What's new in JACK2?

Stéphane Letz, Nedko Arnaudov, Romain Moret

What's new in JACK2?

Stéphane LETZ
Grame
Centre national de création musicale
France
letz@grame.fr

Nedko ARNAUDOV
Sofia,
Bulgaria,
nedko@arnaudov.name

Romain MORET
PlayAll
France
rmoret@playall.fr

Abstract

JACK2 is the future JACK version based on the C++ multi-processors Jackdmp version. This paper presents recent developments: the D-Bus based server control system, NetJack2 the redesigned network components for JACK and profiling tools developed during port on Solaris.

Keywords

Audio server, D-Bus, real-time, networked audio, Solaris

1 Introduction

JACK2 is the new JACK version based on the C++ multi-processor Jackdmp version [1]. Jackdmp development started in 2005 with the goal to explore data-flow based scheduling of the client graph on multi-cores machines. A re-design on the server core was necessary and various improvements [1] have been implemented to remove some limitations on the initial model. D-Bus based control for the server appeared initially as a patch for JACK1 codebase.

Section 2 describes various internal build improvements in the code, section 3 presents the D-Bus services, section 4 describes NetJack2, and section 5 new profiling tools developed during the port on Solaris (Section 6). Additional developments are presented section 7 and ideas for the future in section 8.

2 Code restructuring

Code has been restructured for more flexible access to server side and client side code.

2.0.1 Control API on server side

A clean server control API has been defined to:

- get installed backend and get/set all their parameters.
- get installed internal clients and get/set all their parameters.
- get and set all server control parameters, start/stop the server.

The server side code is now compiled into a **libjackserver.so** dynamic library that exports the full client API as well as the new control API. JACK clients can possibly be linked to the libjackserver.so library and thus embed the JACK server in their process. For specific use cases, this can be especially interesting to create and distribute "stand-alone" JACK applications. The original **jackd** program as well as the new **jackdbus** server control program (see section 3.3) are linked to libjackserver.so. [2]

2.0.2 Server internal clients

Internal clients API has been slightly extended to allow Internal Clients to be configured and loaded using the control API. Using the D-Bus server control access, internal clients can be easily activated. The new NetJack2 design (see section 4.2) is extensively using this model.

3 D-Bus access

D-Bus is an object model that provides IPC mechanism. D-Bus supports autoactivation of objects, thus making it simple and reliable to code a "single instance" application or daemon, and to launch applications and daemons on demand when their services are needed.

3.1 Improvements over classical "jackd" approach

- Simplified single thread model for control and monitor applications. Various D-Bus language bindings make it trivial to

[1] lock-free programming techniques for graph access, "asynchronous/synchronous" server activation modes, two threads model for audio and notifications processing on client side...

[2] All backend (dummy, alsa, coreaudio, OSS...) are also linked to libjackserver.so

write control and monitor applications using scripting languages like Python, Ruby, Perl, etc..

- A log file is available for inspection even when autoactivation happens by the first launched JACK application.
- A real configuration file is used to persist settings to be manipulated through configuration interface of JACK D-Bus object.
- Improved graph inspection and control mechanism. JACK graph is versioned. Connections, ports and clients have unique (monotonically increasing) numeric IDs.
- High level abstraction of JACK settings. Allows applications that can configure JACK to expose parameters that were not known at compile (or tarball release) time. Recent real world examples are the JACK MIDI driver parameters and support for FFADO driver in QJackCtl. Upgrading of JACK requires upgrade of QJackCtl in order to make new settings available in the GUI.

3.2 How it works

3.2.1 Autoactivation and starting/stopping JACK server

First, application that issues D-Bus method call to JACK controller object, causes D-Bus session daemon to activate the object by starting the **jackdbus** executable. Activating controller object does not start the server. Instead controller object has several interfaces. The most important of them is the control interface. Control interface contains methods for starting and stopping JACK server, loading and unloading of internal clients (netjack), setting buffer size and resetting xrun counter. It also contains methods for querying information required by monitoring applications: whether JACK server is started, whether JACK server is running in realtime mode, sample rate, DSP load, current buffer size, latency, xrun counter.

JACK server autostart is achieved by libjack calling "jack server start" method of JACK control D-Bus interface.

3.2.2 JACK settings

Applications that want to manage JACK settings can query and set all settings that were traditionally specified as **jackd** command-line parameters. Interface abstraction provides virtual tree of parameter containers with container leaves that contain parameters. Parameters are accessed using simple addressing scheme (array of strings) where address defines path to parameter, like "drivers", "alsa", "dither".

Overview of the tree of standard settings' addresses:

- "engine"
- "engine", "driver"
- "engine", "realtime"
- "engine", ...more engine parameters
- "driver", "device"
- "driver", ...more driver parameters
- "drivers", "alsa", "device"
- "drivers", "alsa", ...more alsa driver parameters
- "drivers", ...more drivers
- "internals", "netmanager", "multicast_ip"
- "internals", "netmanager", ...more netmanager parameters
- "internals", ...more internals

JACK settings are persisted. I.e. they are automatically saved by **jackdbus** when they are set. Next time user starts JACK server, last saved settings will be automatically used.

Changing JACK settings through the configure D-Bus interface takes effect on next JACK server start. On the fly change of the buffer size, as available in the libjack (jack.h) API, is also possible through the control D-Bus interface.

3.2.3 JACK parameter constraints

JACK internal modules that provide parameters visible through control API can provide information about parameter valid range (like realtime priority) or whether parameter should be presented as enumeration. Enumeration parameters can be strict and non-strict. Example of strict enum parameter is dither parameter of ALSA driver, it has only predefined valid values - "shaped noise", "rectangular", "triangualr" and "none". Example of non-strict parameter is device parameter of ALSA driver. It is useful to provide some detected device strings as choices to user, but still allow user to specify custom string that ALSA layer is supposed to understand.

3.2.4 JACK patchbay

In order to simplify patchbay applications, extended functionality is provided. There is a method that returns the current graph state. Graph state has unique monotonically increasing version number and contains info about all clients, their ports and connections. Connections, ports and clients have unique numeric IDs that are guaranteed not to be reused. Notifications about graph changes are provided using D-Bus signals.

3.3 JACK D-Bus enabled applications

- JACK contains "jack_control" executable - a 300 lines of Python exposing JACK D-Bus functionality. It allows chained execution of several commands. For example **jack_control ds alsa dps midi-driver raw eps realtime on eps relatime-priority 70 start** selects ALSA driver, enables JACK MIDI raw backend, enables realtime mode, sets realtime priority to 70 and starts JACK server.
- LADI Tools is a set of programs to configure, control and monitor JACK . It provides tray icon, Window Maker style dockapp, G15 keyboard LCD display integration application, configuration utility for managing JACK settings and log file monitor application. All tools are written in Python.
- Patchage, the ubiquitous canvas modular patch bay can be compiled to use D-Bus instead of libjack to communicate with JACK . Doing so also enables JACK server start/stop functionality in Patchage.
- LASH, recent developments of the audio session handler by default use D-Bus to communicate with JACK . Various JACK related features are planned:
  - Saving of JACK settings as part of "studio" session.
  - Handling of "JACK server crash" scenario: restarting JACK server, notifying JACK applications that JACK server reappeared so they can reconnect to it, and restoring JACK connections.

3.4 Portability

While D-Bus implementations exist at least for Windows and OSX platforms, D-Bus is not a central part of their desktop environments and thus using **jackdbus** on those operating systems will probably cause more problems than it is supposed to solve.

4 NetJack2

4.1 Introduction

NetJack2 is a feature allowing the use of JACK2 over a local network. It is not really a port of NetJack to JACK2 but can be seen like a refactoring of the main idea: send and receive audio, midi or transport data over a network while staying in the Jack real-time audio context. The differences between NetJack and NetJack2 are mainly about the global architecture and network processing (how to slice data into network packets and how to optimize the network bandwidth use).

4.2 Architecture

NetJack2 is designed around a Master/Slave architecture involving two components. The master is an internal client (see Section 2.02) and the slave is a classical JACK server running the *Net Backend*. Those two components are built as two dynamic libraries, **netmanager.so** and **jack_net.so** (or .dll under Windows), which are both linked with the main JACK server library. This architecture allows several slaves to be captured and running under a single master without any other need of configuration.

- **Master:** the master is a classical JACK server, driven by an audio backend (ALSA, CoreAudio, OSS or PortAudio) and where an internal client called the *NetManager* is running. This *NetManager* is just a 'logistical' component which just creates and removes classical JACK clients in the server as the Slaves appear and disappear.
- **Slave:** the slave is a classical JACK server, driven by the *Net Backend*. This specific backend isn't controlling an audio device, that means the backend's read/write operations aren't executed on some audio hardware, but on a network interface (wireless is not supported because it doesn't really offer real-time networking capabilities).

4.2.1 Multicast communications and NetManager

NetJack2 is based upon the principle that two computers must be able to 'connect' themselves without knowing the parameters by advance. In NetJack2 , slaves are fully configured by the master. The global initialization process is thus quite simple:

1. The slave starts multicasting it's availability on the network by communicating its capabilities (number of available channels) to anyone who wants to hear it
2. The master catches the message and gives back to the slave a full set of parameters (samplerate, backend buffer size...)
3. The slave receives it and starts the stream exchange
4. Bi-directional communication is now running, and the slave is fully synched on the master's incoming network stream

Lets see how this exchange begins...

Multicast communications This first initialization step is done over a Multicast group, thus the slave doesn't have to know the master's address, or even configuration. A newly incoming slave just has to send its 'availability' message to the Multicast group, and if a master is listening to this Multicast group, it will automatically starts the transmission.

NetManager component The NetManager component is dealing with the initialization phase. The NetManager doesn't manipulate audio or midi signals, it just listens to the Multicast group and creates or destroy Masters as Slaves come and go. This component is seen by JACK as an internal client, that means this client is loaded in the JACK server context, using the specifically designed API and utility (jack_load executable). But it can also be loaded and managed by the jack_control system.

4.3 Real-time networking

NetJack2 Provides real-time network capabilities. Audio (or midi or transport) data use network streams as if those streams were classical 'jackified' applications. The main difficulty of such a concept is to handle real-time networking. Connecting two computers in a simple network implies some constraints: packets losses, random transmission delay, routing issues, disconnection timeout management etc. Reducing those constraints to keep a real-time transmission means compromises.

First, we can't afford a total secured transmission, using secured protocol such as TCP for instance. TCP can not be used as a real-time protocol because it involves acknowledgment for each sent packet, automatic re-emission in case of packet loss, what automatically discard such a protocol for a true real-time audio transmission. That's why NetJack2 uses a very simple way to exchange streams: the UDP protocol. NetJack2 main principle is very simple, so the best way to handle it is to choose a quite low level layer of network communication protocols, and just add the few things we need to it. By using UDP, we just get a very simple and highly configurable way to exchange data.

The second main aspect is to consider time. In a network transmission, transmission delays are reflected by the bandwidth notion. The bandwidth is the amount of data we can transmit in a certain amount of time. For real-time transmission, that means the higher the bandwidth is, the quickest we receive our data on the distant computer. Increasing bandwidth means reducing transmission delay. That principle is very simple and linear: a gigabyte networking infrastructure is ten time faster than a 100mb network. The bandwidth use optimization appears as a major aspect to take care of. That's why NetJack2 splits data into packets that are maximum sized (the maximum size of a packet is given by the network MTU). Dealing with larger packets increase the network's speed use, but it also minimizes packets losses.

The third aspect of real-time networking is the fact that a network has a very randomized timing behavior, which is not only depending on the two connected computers. Routers, firewalls and other networking devices add random unpredictable transmission delay. In real-time audio, we can't wait longer than a given amount of time (one audio cycle for example). NetJack2 extensively use the timeout notion on all its internal networking transmission. If we consider we can't wait for data over than a cycle length, the best way to keep synced in time is to skip data that were not received on time.

4.4 Network modes and latency

NetJack2 doesn't have one only functional mode. It actually has three 'networking' modes: fast, normal and slow. What those modes control is in fact the total time latency the network will add. The use of one mode or another is in fact depending on what the user wants to do.

4.4.1 Fast mode

In fast mode, the whole system (composed by the two connected computers) will not add extra latency. In a cycle number n, the master will send its data, the slave is supposed to receive it, produce its own data, and send it back to the master in the same cycle. Everything happens just as if the slave was a classical JACK client plugged into the master's JACK server. The fast mode is designed for fast network, with a small amount of data to exchange (only a few audio channels for example), and a small amount of processing on the slave'side, because data are expected in the same audio cycle (that means we consider the network transmission time to be significantly less than the cycle length). The fast mode can be quite unstable because it uses a long timeout, allowing NetJack2 to use 100% of the available CPU time.

4.4.2 Normal mode

The normal mode takes into account the network, and add one extra-latency cycle, corresponding to the use of the full available bandwidth. The master send its data in the current n cycle, and expect return data from the slave in the n+1 cycle. Don't forget the slave is a backend, and as a backend, it has two 'processing' modes: sync or async. Thus, in async mode, the slave will send the previous 'n-1' cycle data just after receiving the current n cycle. That means in this mode (async), we just don't take into account the slave's processing time and data are sent with no additional delay. The normal mode use a small timeout, considering data has to be available very quickly (because of the extra latency cycle). This mode doesn't 'block' JACK processing while waiting for data.

4.4.3 Slow mode

The slow mode adds two extra-latency cycles. This mode has been introduced for an extensive use of the network with a lot of processing running on the slave. It can be very useful in a multitrack production context, because transport (play/stop, position), which is also sent in NetJack2 , includes latency management, so the whole network will start 'simultaneously' (that means the 'fastest' JACK client will start in the cycle within which all data are available.

4.4.4 Conclusion

We can resume those three mode quite simply:

- only a few channel to exchange on a fast network, the fast mode can be very pleasant because it doesn't add extra-latency
- normal use of the network (up to 48 channels at 48kHz on a 100mb network), normal mode will guarantee a very small amount of packet loss (probably no loss if the network is stable)
- huge use of network and processing: the slow mode is the most 'secured' (the master doesn't wait for the slave's return data before two cycles, giving the slave the network the time to transport data and the slave the time to process it)

4.5 Measurements

JACK can be compiled with the possibility of monitoring NetJack2 networking activity, thus allowing accurate measurements about what's going on between the master and the slave. Those measurements are a pretty good way to understand what is behind the different available networking modes.

This profiling system keeps a track of several timings, and the results can be exploited to get an approximation of NetJack2 's CPU consumption.

Because those measurements are quite uneasy to dissect and explain, more explanation will be available on the JACK wiki.

4.6 Adapters components

Adapters components allow to *adapt* the network stream synced on the master machine clock, in order to be played on the slave machine.

4.6.1 AudioAdapter

When the slave machine is using the *Net Backend* and if the user wants to listen to the network stream on the slave audio card, the network stream has to be "adapted", since the two audio cards may run at different buffer size, sample rate and their clocks are typically not synchronized. This is done using an in server client called *audioadapter*, which ajust the master sample rate and buffer size to the slave audio card sample rate and buffer size. Audioadapter components are using an intermediate ring-buffer and the *libsamplerate* library [3] to resampled the stream when needed. They have been developed on each platform using the available native audio API: CoreAudio on OSX,

[3]http://www.mega-nerd.com/SRC/

ALSA on Linux, PortAudio on Window and OSS an Solaris.

4.6.2 NetAdapter

When the slave machine is using a regular audio backend and if the user still wants to receive a stream from the network, the *netadapter* in server client can be used. This component has basically the same behaviour as the *Net Backend* but also adapt the master sample rate and buffer size to the possibly different slave audio card sample rate and buffer size.

4.7 NetJack2 conclusion

NetJack2 is an easy way to exchange audio, midi or transport control data over a network in a realtime context. This component is working so far, and it will be improved in the future. NetJack2 is currently available with JACK2 , and a few documentation can be found on the JACK wiki[4].

5 Profiling tools

Timing measurements allow developers to help understanding the behaviour of their JACK based applications. The server code base now allows to record various timing while the server and clients are running and generates scripts to interpret them.

While running, timestamps are taken for the server and running clients: *wake-up date* of audio driver, *activation*, *actual wake-up* and *end date* of each running client. Client scheduling latency and duration are also computed. They are saved as a *JackEngineProfiling.log* file containing time points when the server is closed. Scripts for Gnuplot are also generated. Use the command: **gnuplot -persist Timing1.plot** with Timing1.plot up to Timing5.plot. The scripts also generate PDF files. A **jack_profiler** internal client allows to see all measures as *audio signals* to be analyzed or recorded with additional tools.

5.1 Audio driver interrupts

Timing1.plot allows to display the audio driver timing, that is the duration between consecutives interrupts.

The JACK2 server can run in two different graph scheduling mode: in synchronous mode, the audio cycle is composed of: *read audio input buffers, execute the graph, write audio output buffers*. In asynchronous mode on the contrary,

[4]trac.jackaudio.org/wiki/WalkThrough/User/NetJack2

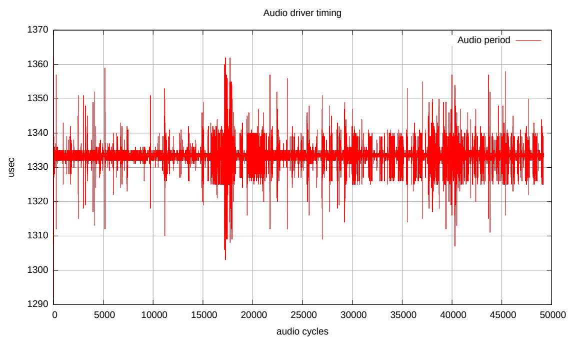

Figure 1: *Audio driver interrupt, at 48 kHz and 64 frames, average is 1333 usec, interrupt period is regular, with small variations of about +/- 30 usec. (Vertical axis shows duration in usec and horizontal axis the number of audio cycles.)*

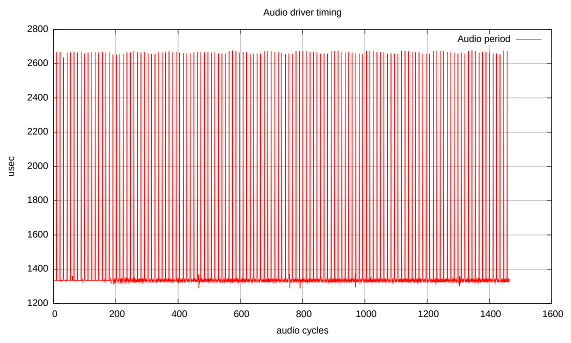

Figure 2: *Audio driver interrupt, at 44.1 kHz and 64 frames, average is 1451 usec, but interrupt period is not regular*

the audio cycle is composed of: *read audio input buffers, write audio output buffers computed at previous cycle, execute the graph.* When the interrupt period is regular, then the server *asynchronous* mode can be safely used (fig 1). On the contrary a non regular driver interrupt force to *synchronous* mode to be chosen (otherwise the graph may lack time to finish its execution if duration between 2 consecutive interrupts is too short) (fig 2).

5.2 Driver end date

Timing2.plot allows to display the audio driver end date. This measure is interesting to distinguish what happens in server synchronous versus asynchronous mode. In synchronous mode, the driver end date takes the graph execution duration in account (fig 3). [5] In asynchronous

[5]The graph here shows several clients launched one

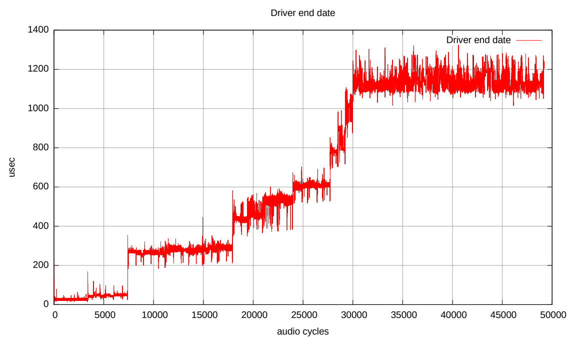

Figure 3: *Driver end date when the server runs in synchronous mode: the date raises each time a new client is launched.*

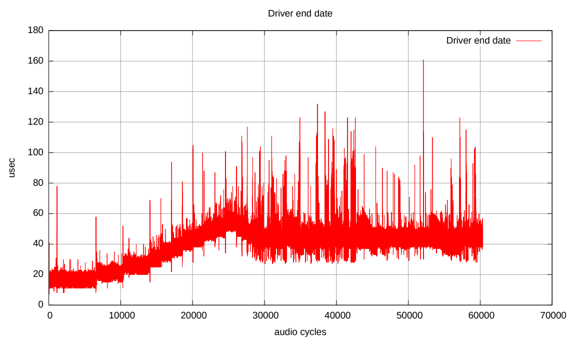

Figure 4: *Driver end date when the server runs in asynchronous mode: the date stay approximatively constant when new clients are launched, since it only depends of cumulated read and write durations.*

mode, the driver does not wait for the graph end, but returns immediately after the write step. Thus the driver end date measures the *read audio input buffers*, *write audio output buffers* parts only (fig 4).

5.3 Clients end date

Timing3.plot allows to display the audio driver timing and all clients end date. This curve gives a global view of all clients DSP use. The audio interrupt duration is displayed as well as the end date of all running clients. The system works correctly if the end date of the last client is still below the audio interrupt duration (fig 5).

5.4 Clients scheduling latency

Timing4.plot allows to display all client scheduling latencies (difference between activation and actual wake-up dates). When the application

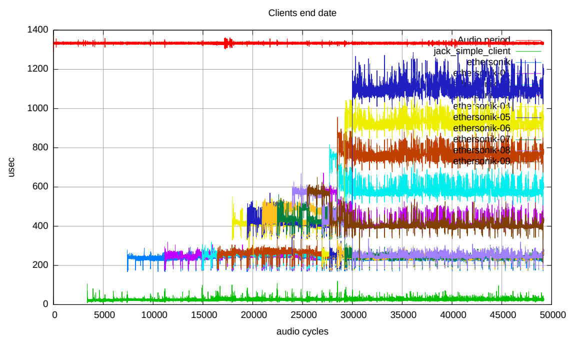

Figure 5: *Clients end date*

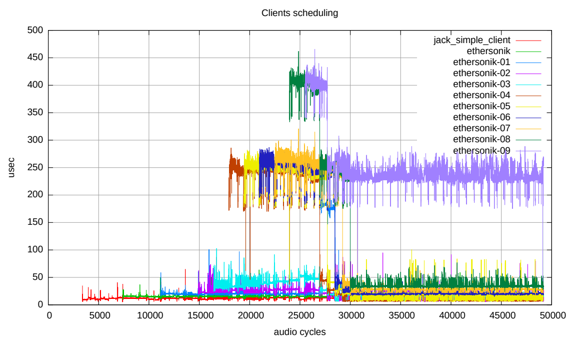

Figure 6: *Clients scheduling latency*

real-time thread becomes runnable, the global scheduling latency depends on the fact a core is effectively available in the machine and the actual OS scheduling latency. Thus this value obviously depends of the topology of the graph and number of available cores on the machines on a given setup. To precisely measure the OS scheduling latency only, the best is to have a number of clients that is less than the number of available cores (fig 6).

5.5 Clients duration

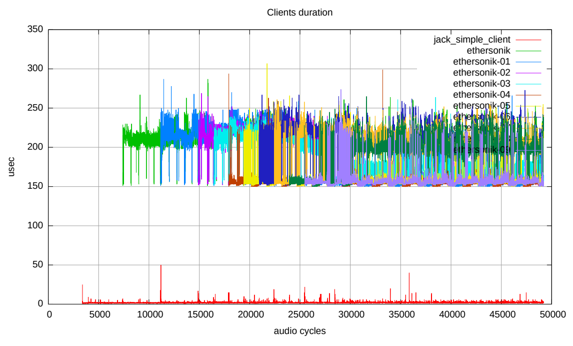

Figure 7: *Clients duration*

by one, the driver end date then raise after each new client is started.

Timing5.plot allows to display all client duration (difference between end date and actual wake-up date) (fig 7).

5.6 Real-time measurements

The **jack_profiler** internal client allows to see all measures as *audio signals.* It can be loaded at anytime using the jack_load tool. JACK output ports will be created and contain timing measures converted in audio signals. At each audio cycle, the measured value is recalibrated to be in the [-1,1] range and copied into the audio buffer (same value for the entire cycle). Three general output JACK ports can be added (respectively using the *-c, -p, -e* parameters in the jack_load parameter line):

- **profiler:cpu_load**: is the DSP CPU load between 0 and 1.
- **profiler:driver_period**: is the driver period variation expressed at the difference between the actual driver period and the expected driver period, then divided by the expected driver period, between -1 and 1.
- **profiler:driver_end_time**: is the driver end time expressed as driver end time divided by the driver period, between 0 and 1.

For each running client, two additional JACK ports will appear:

- **profiler:"client_name":scheduling**: is the client scheduling duration expressed as the scheduling duration divided by the driver period, between 0 and 1.
- **profiler:"client_name":duration**: is the client duration expressed as the client duration divided by the driver period, between 0 and 1.

6 Solaris port

The french radio RTL aims at developing it's future digital radio using a JACK based system, hosting a set of audio applications: playlist manager, audio effects (compressor...), real-time signal analysis components (speaker recognition for instance), encoding and web broadcasting... on a Solaris system, when most of their audio files data base management tools are already running. The timing measurements tools have been developed to characterize the real-time behaviour of this system in several setups. The set of curves in this paper have been done on a Dell XPS 420 Intel 4 cores machine running Solaris 10 and using the internal HD Audio card using OSS 4.0 version. Fig 1 shows the audio driver interrupts when the server is running using a 64 frames buffer size at 48 kHz, in synchronous mode and using the highest scheduling priority that can be obtained (using the following command: **jackd -R -S -P 59 -d oss -p 64).** Then fig 2 to 5 shows the different curves obtained with a highly loaded machine. Using the profiling tools we were able to conclude that:

- It appears that the OSS driver does not guaranty a perfectly regular audio interrupt period, depending of the driver settings. Thus the *synchronous* (-S) mode should be preferably chosen when starting the server.
- Solaris defines several scheduling classes. The Real-Time scheduling class goes from 100 to 159 on a global 0 to 169 scale. The corresponding POSIX priority to be set in JACK will go from 0 to 59 (-P59 for highest priority).
- The Solaris has quite good Real-Time capabilities [2], and we found that using Processor sets allows to decrease the RT threads scheduling latencies and have a more predictable system. We measured a maximum scheduling latency of 80 usec in this configuration.

7 Additional developments

7.1 Better Windows port

On Windows, the JACK code base was initially compiled using Microsoft VC++ tools. All projects have been converted to use the free CodeBlock and MinGW [6] environments.

7.2 Testing and debugging tools

Different tools have been implemented to help developers when coding JACK based applications. Profiling tools has already been presented in section 5. Two more are available:

7.2.1 Checking JACK API

The correctness of JACK API can be tested using **jack_test** tool. This program will open JACK clients, test various aspects of the API: registering ports, setting callbacks, activating the client... Real-time communication between

[6]see www.codeblocks.org and www.mingw.org

two clients is also tested as well as Transport API.

7.2.2 Checking use of JACK API

When launching JACK applications, a **JACK_CLIENT_DEBUG** environment variable can be set [7] to launch the client in debug mode: all calls to the API are traced and a log file is generated. Correct use of the API will be checked (opened clients should be closed, activated clients should be deactivated, port registration/unregistration matches is verified...)

8 Blue sky for singing penguins living on the planet JACK

The future could (and probably should) try to solve some problems that are getting bigger with increasing popularity of JACK :

- There can be better handling of misbehaving JACK clients. Is it wise to allow clients to change parameters that will disrupt JACK operation? Today, obnoxious clients auto-connect without allowing configuration. Tomorrow, what if a client thinks it must reset the sample rate or the period size because of cutting corners in the dsp code? The Access Control Lists (ACL) looks like obvious and traditional solution but applying it to JACK domain may lead to quite complex implementation because of authentication that needs to preceed even simplest authorization checks.
- Developing a control application based on OSC could be quite interesting, allowing access and control of any JACK server on the network.
- MIDI backends: interfacing JACK MIDI with the native MIDI API on each plarfom (CoreMidi on OSX, ALSA Midi on Linux and WinMME on Windows) has to be done.

9 Acknowledgements

Juuso Alasuutari has significantly contributed to **jackdbus** for JACK1 and Marc-Olivier Barre contributed to **jackdbus** by making it use XDG compliant directories.

Work on profiling tools and Solaris port has been funded by RTL french radio (EDIRADIO).

[7] using "export JACK_CLIENT_DEBUG = on"

References

[1] Stephane Letz, Dominique Fober, and Yann Orlarey. *jackdmp: Jack server for multi-processor machines.* Linux Audio Conference, 2005.

[2] J Litchfield. *Real-time in the Operating Environment Solaris 8.* http://www.opengroup.org/rtforum/oct2000 /slides/litchfield.pdf, 2008.