port's connection

activate client if you have done the setup the client is ready to roll

close client the work is done so close the client

After being familiar with JACK we had to think how our application should look like. Our application got two parts. The part which should be done by jack is to capture one or more audio input streams - audio streams are from any audio source hooked to your sound card's input jack's or any application which could be run as a jack client. Generating and sending RTP packets and to implement the RTSP server where our stream should be accessible is the part live.com should take care of. This means we have got two callbacks one called by the jack server to perform capture from the input ports and the callback of live called whenever a module wants to get more data - we can send another RTP packet. Between these two functions we need a structure which can be safely accessed by both. JACK offers a set of library functions to make lock-free ring buffers available. This is a short introduction by the ring buffer reference.

> The key attribute of a ring buffer is that it can be safely accessed by two threads simultaneously – one reading from the buffer and the other writing to it – without using any synchronization or mutual exclusion primitives. For this to work correctly, there can only be a single reader and a single writer thread. Their identities cannot be interchanged.[9]

The architecture of our application has got many similarities with the capture_client of the example-clients of JACK which captures audio input streams and record them as a wave file done in a separate processing thread. In our case the separate thread is the callback function of live. In fact this example client helped a lot to develop our client.

First we described a data structure accessible by both callbacks. This structure communicates information about the jack demon the ring buffer and control informations. We got the sample rate of the jack client, number of input channels, nframes a collection of audio values processed when the our process() function is called, a control value to indicate when the jack client is ready to roll(the ports are set up), one control value when live is ready to generate RTP packets and a pointer to the ring buffer. When starting the program the client try to connect to the jack daemon. For this you have to run the jack server. The number of input channels will be read as a command line argument. We register our process function which will be called at the right time by the JACK server and set a shutdown function to be called when the jack server shutdown. In this case we have to free our ring buffer and terminates the process which calls the function. We get the sample rate of the jack system, as set by the user when jackd was started (default 48000Hz), get the maximum size that will ever be passed to our process callback (nframes) and allocate a ring buffer data structure of a size we specified.

Now we start to set up our usage environment for simple, non-scripted, console applications needed by live.com. We have to initialize the event loop which search on the delay queue for tasks that need to be done and perform them. After this basic setup we create the RTSP server. This set up is the same than in our first application the sinus streamer. If this is done we begin by setting up our input ports. We register our ports with the flag *JackPortisInput* set, that the port can receive data. If the ports are made known to JACK we can set our *can_process* variable to true to indicate that our ports are set up probably. We can activate our JACK client now and start our event loop of live.com. The subsession is like in the first application derived from the FramedSource of live.com. Our L16ServerMediaSubsession have to set up our source and our sink. Our sink class has got a mime type of "L16", the payload format code will be 96 a dynamic RTP payload type if we run our application as default(jack daemon running at 48000Hz with two channels and 16 Bitspersample). The problem we face is that JACK likes to work in terms of floating point numbers. Samples turn out to be values between -1 and 1. Our team don't know about a payload type supporting 32bit floating point values. Also a multichannel implementation isn't previewed. We have to do a conversion from float(32 bit) to short (16 bit) to be able to put the data into RTP packets if the sink module wants more data and call our "dogetNextFrame()" of our source class. First question is:

How many bytes can we put into a RTP packet? In the live.com code it says that a RTP packet should always be no greater than 1400 bytes. The answer why it should be like that is the following : Ethernet normally has an MTU(Maximum Transmission Unit) of 1500 bytes. IP4 has an IP header of 20 bytes and IP6 of 40bytes. The rest is RTP-Header (12 Bytes), UDP-Header (8 Bytes) and the payload(the data). So how many samples we should read of the ring buffer. We said that we would like to met samples of the size of short into our packets and that a RTP packet has got the maximum size of

1400 bytes so we can put 700 samples into one packet. Now we check how many bytes are available in the ring buffer which we access via the pointer we declared in our structure accessible by both callbacks. If there are enough bytes to read to fill a packet we read the data convert it to short multiply by 32767.0 (don't forget JACK offers floating point values in the range of -1 and 1) every sample. There is still one subtility to handle. Bytes on our Linux system are represented in little-endian(the low-order byte of the number is stored in memory at the lowest address, and the high-order byte at the highest address) and the big endian order(opposite of little endian) also called network byte order is used if we want to send data. If the program gets this far the "fDurationInMicroseconds" which tells the RTP transmitting code how long to delay - after sending a packet - before asking for more data is set. If there is still enough data in the ring buffer to fill one packet we set it to 0, otherwise we calculate the time we have to wait. Now we have only to discuss the process() function to come to the part of testings.

Once again : "the process function is called by JACK when there is processing to be done". As parameters we are given nframes and our structure. Now we check if the both control variables are set so that we are able to start to process. If this is the case we start by getting the memory area associated for our input ports and give this to a pointer. From this area we read the data and write it to the ring buffer. We do this in a interleaved manner which means that we read from (input port n) a frame ,than a frame from (input port n+1), and this is done nframes time.

7.1.2 Test

The program can be run in a command-line shell given the number of channels (input ports of our client) as argument. It assumes that the JACK server is running. we have to start the JACK server.

```
>./JackLive 2
"JackLive" We do a Stream from the Jack Server""
Play this stream using the URL
"rtsp://129.102.24.7:7071/JackLive"
```

As we see in that picture we got the jack server running with "JackLive" as a writable JACK client and two input ports. Sound is generated by "Hydrogen" a pattern based drum machine. We have connected the output ports of "Hydrogen" with the input ports of our application. This is done with the help of the graphical user interface but is also possible by using command-line commands. At this time a RTSP client wasn't already developed. To access the stream we used "openrtsp" a command-line program that can be used to open, stream, receive and record media streams specified by a rtspurl. "openrtsp" is a test program of the live.com library. We open a new terminal and run "openrtsp" with given rtspurl of our stream.

```
>./openRTSP rtsp://129.102.24.7:7071/JackLive
Opened URL "rtsp://129.102.24.7:7071/JackLive",
```

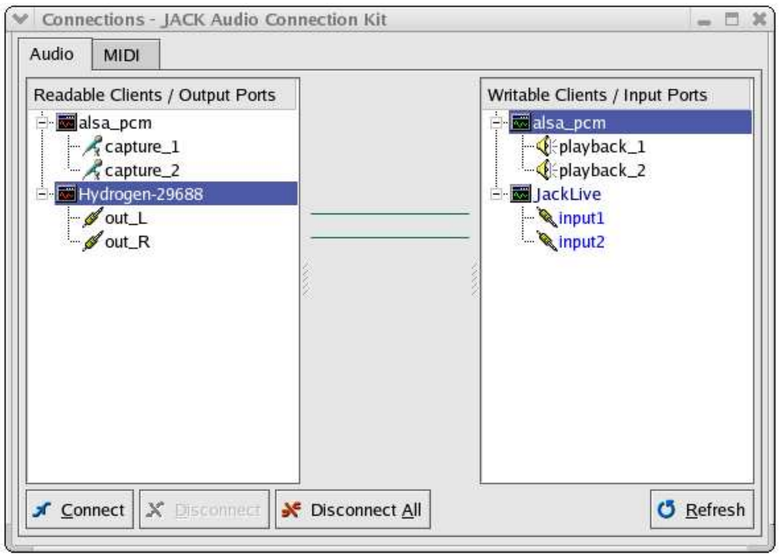


Figure 8: JackLive a jack client RTSP server

```
returning a SDP description:

v=0
o=- 1092119299994976 1 IN IP4 129.102.24.7
s=Session streamed by "JackLive"
i=JackLive
a=tool:LIVE.COM Streaming Media v2004.04.23
a=type:broadcast
a=control:*
t=0 0
m=audio 0 RTP/AVP 96
c=IN IP4 0.0.0.0
a=rtpmap:96 L16/48000/2
a=control:track1

Created receiver for "audio/L16" subsession
(client ports 35028-35029)

Setup "audio/L16" subsession (client ports 35028-35029)
Created output file: "audio-L16-1"
Started playing session
Receiving streamed data
(signal with "kill -HUP 394" or "kill -USR1 394" to terminate)...
```

If we have a look at the output of ”openrtsp” we see that we have Session with the stream name ”JackLive” with a payload type of 96 (16 bit, 48000Hz, 2 channels). After stopping the two applications we are curious to see what’s inside the stored file "audio-L16-1". To edit our sound file we use "snd" a powerful sound editing tool under Linux.

```
>snd audio-L16-1
```

If we choose little endian 16 bits as byte order for the created file we will hear noise coming out of our loudspeakers. We changed the byte order before sending the RTP packets and "openrtsp" just received those packets but didn't changed the byte order back. Fortunately we can change the byte order with "snd" to big endian (16 bits) so that we can hear our transmitted sound.

7.2 JackLiveClient a jack client RTSP client

7.2.1 Implementation

Having developed a RTSP server the next logic step was the developing of a RTSP client receiving the stream. As an example of how a RTSP client could be implemented with the live.com library we studied th "openrtsp" application of the test programs. These are the basic things a client must be able to do :

create a RTSP client object

OPTIONS request the server return the options allowed

DESCRIBE request the server return the sdp description

create media session from the sdp description with n subsessions

check for the number of channels to create the output ports for our JACK client

create RTP and RTCP Groupsocks on which to receive incoming data

create RTP source to handle incoming RTP packets

SETUP request setup the subsessions

create a sink object to encapsulate the data of the RTP packets

PLAY request to play the subsessions

We want to describe the sink object further. The sink module is called when there is new data(RTP packets). We write the frame(RTP packet) to a buffer as large as the largest expected input frame (1400 bytes the maximum size of a RTP packet). Now we have to for the available space in our ring buffer. Also in the client we got a shared element with the Jack Audio Connection Kit process. If we have enough size for 2 packets (we have to convert from short to float) we can start with changing the byte order to little endian. The samples have to be from short to float - how samples are represented in JACK. This is not enough because the range of floating point numbers in JACK is from [-1,1]. Also a lot of DSP packages expect them. So we have to do the following :

```
*float_sample = static_cast<float>(short_sample) * fScaler16;
```

where fscaler is defined as:

```
const float fScaler16 = (float)(1.0f / 0x7FFFL);
```

after doing all the changes and conversions the sample is ready to be written to the ring buffer. This + procedure has to be done for two RTP packets. The JACK part of the RTSP client is nearly the same than in the RTSP server. Create a new client, set callback, create ports, get buffer, activate client, and close the client when finished. The ports we want to register have to be output ports and the number of ports we got after extracting the number of channels from the sdp description. In our process function called by the JACK daemon we have to get the bytes available to read from the ring buffer and if we have got enough to write nframes to each channel we will proceed. The server has written interleaved data into the RTP packets and hopefully read also from the ringbuffer if we made all the conversions and byte order changes correctly. The process function can only proceed if the control variables indicate that the output ports of our JACK client are correctly set and the sink module is ready to proceed.

7.2.2 Test

To see if our implementation works correctly we will start the RTSP server on the laptop of the team. We use "Hydrogen" as audio source.

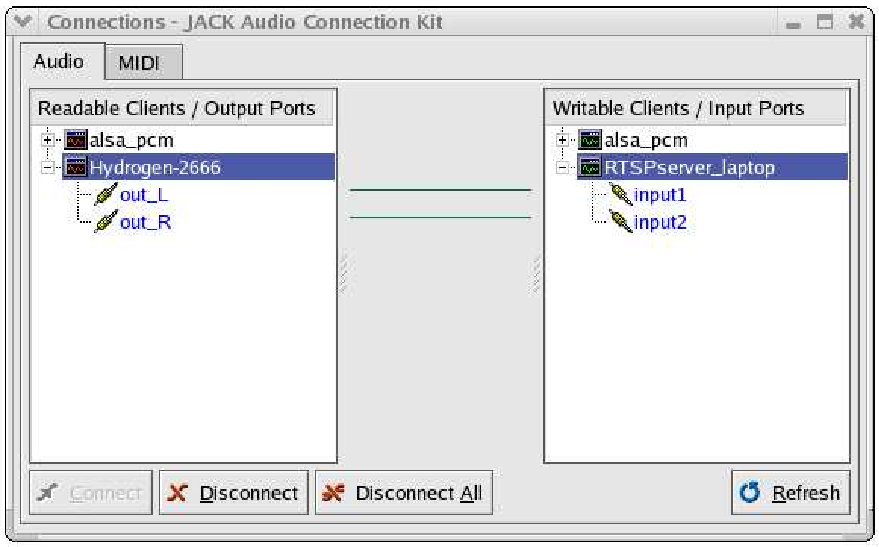


Figure 9: RTSP server on laptop

```
>./JackLive 2
 "Hydrogen_laptop" We do a Stream from the laptop""
Play this stream using the URL
"rtsp://129.102.24.10:7071/Hydrogen_laptop"
```

Now we want to access this stream by the given rtspurl with a RTSP client running on "licorne" another computer of the team. The output ports are connected with the input ports of the pcm playback device.

```
>./JackLiveClient rtsp://129.102.24.10:7071/Hydrogen_laptop
RTSP "OPTIONS" request returned:
OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE

We have to make 2 output-channels for the JackClient!
Created receiver for "audio/L16" subsession
(client ports 35070-35071)

socketbuffer "107520 bytes
Setup "audio/L16" subsession (client ports 35070-35071)
We got the Jack client activated!
Outputting data from the "audio/L16 to the JackRingbuffer!
We play the subsession now
```

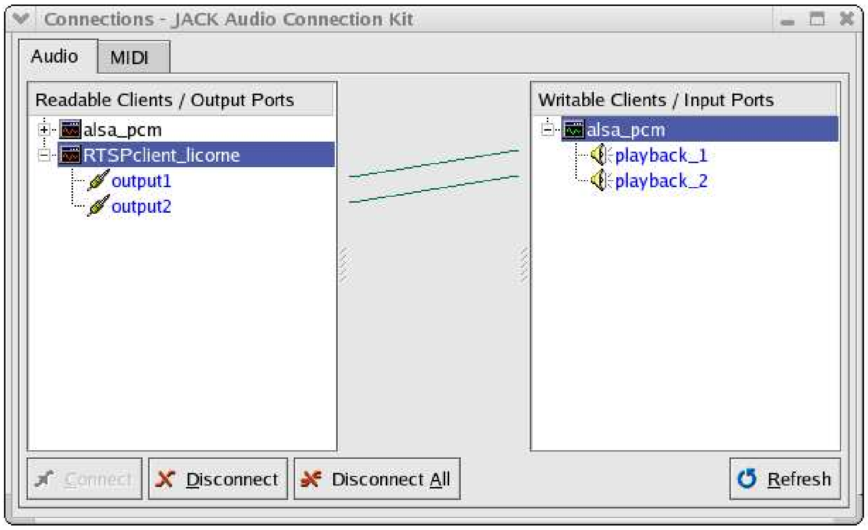


Figure 10: RTSP client on licorne

The next test will be to have a RTSP server running on licorne accessed by a RTSP client at the laptop and a RTSP server running on the laptop accessed by a RTSP client at licorne. Have a look at the pictures below if the description of this architecture confuses you. This is the RTSP server/licorne :

```
>./JackLive 2
"amsynth_licorne" We do a Stream from licorne""
Play this stream using the URL
"rtsp://129.102.24.7:7071/amsynth_licorne"
```

RTSP client on licorne playing the stream of the laptop :

```
>./JackLiveClient rtsp://129.102.24.10:7071/Hydrogen_laptop
RTSP "OPTIONS" request returned:
OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE

We have to make 2 output-channels for the JackClient!
Created receiver for "audio/L16" subsession
(client ports 35076-35077)

socketbuffer "107520 bytes
Setup "audio/L16" subsession (client ports 35076-35077)
We got the Jack client activated!
Outputting data from the "audio/L16 to the JackRingbuffer!
We play the subsession now
```

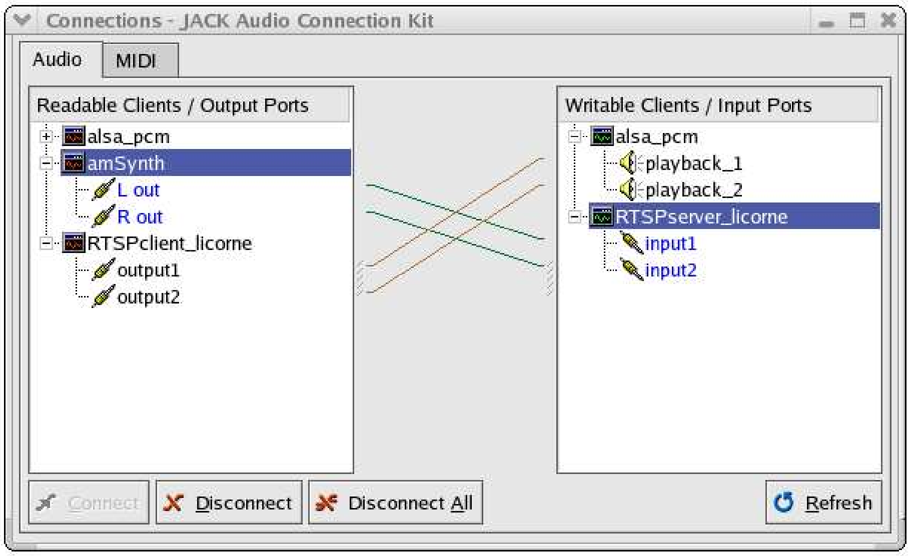


Figure 11: RTSP server and client on licorne

This is the RTSP server/laptop :

```
./JackLive 2
 "Hydrogen_laptop" We do a Stream from the laptop""
Play this stream using the URL
"rtsp://129.102.24.10:7071/Hydrogen_laptop"
```

RTSP client on the laptop playing the stream of licorne :

```
./JackLiveClient rtsp://129.102.24.7:7071/amsynth_licorne
RTSP "OPTIONS" request returned:
OPTIONS, DESCRIBE, SETUP, TEARDOWN, PLAY, PAUSE

We have to make 2 output-channels for the JackClient!
Created receiver for "audio/L16" subsession
(client ports 32774-32775)

socketbuffer "107520 bytes
Setup "audio/L16" subsession (client ports 32774-32775)
We got the Jack client activated!
Outputting data from the "audio/L16 to the JackRingbuffer!
We play the subsession now
```

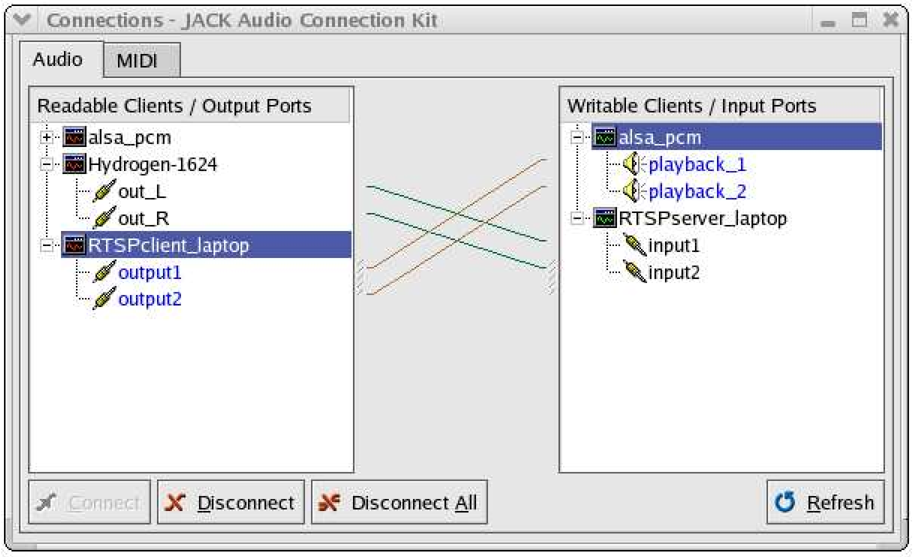


Figure 12: RTSP server and client on laptop

The result was satisfying. We had a good sound at both outputs. The delay seemed acceptable to our ears. To confirm our impression we decided to measure the round-trip-time in this implementation. For our test we had to prepare :

1. *a constant audio signal* we used ”Pure Data” a real-time music and multimedia environment developed by Miller Puckette to create a sigõbject. A sig object converts numbers to an audio signal. The constant 0.3 is given to the input of this object.

2. *a RTSP server running on licorne* with the input ports connected to the output ports of Pure Data and streaming to a rtspurl

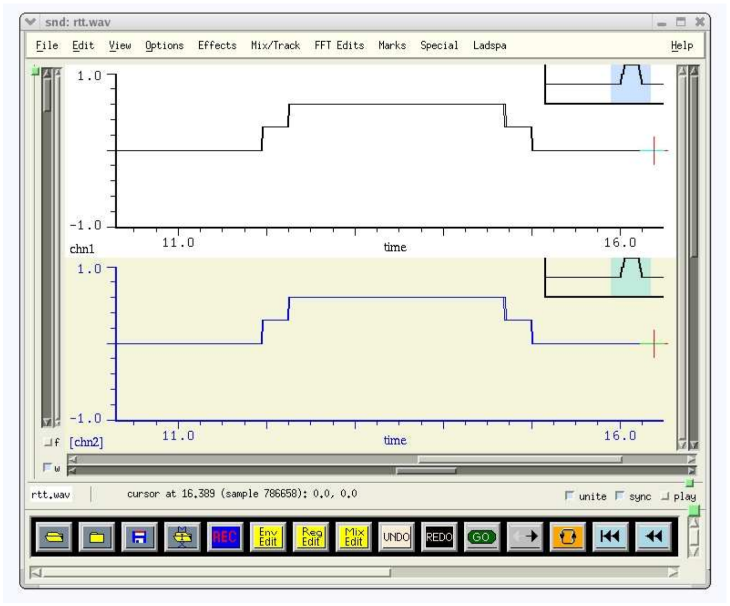
3. *a RTSP client receiving the stream of licorne on the laptop* play the stream of the server at licorne

4. *a RTSP server receiving the stream on the laptop* with the input ports connected to the output ports of the RTSP client

5. *a RTSP client receiving the stream of the laptop on licorne*

6. *record the result on licorne* we use "Qarecord" to record the signal generated by PD and the received audio signal by the RTSP client on licorne, the output ports of PD and the RTSP client are connected to the input ports of Qarecord

First we have to start to record. At this moment there is no audio signal recorded. After starting PD to send a signal Qarecord will receive the signal coming direct from the signal generator PD and after a moment receiving the signal made the round-trip from licorne to the laptop and back. After having received the signal we can stop PD and after a short delay also stop to record with Qarecord. Quarecord has created a 48000Hz wave file with our input. To have a look at our result we edit the file with "snd". In the picture we can see



cursor at 16.389 (sample 786658): 0.0, 0.0

Figure 13: Round-Trip-Time on the local net

that the signal received is 0 at the beginning and will move to 0.3 when the first signal the direct input of PD is received. The signal moves to 0.6 when the signal received from the RTSP client running on licorne is recorded. This is the second step in our picture. The 2 steps backward mark the time we stopped our signal. No we have to substitute the time at the first step from the time at the second step to receive the rtt.

This measurement was taken using the implementation described above. To see if the scheduling latency has got an influence of the global latency we tested different ways to run the JACK daemon. The scheduling latency is the time the JACK server delays to call the process() function of the jack client to proceed.

Table 2: Round-trip-time calculation - unicast on demand implementation

Scheduling latency JACK daemon	time signal direct	time signal after rtt in sec	round-trip-time in sec	one-trip-time in sec
42.7msec	11.941	12.251	0.310	0.155
21.3msec	4.556	4.906	0.350	0.175
	3.050	3.385	0.335	0.168
10.7msec	4.034	4.127	0.093	0.046 *
	6.244	6.448	0.204	0.102
	2.73	3.006	0.276	0.138
	2.569	2.843	0.274	0.137
	1.812	2.076	0.264	0.132
	4.501	4.810	0.309	0.154
	3.959	4.305	0.346	0.172
	2.888	3.217	0.329	0.164
	5.524	5.601	0.077	0.038 *

We changed the Frames/Period (default 1024) to achieve a lower latency. This was done on both computers "licorne" and the laptop.

Frames/Period = 1024 - 42.7msec (sample rate 48000, periods/buffer 2)

Frames/Period = 512 - 21.3msec (sample rate 48000, periods/buffer 2)

Frames/Period = 256 - 10.7msec (sample rate 48000, periods/buffer 2)

The latency vary from 132 msec to 175msec. We got several measures in this range. *How can we explain the two measures marked with a "*"?* The latency is really low in both of them. In fact they were taken after starting all the applications (RTSP server, RTSP client). We have also remarked that the value is ascending by every measure we take. We suppose that's because our buffer between the RTP part of our application and the JACK part is filled by the time and is stable after a while. Changing the scheduling latency didn't brought an remarkeble effect on the rtt. We suppose that we have to check our buffering technique to achieve lower latency introduced by the software.

8 Multicast

8.1 Background

Multicast is a technique developed to send packets from one location in the Internet to many other locations, without any unnecessary packet duplication. One packet is sent from a source and is replicated as needed in the network to reach as many receivers as necessary, instead of sending individual packets to each destination. The concept of a group is crucial to multicasting. A member of a multicast group transmits his packets to a multicast group signified by a multicast address and only the other members of the group can receive the multicast data. This means that you have to join a multicast group if you want to receive traffic that belongs to that group. While in broadcasting data are sent to every possible receiver multicast packets are only sent to receivers that want them. The multicast technique is interesting because it conserves bandwidth, what is in many cases the most expensive part of network operations. Using unicast a server would need to send out as many streams as there are receivers which increases also the CPU load. It is very economical if you want to send from one location to many other locations simultaneously which is the case in our application. But multicast also has got one major limitation. Every router between the recipient and the source must be multicast enabled if you don't want to do multicast tunneling which means multicast packets are encapsulated in unicast packets. But this won't offer a good performance as seen during the concert with the jMax streaming engine. Since multicast is still a ,in our opinion, not well supported technique not all networks are mulicast enabled. We hope that multicast will be ubiquitous soon.

8.2 Multicast Address

A multicast group has got one Class D multicast group address, we call it G. A group has got as described above more than one source (*,G). Each of this source has got a regular Class A,B or C Internet address, S. So if we say (*,G) this means every source in a multicast group address and (S,G) is one specific source in a multicast group address. But what is a Class A,B,C or D address? An IP-Address is an identifier for a computer device on a TCP/IP network. This address is used to route messages in a network to this destination. The format is a 32bit address written as four numbers separated by periods and a number has to be in the range of 0 to 255. An IP-Address has got a part to identify the network and a part that indicates the host within the network. This is a briefly overview of the different classes.

Table 3: The different Classes of IP-Addresses / the fields marked with a * are examples

	Use	binary address starts with	Net*	Host or Note*	% of available addresses
Class A	large networks	0	115.	24.53.106	1/2
Class B	medium-sized networks	10	145.24.	53.106	1/4
Class C	small-sized networks	110	195.24.53	106	1/8
Class D	multicast	1110	224.	24.52.106	1/16
Class E	experimental	1111	240.	24.52.106	1/16

Let's have now a further look at the Class D addresses the Multicast addresses. Class D addresses all have 1110 as the four first bits. Since 11100000 is decimal 224 and 11101111 is decimal 239, this means that all Internet addresses from 224.0.0.0 to 239.255.255.255 are assigned to multicasting. But you cannot use any multicast address that you want. The use of the multicast space is described int the RFC 3171 a guideline of IANA[7] for IPv4 Multicast Address Assignments.
The following part is taken from the RFC 3171[5].

```
Unlike IPv4 unicast address assignment, where blocks of addresses are
delegated to regional registries, IPv4 multicast addresses are
assigned directly by the IANA.Current assignments appear as follows
[IANA]:

224.0.0.0   - 224.0.0.255     (224.0.0/24)  Local Network Control Block
224.0.1.0   - 224.0.1.255     (224.0.1/24)  Internetwork Control Block
224.0.2.0   - 224.0.255.0                   AD-HOC Block
224.1.0.0   - 224.1.255.255   (224.1/16)    ST Multicast Groups
224.2.0.0   - 224.2.255.255   (224.2/16)    SDP/SAP Block
224.252.0.0 - 224.255.255.255               DIS Transient Block
225.0.0.0   - 231.255.255.255               RESERVED
232.0.0.0   - 232.255.255.255 (232/8)       Source Specific Multicast
                                               Block
233.0.0.0   - 233.255.255.255 (233/8)       GLOP Block
234.0.0.0   - 238.255.255.255               RESERVED
239.0.0.0   - 239.255.255.255 (239/8)       Administratively Scoped Block
```

Most users will use an address from the SDP/SAP Block, GLOP Block, the Administratively Scoped Block and the SSM Block. The SDP/SAP Block contains addresses used by applications that receive addresses through the Session Announcement Protocol [RFC2974] for use via applications like the session directory tool (SDR, Multikit). Administrative scoping is for internal use only. You can use this block within your own network for multicast sessions similar to the way that 10.0.0.0/8, 172.16.0.0/12, and 192.168.0.0/16 are used for private unicast addresses. It is described in RFC 2365. The "GLOP" address space 233/8 allows you to use multicast with your own autonomous system number(ASN). The ASN is a 16bit number and can be written as a binary number(left padded with zeroes), and then inserted into the middle two octets of this address block. The result is a unique /24 block of addresses for anyone with their own AS number. For example the AS number 1000 will gives you

[7]The Internet Assigned Numbers Authority

the address space of 233.3.232.0/24. This space could then be used by you for multicast addresses. Further descriptions can be found in RFC 2770.

The network layer service provided by SSM is a "channel", identified by an SSM destination IP address (G) and a source IP address S. A source S transmits IP datagrams to an SSM destination address G. A receiver can receive these datagrams by subscribing to the channel (Source, Group). This means that a specific source of a specific group can be directly accessed. In order to support SSM (Source Specific Multicast), a computer needs to support IGMP version 3. This leads us directly to the following subsection. To join a multicast group your computer has to send an Internet Group Management Protocol(IGMP) membership report message. IGMP is common to all multicast router protocols - isolates end users from the routing protocol in use. IGMP packets are always sent with a TTL of 1. IGMP v3 allows for specific channels joins and leaves through the addition of source specific INCLUDE reports to make SSM possible. Scoping means the restriction of multicast data transport to certain limited regions of the Internet. Administrative Scoping : The restriction of multicast transport based on the address range of the multicast group. It is defined by RFC 2365. TTL scoping : TTL means Time-To-Live and exist for unicast and multicast. The basic technique is similar for both. A computer sending a packet gives him a TTL of 8bit(0-255). The TTL is a count of the number of hops the packet is allowed. Every router which forward the packet decrement the TTL. If the TTL is 0 the packet won't be forwarded any more. Unicast uses this technique to prevent the packets from looping in the network because of a wrong configuration. The TTL field in multicast is set to prevent packets from leaving a particular domain. For example multikit the session announcement application of live.com gives following TTL's.

Table 4: TTL

same machine	local net	site	region	world
0	1	15	63	255

The last topic to discuss in this field is the multicast routing. We got two different modes. The *dense mode* assume that all possible receivers want multicast traffic initially. Multicast traffic is flooded throughout the network until receivers request to leave the multicast group. this technique is easy to implement but not very efficient as the size of the network increase. The *sparse mode* assume that that no receiver want multicast traffic until he hasn't asked for it. This protocol is much more usable for larger networks because there is no unwanted traffic. Today's standard for multicast routing is Protocol Independent Multicast - Sparse Mode or PIM-SM. It is called Protocol Independent Multicast because it uses the unicast routing information to route multicast data. PIM-SM is belonging to the shared-tree, a set of paths from the RP to the complete set of group members, family. We have got a RP (rendez-vous) point which maintains a table with the group and the source information. When a receiver wants to join a group, G, he sends a IGMP member report to its first hop router, which sends a (*,G) join to the RP. PIM-SM requires at least one RP per domain to function. Please have a look at RFC 2362 for further informations.

8.3 Possibilities offered to our Implementation

9 Implementation of Multicast

9.1 JACK client streaming to a multicast address

9.2 JACK client receiving from a multicast address

10 Future Works

10.1 Synchronization mechanism

10.2 Compensating clock skew

What envoques clock skew? The frequence of a sound card is given by a quartz. We consider that this frequency isn't constant - we have got a jitter. This jitter will become bigger if the quartz get hotter. For example we supose our sound card sampling with the frequence of 48000 the sampling frequency will maybe vary in a range of [47999-48001]. This range depends on the quality of the quartz used by your sound card. The first impression that this isn't very much vanish if we consider two audio cards running on two distributed computers. If we have got one sampling on a higher frequency the buffer of the other application is full...

-compensating -Synchronization -multicast conclusion -test multicast, finish test unicast

11 Conclusion

11.1 Reached Milestones in the project

1. *Properties of RTP* see [4]

2. *Overview of existing RTP librarys* see [5]

3. *Stream of a sinus using the RTP protocol* : using the live.com library, Implemented RTSP server

4. *JACK client streaming audio using RTP* : Implemented on demand unicast RTSP server

5. *JACK client receiving audio using RTP* : a RTSP client

6. *JACK client streaming to a multicast address* : using RTP as transport RTCP as control protocol

7. *JACK client receiving from a multicast address* : using RTP as transport RTCP as control protocol

8. *Testing* : in local network

9. *Created sourceforge project* : called jackrtp including jackrtp-send and jackrtp-receive

11.2 Todo

1. *Bug fixing* : latency added by the software

2. *Support of several sender* : for the multicast implementation, synchronization mechanism, dynamic allocation of ring buffers, dynamic allocation of JACK output ports

3. *Including an algorithm compensating clock skew* : compensate for the clock skew difference

4. *Test* : a concert inside the "renater" network, musical interaction of the musicians

1) The latency added by the software is nowadays to high. This can have several reasons. The result of timing the moment of writing audio data in RTP packets and the moment the RTP library is sending the RTP packets was in the range of 0.5 and 1 msec. Ross Finlayson of live.com confirmed that there are no special buffering techniques used to store data. Also the time our RTP source delay before asking for more data is set to 0 which should be right in our case of a "live" situation. So we suppose that the used RTP library don't add relevant latency. We also tested to change scheduling parameters inside the JACK daemon process. Changing the latency of calling the JACK clients by the daemon had no satisfied results. We suppose that our buffering technique used between the RTP library and the JACK client isn't sufficient developed.

This MUST be checked.
2) Next step is the support of several simultaneous sender. For that we need a synchronization mechanism based on RTCP. Determine the identity of the sender(SSRC) and determine new or leaving sender by control packets. A hash table structure of known members of the session is also included in the live.com library. The number of sender and number of ring buffers is equal, the number of output ports of our receiver must be twice because each sender got 2 channels. This allocation has to be dynamically.
3) We have to include an algorithm to compensate clock skew like described in the section before.
4) If these demands on our application are successfully accomplished musicians are able to try if our proposed musical tool is useful or not. After testing the network latencies inside the "renater" network (round-trip-time of 9-10ms) we suppose that a concert without adding a fixed latency will be possible. Because of being limited to people with access to this network the synchronization mechanism will still be useful. Anyway a feedback of musicians using this application will be the best way to improve their quality and right to existence.

The achieved results of the project and the research make us feeling happy for the future of the project.

11.3 Musical Aspect - performing with latency?

People often asked me for the musical aspect of the project. I think there are several. Imagine the musicians playing together with a fixed delay. As we have got already experiences with a concert with a fixed delay of one measure we will use it as scenario. This means we got a metronome synchronize the start of the session - each musician begin to play at the same time but is virtual distributed from the others. He send his stream to the monitor (which synchronize the streams) we got different delays for each musician. So we have to synchronize them with the maximum delay. The musician hears the monitored streams (also what he played) with one measure after playing it. These are the technical basics. Now we have got two cases to play together. The musicians have got a partition or they are improvising. In the first case playing together with a fixed delay if we are consider that the musicians are trained to play in such environments shouldn't be a problem. If we have got a given partition we can more or less play together without having a constant interaction between each other. This is a bit like a recording session - you don't play the music together in real time in case of interaction - the difference in our session is that you hear the result with a delay.Improvisation is a bit more difficult to handle in an environment with a delay. Percussion groups for example use sometimes "signal patterns" to call each other to change rhythms. Those signals could be used to work in such an environment. We can also say that the virtual distributed concert is preferred to some kinds of existing music, electronic music for example. Dj's using songs with long unchanging patterns could easily compensate a delay of one bar and there are a lot of other solutions which could be found.

Could we use that delay for new musical innovations? I discussed that project with several persons involved in music. The first idea was two write partitions while being aware of the delay. For example a choir performing a canon. The musicians could have different rhythms 3/4 5/4....

11.4 Social Aspect - playing on your own?

To feel yourself interacting you need a feedback. You can get a feedback of the other performers or of the auditorium traditionally by visual signs. On the one hand these signs are important to feel that you interact. You are doing something for someone. Even if some musicians affirm that they don't care about the response related to their musical experiment. During a concert the drummer play a tight beat and the first line of the listeners are starting to dance. This inspires the bassist to launch a grooving baseline making even the people at the bar moving their heads. Pushed by this success he is starting a solo and so on and so on.

Can we create such an environment in a distributed virtual concert? This kind of feedback could be possible using a video stream in real time of the auditorium received by every musician with the same delay than the synchronized streams of all the musicians. On the other hand visual signs are important to exchange conventions between the performers of how to play the musical piece. For example the bass player isn't confirm with the tight play of the drummer and want to play him more lay-ed back. So he will sign the drummer to change his play. Even if the band play in one room this can become difficult. Everyone being in a live situation know what it means if the other actors don't understand your signs. To resolve the problem of being distributed video conferencing techniques could help the musicians having communication. Some musicians mentioned that played already together in the same room could help to act in such an environment. The developed application could also be used to give virtual distributed lessons. If the teacher isn't able to be present in person he could teach his class at home or somewhere else. This leads to the point that the results of the project can be applied to other domains than musical applications.

11.5 Personal conclusion

The project "Virtual distributed concerts" is not a project easy to understand. The field of computer music is complex and I had to do a lot of research to understand the goals of the project. "The computer music tutorial" of Curtis Roads[x] and discussions with people inside of IRCAM especially François Déchelle and Patrice Tisserand helped a lot to get familiar with the subject. Also the subject of distributed architectures, more precisely on multimedia distributed applications over Internet demand a lot of studying. I spent a big part of my time on research of protocols for distributed multimedia applications and on multicast. In general I think there is still a lot of research to do and standards could help to clear misunderstandings and exclude undesired work. The demand of real-time behavior of our application didn't made it easier and can lead to short term brain confusion.

The fact of being a musician made it easier for me to feel the muss of musical interaction. This is a part of the project I had a lot of interest in. Working in a open source environment was really great for me. It's a very fast way of learning because you have got full access to sources and ideas of other people which exclude undesired work. From a scientific point of view I can't imagine to work different. Also the communication with other people of the open source community was a pleasure. Working in this project has brought a great evolution to me related to my academic and personal career.

11.6 Thanks

First of all I would like to thank the free software community for the great common work of millions of people and the gently, open minded behavior of many people involved in free software. This project is absolutely based on free software and would not have existed without it. Especially I would like to thank the developers of the Jack Audio Connection Kit and the live.com library on which great work this project is directly based on. The fact that those people made their software free permitted me to carry on their work and doing research in new areas. Big Thanks also to François Déchelle and Patrice Tisserand for their untiring efforts to explain the technical aspects of audio applications to me and the way they involved me into the free software team at IRCAM. I would like to decline with thanks IRCAM for hosting my scholarship and my Bachelor Thesis and the Fachhochschule Kiel for hosting my studies, especially Dr Helmut Dispert which was in charge of my Bachelor Thesis. And last but not least my mother, Déborah Damien, Michael Fourcade and all the people who filled me with new courage when I was tired or unsatisfied with my work ;)

11.6.1 Used audio software

References

[1] J.R. Cooperstock and S. Spackman : *The Recording Studio that Spanned a Continent* IEEE International Conference on Web Delivering of Music (WEDELMUSIC), Florence, 2001

[2] N. Bouillot: *The auditory consistency in distributed music performance: a condoctor based synchronization* ISDM (Info / com Sciences for Decision Making vol. 13(0), 2004

[3] N. Posse Lago and F. Kon: *The Quest for Low Latency* Department of Computer Science, University of Sao Paulo

[4] S. Cheshire: *Latency and the Quest for Interactivity* 1996

[5] IANA *IANA Guidelines for IPv4 Multicast Address Assignments RFC 3171* IANA August 2001

[6] Multicast Technologies,Inc:*Multicast Tech FAQs* ¡http://www.multicasttech.com/faq/¿ 2003

[7] S. Bhattacharyya *An Overview of Source-Specific Multicast (SSM) RFC 3569* July 2003

[8] Sprint *Sprintlink Multicast Frequently Asked Questions* ¡http://www.sprint.net/multicast/faq.html¿ 2001

[9] P. Davis and R. Drape*Linux Cross Reference, JACK/jack/ringbuffer.h* ¡http://jackit.sourceforge.net/cgi-bin/lxr/http/source/jack/ringbuffer.h¿ 2003

[10] M. Handley, C. Perkins and E. Whelan *Session Announcement Protocol RFC 2974* October 2000

[11] S. Schampijer: *Ptotocols for Multimedia* IRCAM, France, 2004

[12] S. Schampijer: *Applications/Libraries for Multimedia* IRCAM, France, 2004

[13] P. Shirkey: *JACK user documentation* ¡http://www.djcj.org/LAU/jack/¿ 2003

[14] C. Roads: *The Computer Music Tutorial* MIT Press, England, 1996

[15] N. Bouillot: *Transport du son produit en temps réel sur les réseaux best effort.* Grame France (2003)

VERBATIM COPYING

COPYING IN QUANTITY

MODIFICATIONS

COMBINING DOCUMENTS