JACK AUDIO SERVER: MACOSX PORT AND MULTI-PROCESSOR VERSION

S.Letz, D.Fober, Y.Orlarey
Grame - Centre national de création musicale
letz, fober, orlarey@grame.fr

P.Davis
Linux Audio System
paul@linuxaudiosystems.com

ABSTRACT

Jack is a low-latency audio server, written for POSIX conformant operating systems such as GNU/Linux. It can connect a number of different applications to an audio device, as well as allowing them to share audio between themselves. We present the port for Apple's MacOS X, and new developments to take advantage of multi-processor architecture.

1. INTRODUCTION

There is a long history of specialized systems designed to simplify the development and use of musical applications. In the MIDI domain, MIDI Manager, Opcode MIDI System and MidiShare [3] on Macintosh and more recently the ALSA sequencer [1] on Linux have been designed to allow separate MIDI applications to run together, share the hardware MIDI input/ouput, and exchange MIDI events in real-time. The main idea behind these approaches is to simplify the task of developers by providing a library of commonly needed services and let users configure their system in a very flexible manner by using several separated components than can run and collaborate together.

In the audio domain, specialized API are available on each operating system: for example CoreAudio on MacOSX, DirectSound and ASIO on Windows, ALSA on Linux. Some projects like PortAudio [2] provide a uniform and more abstract access to all these API. Other ones like the aRts sound server [5] on Linux provides sharing of audio I/O and some form off application collaboration though the MCOP Multimedia Communication Protocol allowing multimedia applications to be network transparent.

None of them has been designed to allow several applications to collaboration in real-time on the same machine. Jack design focus on two key areas: synchronous execution of all clients, and low latency operation. Jack is a low-latency audio server written by Paul Davis and other developers for POSIX conformant operating systems with two main goals:

- for programmers, the Jack API provides a high level abstraction by removing the details of audio interface hardware access. Using a callback based model, applications can send and receive audio streams from/to each other as well as to the audio interface.
- for users, using the Jack system allows a more flexible way of working: instead of using a "monolithic" general purpose heavy application, users can build their setup by having several smaller and goal focused applications that collaborate, dynamically connecting them to meet their specific needs.

For both programmers and users, the Jack system allows important new functionality to be added to the system as a whole, without it being added to each application. Two examples might include:

- live network audio streaming: this can be handled by a special client that accepts data from other Jack clients and sends it to a streaming server. This allows all Jack compatible applications to gain streaming functionality without any modification to their source code.
- integration of external transport/sync signals a special client can be written that understands, for example, ADAT sync, and maps it into the Jack transport API. This allows all Jack transport-aware applications to sync to the external signal, again with no modification to their source code.

As Jack has evolved, it has also come to include a shared transport system, allowing multiple applications to start/stop/locate with sample-accurate synchronization.

The section 2 presents the Jack system, section 3 describes the port on MacOSX, and finally section 4 describes the multi-processor version.

2. THE JACK SYSTEM

2.1. Model

Jack provides a high level abstraction for programmers that removes the audio interface hardware from the picture and allows them to concentrate on the core functionality of their software. It allows applications to send and receive audio data to/from each other as well as the audio interface. Most significantly, there is no difference in how an application sends or receives data regardless of whether the target/source is another application or an audio interface.

For programmers with experience using several other audio APIs such as PortAudio, Apple's CoreAudio, Steinberg's VST and ASIO as well as many others, Jack presents

a familiar model: the program provides a "callback" function that will be executed at the right time. The callback can send and receive data as well as do other signal processing tasks. The application programmer is not responsible for managing audio interfaces or threading, and there is no "format negotiation": all audio data within Jack is represented as 32 bit floating point values.

For those with experiences rooted in the Unix world, Jack presents a somewhat unfamiliar API. Most Unix APIs are based on the read/write model spawned by the "everything is a file" abstraction that Unix is rightly famous for. Despite the many merits of this model, it has allowed too many application developers to avoid the real-time nature aspect of their software's operation. The result has been a combination of large scale buffering of data, leading to greatly increased latency for audio applications, along with audio dropouts when that buffering is still inadequate to deal with OS delays or poor software design.

In addition, the Unix read/write model makes it more difficult to facilitate inter-application audio routing, because it tends to result in different programs are running asynchronously; that is, one program is processing data corresponding to one particular time interval, while another program is working on a different time interval. This situation does not lend itself to sharing data in a low latency system.

2.2. Code example

Using Jack API within a program is very simple, and typically consists of the following steps:

- connecting to the Jack server.
- registering "ports" to enable data to be moved to and from the application.
- registering a "process callback" which will be called at the right time by the Jack server.
- telling Jack that the application is ready to start processing data.

A simple audio thru client could be described with the following code:

```
#include<jack/jack.h>

jack_default_audio_sample_t * out;
jack_default_audio_sample_t * in;
jack_client_t* client;
jack_port_t* input_port;
jack_port_t* output_port;

int process(jack_nframes_t nframes, void *arg)
{
   out = (jack_default_audio_sample_t *)
     jack_port_get_buffer(output_port,nframes);
   in = (jack_default_audio_sample_t *)
     jack_port_get_buffer(input_port,nframes);
   memcpy(out,in,
     sizeof(jack_default_audio_sample_t)*nframes);
   return 0;
}
```

```
int main(int argc, char *argv[])
{
   client = jack_client_new("foo");
   input_port =
     jack_port_register(client,
                        "input",
                        JACK_DEFAULT_AUDIO_TYPE,
                        JackPortIsInput,
                        0);
   output_port =
     jack_port_register(client,
                        "output",
                        JACK_DEFAULT_AUDIO_TYPE,
                        JackPortIsOutput,
                        0);
   jack_set_process_callback(client,process,0);
   jack_activate(client);
   sleep(30);
   jack_client_close(client);
   return 0;
}
```

More complex programs have a larger API to draw on if they wish, but many interesting applications will need nothing more than these steps.

2.3. Internals

2.3.1. Introduction

Jack system is built around several components: a server, a driver and several clients (Fig 1). Since Jack clients will typically be separated applications, the system has to be able to transfer data (like audio buffers) between different processes, activate them when needed and possibly notify them when global state changes occur.

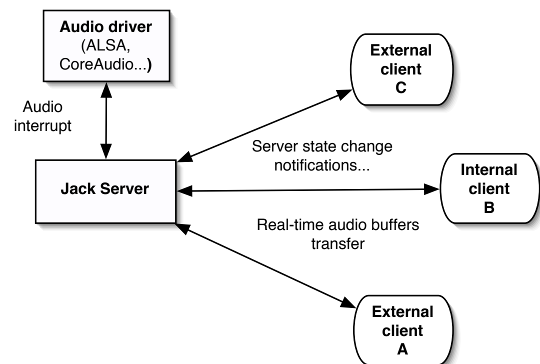

Figure 1. *Archictecture of Jack server/client system*

2.3.2. Server

Jack is based on a server/client model.[1] The Jack server is the center of the system. It interacts with the driver, and communicates with all registered clients. Triggered by the driver, the server activates the client graph, a set of connected "nodes", each of which must be "executed" on a periodic basis. In the case of Jack, the graph is made up

[1] The following description reflects the state of the Linux implementation.

of Jack clients, and each one has its *process* function to be called in a specific order. The connections between each node may take any configuration whatsoever. Jack has to serialize the execution of each client so that the connections represented by the graph are honored (e.g. client A sends data to client B, so client A should execute before client B). In the event of feedback loops, there is no ”correct” ordering of the graph, so Jack just picks one of the legal possibilities.

Data within a Jack graph is shared between clients (and the server) using shared memory. Each ”output port” owned by a client has a shared memory buffer into which the client can write data. When an ”input port” is connected to the output port, reading from the input port simply uses the shared memory buffer. This permits zero-copy semantics for audio processing in many simple scenarios, and minimal copying even in complex ones.

2.3.3. Driver

The whole graph is executed synchronously by a driver which interacts with the hardware, typically waking the server at regular intervals determined typically by its buffer size. The server then ”distributes” this audio interrupt to all running clients. The basic requirement for the system proper functioning is that the server and all clients do their job, including server/client communications, audio data transfer and processing between two consecutive audio interrupts (for example with a buffer size of 128 frames at 44100 Hz, this represent a 3 ms duration).

2.3.4. Clients

Clients dynamically register to the server, and establish connections between themselves. Clients can be internal, running in the server process, or external. Since the driver that controls the audio interface presents itself as just another client, sending data to and from the audio interface is identical to sending it to and from any other client.

Each client will be woken by the engine when it is time to operate on its data. Great care has been taken to use the most efficient server/client communication scheme. On Linux, POSIX FIFO’s are used, which have been shown to be the fast Linux blocking IPC mechanism until the introduction of futexes in kernel version 2.6 (see 4.3). On MacOSX, low level IPC mechanisms are used for the same purpose and reason.

Jack contains an important optimization in its design for waking clients, a step that requires a full context switch to another process. On systems with virtual memory (i.e. any hardware on which running Jack might make sense) the cost of a context switch does not scale with CPU speed, and this switch represents the primary overhead of Jack compared to entirely in-process plugin systems like VST and AudioUnits. Jack works hard to reduce the total number of context switches by chaining clients - rather than switch from the server to a client and then back to the server before moving on to the next client, we instead have each client wake the next one in the graph.

2.3.5. Client library

Applications access the server through the client library: it contains the client side of a Jack application, takes care of server/client communication, and exposes the API available for programmers.

Clients must implement a real-time safe *process* function. In other words its code must be deterministic and not involve functions that might block for a long time. The general form used to describe RT-safety for Jack purposes is: cycles = (A * nframes) + C, that is, the time to execute the *process* function must be a direct function (A) of the number of frames of audio data to be processed, combined with some constant overhead (C). Profiling code is constantly checking the system behavior in the server:

- too big kernel scheduling latencies or client graph process overloading is notified to the clients as *xruns*
- too slow clients during several consecutive audio cycles are usually removed from the graph.

3. MACOSX PORT

3.1. Porting the Linux code on MacOSX

Since Jack code is ANSI C POSIX, the initial port of the Linux code mainly required to adapt some missing API for MacOSX. Some specific adapdations were later done to yield a more efficient implementation:

- The Jack server audio cycle has been re-designed for a callback based activation scheme, where the audio cycle is directly triggered by the driver.
- MacOSX low-level communications between user space and kernel are based on Mach ports and messages. When developing server/client systems, new function calls between the server and clients can be defined using the *Mach Interface Generator*. Remote Procedure Synchronous Call are particularly efficient since they by-pass the scheduler.

  The Linux client activation model has been adapted to take advantage of MacOSX fast Remote Procedure Call facility: all clients are directly activated from the server audio cycle using a synchronous RPC.
- On MacOSX, real-time threads are actually time-constraint threads defined by a *period*, *constraint* and *computation* parameters. These parameters have to be carefully choosen to allow the system to correctlly interleave thread computation, especially when threads are running with different period values.

3.2. Integration into the CoreAudio architecture

Having the Jack API available on MacOSX simplifies the port of Jack Linux based applications. But audio applications on MacOSX use the standard CoreAudio API and

can not be directly used with the "native" Jack applications. A layer has been developed to allow the transparent use of CoreAudio applications with Jack applications, based on several components (Fig 2):

- the *Jack Audio Router* driver (JAR) is a user-space driver that behaves like a "bridge" between regular CoreAudio applications and the Jack server. The JAR driver is a Jack client and a CoreAudio driver at the same time. For CoreAudio applications, it presents the Jack model as a *n mono non interleaved*, *fixed buffer size* and *fixed sample rate* driver.

  The CoreAudio API has to be "mapped" on the Jack API by translating or adapting all CoreAudio functions call into the corresponding Jack functions. By accessing this driver, CoreAudio applications become Jack clients and can take advantage of Jack capabilities.

- CoreAudio applications can be extended using a *plug-in* model: audio effects or synthesizer for example, can be dynamically loaded and activated in the application. Jack plug-in components are available to extend the routing capabilities. *JackAU* (AudioUnit plug-in format) and *JackVST* (VST plug-in format) have been developed.

- *JackPilot* is a configuration application to setup global parameters, manage audio connections between Jack clients and save/restore global state.

A typical setup where native Jack clients and "Jackified" CoreAudio applications run under control of the Jack server is presented here:

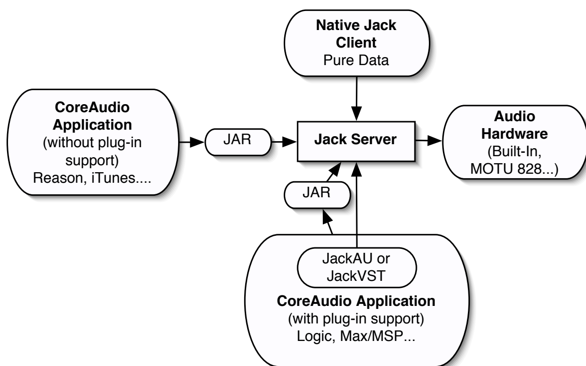

Figure 2. *Typical setup including native Jack applications and CoreAudio "Jackified" ones (CoreAudio applications hosting plug-ins can access the Jack server using JAR and/or Jack plug-in)*

3.3. Availability

Jack is an open-source project hosted at SourceForge (jackit.sourceforge.net) and collectively developed by the Linux audio community.

Jack on MacOSX is an open-source project hosted at SourceForge with its home at www.jackosx.com. All MacOSX specifc components are developed thanks to the help of Johnny Petrantonni and Dan Nigrin and are distributed in a freely available binary package called JackOSX.

4. MULTI-PROCESSOR VERSION

4.1. Current activation model

In the current activation model (either on Linux or MacOSX), knowing the data dependencies between clients allows to sort the client graph to find an activation order. This sorting step is done each time the graph state changes, for example when connections are made or removed or when a new client opens or closes. This order is used by the server when activating clients.

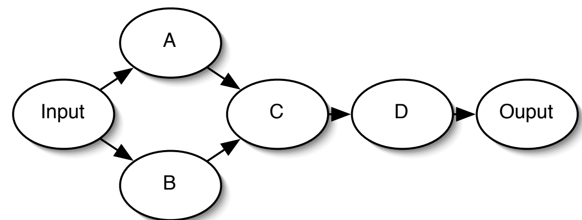

Figure 3. *Client graph: Client A and B could be executed at the same time, C must wait for A and B end, D must wait for C end.*

Forcing a complete serialization of client activation is not always necessary: for example clients A and B (Fig 3) could be executed at the same time since they both only depend of the "Input" client. In this graph example, the current activation strategy choose an arbitrary order to activate A and B. This model is adapted to mono-processor machines, but cannot exploit multi-processors architectures efficiently.

4.2. Data flow model

A graph of Jack clients typically contains *sequenced* and *parallel* sub-parts (Fig 4). When parallel sub-graph exist, clients can be executed on different processors at the same time. A data-flow model can be used to describe this kind of system: a node in a data-flow graph becomes *runnable* when all inputs are available. The client ordering step that was used in the mono-processor model is not necessary anymore. Each client uses an *activation counter* to count the number of input clients which it depends of. The state of client connections is updated each time a connection between ports is done or removed.

Activation will be transfered from client to client during each server cycle as they are executed: a suspended client will be resumed, executes itself, propagates activation to the output clients, go back to sleep, until all clients

have been activated. [2]

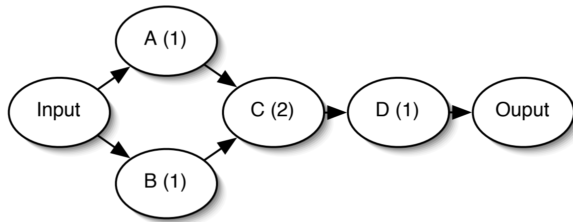

Figure 4. *Client graph: each client has an activation counter containing the number of input clients which it depends of. Client A and B can be executed in parallel: the first one that finish decrements the C activation from 2 to 1, the second one decrements the C activation from 1 to 0 and finally resumes C.*

4.3. Implementation

For easier evolution of the Jack server, the system has been simplified to better isolate platform specific sub-parts. Some data structure have been re-designed, for example the global port connection state is now located in shared memory.

Starting from this new version, the data-flow model has been implemented. Feedback loops between clients are detected and forbidden when connections are done. Each client uses an inter-process *suspend/resume* primitive associated with an *activation counter*. An implementation could be described with the following pseudo code. A server execution cycle consists of:

1. read audio input buffers
2. write ouput audio buffers computed the previous cycle
3. for each client in client list, reset the activation counter to its initial value
4. activate all client that depends of the input driver client or that do not have input dependencies
5. suspend until the next cycle

Activation of a client consists of:

1. atomically decrement its activation counter
2. resume the client if its activation counter equals zero

After being resumed by the system, execution of a client consists of:

1. call the client process function
2. propagate activation to ouput clients
3. suspend until the next cycle

[2]The data-flow model will still work on mono-processor machines and will correctly guaranty a minimum global number of context switches like the "pre-sorting step" model.

On each platform, an efficient synchronization primitive must be found to implement the suspend/resume operation.

- MacOSX low-level communications between user space and kernel are based on Mach ports and messages. More complex synchronization primitives are then built in the system on top of these low-level primitives. The Mach semaphores can be used to implement thread synchronization. On MacOSX 10.3 version, additional code has to be developed to have inter-process semaphores.
- Linux kernel 2.6 features the Fast User space mutEx (futex), a new facility that allows two process to synchronize (including blocking and waking) with either no or very little interaction with the kernel. Although we have not yet measured the performance of futexes in Jack it seems likely that they are better suited to the task of coordinating multiple processes than the FIFO's that the Linux implementation currently uses.

4.4. Performances

The multi-processor version has been implemented on MacOSX. Preliminary benchmarks have been done on a mono and dual 1.8 Ghz G5 machine. Five *jack-metro* clients generating a simple bip are running.

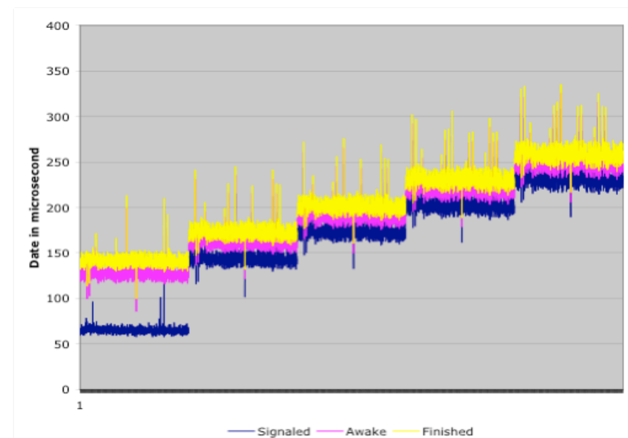

Figure 5. *For a server cycle: signal (black), awake (grey) and finish (white) date measured during 5 sec for each 5 client on a mono 1.8 Ghz G5. The behavior of each client is then represented in a "slice" in the graph and all slices have been concatenated on the X axis. Clients are connected in sequence thus computations are inevitably serialized. End date is about 250 microsecond on average.*

For a server cycle, the *signal date* (when the client resume semaphore is activated), the *awake date* (when the client actually wakes up) and the *finish date* (when the

client ends its processing and go back to suspended state) relative to the server cycle start date *before reading and writing audio buffers* have been measured. The first slice in the graph also reflects the server behavior: the duration to read and write the audio buffers can be seen as the *signal* date curve offset on the Y-coordinate. After having signaled the first client, the server returns to the CoreAudio HAL (Hardware Abstract Layer), that mix the output buffers in the kernel driver (offset between the first client *signal* date and its *awake* date (Fig 5)). Then the first client will be resumed.

With all clients running at the same time, the measure is done during 5 seconds. The behavior of each client is then represented as a 5 seconds "slice" in the graph and all slices have been concatenated on the X axis, thus allowing to have a global view of the system.

Two benchmarks have been done. In the first one, clients are connected in sequence (client 1 is connected to client 2, client 2 to client 3 and so on), thus computations are inevitably serialized. One can clearly see that the *signal* date of client 2 happens after the *finished* date of client 1 and the same behavior happens for other clients. Measures have been done on the mono (Fig 5) and dual machine (Fig 6).

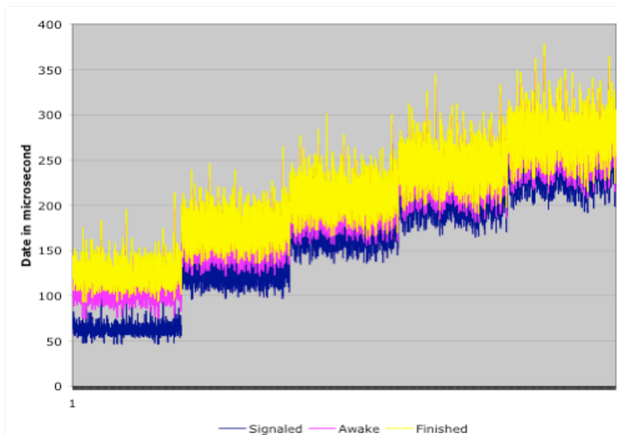

Figure 6. *For a server cycle: signal (black), awake (grey) and finish (white) date measured during 5 sec for each 5 client on a dual 1.8 Ghz G5. Since client are connected in sequence, computations are also serialized, but client 1 can start a little earlier on the second processor. End date is about 250 microsecond on average.*

In the second benchmark, clients all only connected to the input driver, thus they can possibly be executed in parallel. The input driver client signal all clients at (almost) the same date [3]. Measures have been done on the mono (Fig 7) and dual (Fig 8) machine. When parallel clients are executed on the dual machine, one see clearly that computations are done at the same time on the 2 processors and the end date is thus lowered.

[3]Signaling a semaphore has a cost that appears as the "slope" of the signal curve.

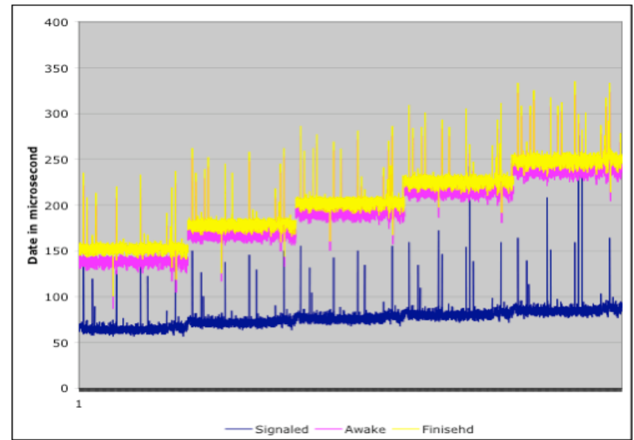

Figure 7. *For a server cycle: signal (black), awake (grey) and finish (white) date measured during 5 sec for each 5 client on a mono 1.8 Ghz G5. Although the graph can potentially be parallelized, computations are still serialized. End date is about 250 microsecond on average.*

Other benchmarks with different parallel/sequence graph to check their correct activation behavior and comparaison with the same graphs runned on the mono-processor machine have been done. A worst case additional latency of 150 to 200 microseconds added to the average finished date of the last client has been measured.

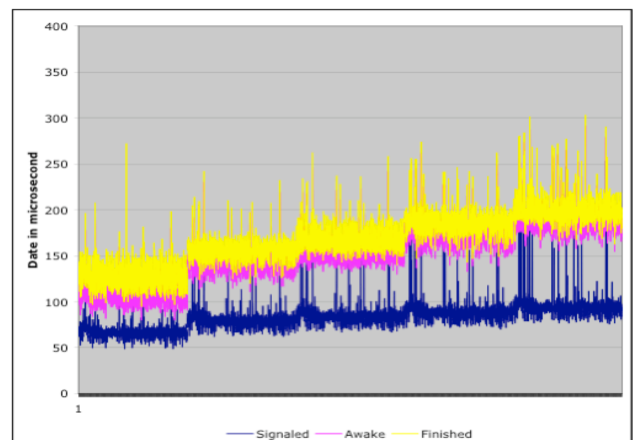

Figure 8. *For a server cycle: signal (black), awake (grey) and finish (white) date measured during 5 sec for each 5 client on a dual 1.8 Ghz G5. Computations are done in parallel. End date is about 200 microsecond on average.*

More complex setup using CoreAudio "jackified" applications (using the JAR driver) have been tested to check the real effect of parallelization with more demanding applications.

4.5. Optimizations and evolutions

Further optimizations will be tested and possibly implemented.

4.5.1. Suspend/resume semaphore sharing

Each client is currently associated with a semaphore. The client real-time audio thread is resumed when its semaphore is signaled by the previous client in the graph. When clients have the same input dependencies, they could share the same semaphore. The *shared* semaphore could be signaled only once and resume a group of waiting real-time threads. This optimization can be detected and done during the connection/disconnection step.

4.5.2. Port sharing

When an input port of client B (port 3) is connected to several output ports of client A (port 1 and 2), mixing is done lazily only when client B asks for its input port 3 (Fig 9). When several input ports have the same dependencies, the mixed buffer they will use could be computed only once and then shared (here port 3 and port 4 both have port 1 and 2 as inputs). A *shared* port can be used to represent this special case. Mixing will be done once and all possible users of the shared port will see the same mixed result. Detecting that ports can be shared can be done during the connection/disconnection step.

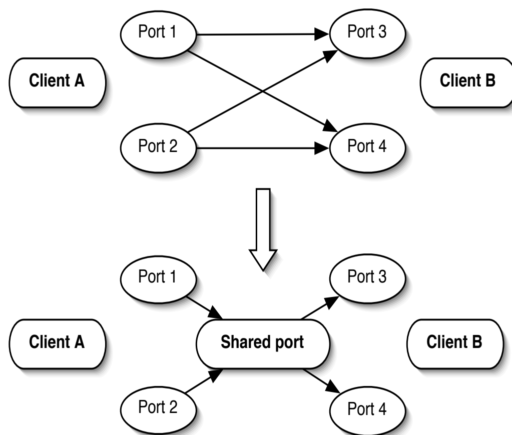

Figure 9. *Port sharing: port 3 and 4 have the same inputs (port 1 and 2). A shared port can be created to allow mixing to be done only once.*

5. CONCLUSION

The Jack system is a fundamental part of the Linux audio world, where most of music-oriented audio applications are now Jack compatible. On MacOSX, it has extended the CoreAudio architecture by adding low-latency inter-application audio routing capabilities. The multiprocessor version is a first step towards a completely distributed version, that will take advantage of multi-processor on a machine and could run on multiple machines in the future.

6. REFERENCES

[1] ALSA, *Advanced Linux Sound Architecture,* http://www.alsa-project.org, Nov. 2002.

[2] BENCINA ROSS, BURK PHIL, *PortAudio - an Open-Source Cross Platform Audio API,* Proceedings of the International Computer Music Conference ICMA, 2001

[3] ORLAREY YANN, LEQUAY HERVE, *MidiShare: a Real Time multi-tasks software module for Midi applications* Proceedings of the International Computer Music Conference ICMA, 1989 p 234-237

[4] VEHMANEN KAI, WINGO ANDY AND DAVIS PAUL, *Jack Design Documentation* http://jackit.sourceforge.net/docs/design/

[5] WESTERFELD STEFAN, *The aRts Handbook* http://www.arts-project.org/doc/handbook/