jackdmp: Jack server for multi-processor machines

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

Abstract

jackdmp is a C++ version of the Jack low-latency audio server for multi-processor machines. It is a new implementation of the jack server core features that aims in removing some limitations of the current design. The activation system has been changed for a data flow model and lock-free programming techniques for graph access have been used to have a more dynamic and robust system. We present the new design and the implementation for MacOSX.

Keywords

real-time, data-flow model, audio server, lock-free

1 Introduction

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 (Vehmanen, Wingo and Davis 2003). The current code base written in C, developed over several years, is available for GNU/Linux and MacOSX systems. An additional integration with the MacOSX CoreAudio architecture has been realized (Letz, Fober and Orlarey 2004).

The system is now 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 in a transparent manner. [1]

The new design and implementation aims in removing some limitations of the current version, by isolating the "heart" of the system and simplifying the implementation:

- the sequential activation model has been changed to a new graph activation scheme based on a data-flow model, that will naturally take profit of multi-processor machines

- a more robust architecture based on *lock-free* programming techniques has been developed to allow the server to keep working (not interrupting the audio stream) when the client graph changes or in case of client execution failure, especially interesting in live situations.

- various simplifications have been done in the internal design.

The section 2 explains the requirements, section 3 describes the new design, section 4 describes the implementation, and finally section 5 describes the performances.

2 Multi-processing

Taking profit of multi-processor architectures usually requires applications to be adapted. A natural way is to develop multi-threaded code, and in audio applications a usual separation consists in executing audio DSP code in a real-time thread and normal code (GUI for instance) in one or several standard threads. The scheduler then activates all runnable threads in parallel on available processors.

In a Jack server like system, there is a natural source of parallelism when Jack clients depend of the same input and can be executed on different processor at the same time. The main requirement is then to have an activation model that allows the scheduler to correctly activate parallel runnable clients. Going from a sequential activation model to a completely distributed one also raise synchronization issues that can be solved using *lock-free* programming techniques.

[1]All CoreAudio applications can take profit of Jack features without any modification

3 New design

3.1 Graph execution

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 topological sorting step is done each time the graph state changes, for example when connections are done or removed or when a new client opens or closes. This order is used by the server to activate clients in sequence.

Forcing a complete serialization of client activation is not always necessary: for example clients A and B (Fig 1) 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-processor architectures efficiently.

3.2 Data flow model

Data flow diagrams (DFD) are an abstract general representation of how data flows around a system. In particular they describe systems where the ordering of operations is governed by *data dependencies* and by the fact that only the availability of the needed data determines the execution of one of the process.

A graph of Jack clients typically contains *sequencial* and *parallel* sub-parts (Fig 1). 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 done 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 on. 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]

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

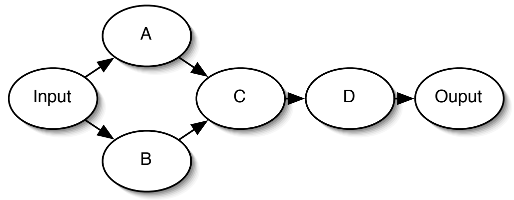

Figure 1: *Client graph: Client A and B could be executed at the same time, C must wait for A and B end, D must wait for C end.*

3.2.1 Graph loops

The Jack connection model allows loops to be established. Special *feedback connections* are used to close a loop, and introduce a one buffer latency. We currently follow Simon Jenkins [3] proposition where the feedback connection is introduced at the place where the loop is established. This scheme is simple but has the drawback of having the activation order become sensitive to the connection history. More complex strategies that avoid this problem will possibly be tested in the future.

3.3 Lock-free programming

In classic lock-based programming, access to shared data needs to be serialized using mutual exclusion. Update operations must appear as *atomic*. The standard way is to use a mutex that is locked when a thread starts an update operation and unlocked when the operation is finished. Other threads wanting to access the same data check the mutex and possibly suspend their execution until the mutex becomes unlocked. Lock based programming is sensitive to priority inversion problems or deadlocks. Lock-free programming on the contrary allows to build data structures that are safe for concurrent use without needing to manage locks or block threads (Fober, Letz, and Orlarey 2002).

Locks are used at several places in the current Jack server implementation. For example, the client graph needs to be locked each time a server update operation access it. When the real-time audio thread runs, it also needs to access the client graph. If the graph is already locked and to avoid waiting an arbitrary long time, the Real-Time (RT) thread generates an empty buffer for the given audio cycle, causing an annoying interruption in the audio stream.

A lock-free implementation aims at remov-

[3]Discussed on the jack-dev mailing list

ing all locks (and particularly the graph lock) and allowing all graph state changes (add/remove client, add/remove ports, connection/disconnection...) to be done *without interrupting the audio stream.* [4] As described in the implementation section, this new constraint requires also some changes in the client side threading model.

3.3.1 Lock-free graph state change

All update operations from clients are serialized through the server, thus only one thread updates the graph state. RT threads from the server and clients have to see the *same coherent* state during a given audio cycle. Non RT threads from clients may also access the graph state at any time. The idea is to use two states: one *current* state and one *next* state to be updated. A state change consists in *atomically* switching from the current state to the next state. This is done by the RT audio server thread at the beginning of a cycle, and other clients RT threads will use the same state during the entire cycle. All state management operations are implemented using the CAS [5] operation and are described with more details in the implementation section.

3.4 A "robust" server

Having a robust system is especially important in live situations where one can accept a temporary graph execution fault, which is usually better that having the system totally failing with a completely silent buffer and an audio stream interruption for example. In the current sequential version, the server waits for the client graph execution end before in can produce the output audio buffers. Thus a client that does not run during one cycle will cause the complete failure of the system.

In a multi-processor context, it is interesting to have a more *distributed* system, where a part of the graph may still run on one processor even if another part is blocked on the other one.

3.4.1 Engine cycle

The engine cycle has been redesigned. The server no longer waits for the client execution end. It uses the buffers computed at the previous cycle. The server cycle is fast and take almost constant time since it is totally decoupled from the clients execution. This allows the system to keep running even if a part of the graph can not be executed during the cycle for whatever reason (too slow client, crash of a client...).

The server is more robust: the resulting output buffer may be incomplete, if one or several clients have not produced their contribution, but the output audio stream will still be produced. The server can detect abnormal situations by checking if all clients have been executed during the previous cycle and possibly notify the faulty clients with an *XRun* event.

3.4.2 Latency

Since the server uses the output buffers produced during the previous cycle, this new model adds a *one buffer more latency in the system.*[6] But according to the needs, it will be possible to choose between the current model where the server is synchronized on the client graph execution end and the new more robust distributed model with higher latency.

4 Implementation

The new implementation concentrates on the core part of the system. Some part of the API like the *Transport* system are not implemented yet.

4.1 Data structure

Accessing data in shared memory using pointers on the server and client side is usually complex: pointers have to be described as offset related to a base address local to each process. Linked lists for example are more complex to manage and usually need locked access method in multithread cases. We choose to simplify data structures to use fixed size preallocated arrays that will be easier to manipulate in a lock free manner.

4.2 Shared Memory

Shared memory segments are allocated on the server side. A reference (index) on the shared segment must be transfered on the client side. Shared memory management is done using two classes:

- On the server side, the **JackShmMem** class overloads **new** and **delete** operators. Objects of sub-classes of JackShmMem will

[4] Some operations like buffer size change will still interrupt the audio stream.

[5] CAS is the basic operation used in lock-free programming: it compares the content of a memory address with an expected value and if success, replaces the content with a new value.

[6] At least on OSX where the driver internal behaviour concerning input and output latencies values cannot be precisely controlled

be automatically allocated in shared memory. The **GetShmIndex** method retrieves the corresponding index to be transfered and used on the client side.

- Shared memory objects are accessed using a standard pointer on the server side. On the client side, the **JackShmPtr** template class allows to manipulate objects allocated in shared memory in a transparent manner: initialized with the index obtained from the server side, a JackShmPtr pointer can be used to access data and methods [7] of the corresponding server shared memory object.

Shared memory segments allocated on the server will be transfered from server to client when a new client is registered in the server, using the corresponding shared memory indexes.

4.3 Graph state

Connection state was previously described as a list of connected ports for a given port. This list was duplicated both on the server and client side thus complicating connection/disconnection steps. Connections are now managed in shared memory in fixed size arrays.

The **JackConnectionManager** class maintains the state of connections. Connections are represented as an array of port indexes for a given port. Changes in the connection state will be reflected the next audio cycle.

The **JackGraphManager** is the global graph management object. It contains a connection manager and an array of preallocated ports.

4.4 Port description

Ports are a description of data type to be exchanged between Jack clients, with an associated buffer used to transfer data. For audio input ports, this buffer is typically used to mix buffers from all connected output ports. Audio buffers were previously managed in a independent shared memory segment.

For simplification purpose, each audio buffer is now associated with a port. Having all buffers in shared memory will allow some optimizations: an input port used at several places with the same data dependencies could possibly be *computed once and shared.* Buffers are preallocated with the maximum possible size, there is no re-allocation operation needed anymore. Ports are implemented in the **JackPort** class.

[7] Only non virtual methods

4.5 Client activation

At each cycle, clients that only depend of the input driver and clients without inputs have to be activated first. To manage clients without inputs, an internal *freewheel* driver is used: when first activated, the client will be connected to it. At the beginning of the cyle, each client has its activation counter containing the number of input client it depends on. After being activated, the client decrements the activation counter of all its connected output. The *last* activated input client will resume the following client in the graph. (Fig 2)

Each client uses an inter-process *suspend/resume* primitive associated with an *activation counter*. An implementation could be described with the following pseudo code. Execution of a server cycle follows several steps:

- read audio input buffers
- write output audio buffers computed the previous cycle
- for each client in client list, reset the activation counter to its initial value
- activate all clients that depends on the input driver client or without input
- suspend until next cycle

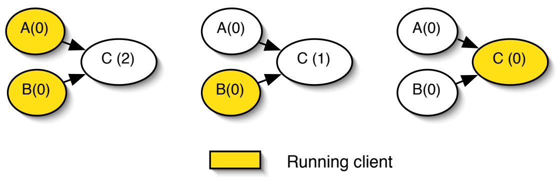

Figure 2: *Example of graph activation: C is activated by the last running of its A and B input.*

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

- call the client process callback
- propagate activation to output clients
- suspend until the next cycle

On each platform, an efficient synchronization primitive is needed to implement the suspend/resume operation. Mach semaphores are used on MacOSX. They are allocated and published by the server in a global namespace (using the *mach bootstrap service* mechanism). Running clients are notified when a new client is opened and access the corresponding semaphore.

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. 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.

4.6 Lock-free graph access

Lock-free graph access is done using the **JackAtomicState** template class. This class implement the two state pattern. Update methods use on the *next state* and read methods access the *current state*. The two states can be atomically exchanged using a CAS based implementation.

- code updating the next state is *protected* using the **WriteNextStateStart** and **WriteNextStateStop** methods. When executed between these two methods, it can freely update the next state and be sure that the RT reader thread can not switch to the next state.[8]
- the RT server thread switch to the new state using the **TrySwitchState** method that returns the current state if called concurrently with a update operation and switch to the next state otherwise.
- other RT threads read the current state, valid during the given audio cycle using the **ReadCurrentState** method.
- non RT threads read the current state using the **ReadCurrentState** method and have to check that the state was not changed during the read operation (using the **GetCurrentIndex** method):

```
void ClientNonRTCode(...)
{
    int cur_index,next_index;
    State* current_state;
    next_index = GetCurrentIndex();
    do {
       cur_index = next_index;
       current_state = ReadCurrentState();
       ...
       < copy current_state >
       ...
       next_index = GetCurrentIndex();
    } while (cur_index != next_index);
}
```

[8]The programming model is similar to a lock-based model where the update code would be written inside a *mutex-lock/mutex-unlock* pair.

4.7 Server client communications

A global *client registration* entry point is defined to allow client code to register a new client (a **JackServerChannel** object). A private communication channel is then allocated for each client for all *client requests*, and remains until the client quits. Possible crash of a client is detected and handled by the server when the private communication channel is abnormally closed. A notification channel is also allocated to allow the server to notify clients: graph reorder, xrun, port registration events...

Running clients can also detect when the server no more runs as soon as waiting on the input suspend/resume primitive fails. (Fig 3)

The current version uses socked based channels. On MacOSX, we use MIG (Mach Interface Generator), a very convenient way to define new Remote Procedure Calls (RPC) between the server and clients. [9]

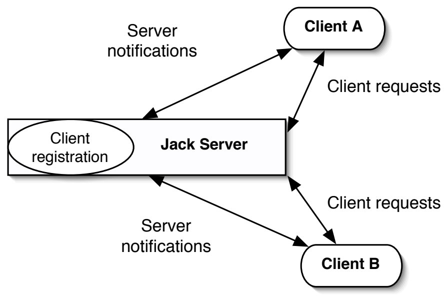

Figure 3: *The server defines a public "client registration" channel. Each client is linked with the server using two "request "and "notification" channels.*

4.8 Server

The Jack server contains the global client registration channel, the drivers, an engine, and a graph manager. It receives requests from the global channel, handle some of them (BufferSize change, Freewheel mode..) and redirect other ones on the engine.

4.8.1 Engine

The engine contains a **JackEngineControl**, a global shared server object also visible for clients. It does the following:

[9]Both synchronous and asynchronous function calls can be defined

- handles requests for new clients through the global client registration channel and allocates a server representation of new external clients
- handles request from running clients
- activates the graph when triggered by the driver and does various timing related operations (CPU load measurement, detection of late clients...)

4.8.2 Server clients

Server clients are either internal clients (a **JackInternalClient** object) when they run in the server process space[10] or external clients (a **JackExternalClient** object) as a server representation of an external client. External clients contain the local data (for example the notification channel, a **JackNotifyChannel** object) and a **JackClientControl** object to be used by the server and the client.

4.8.3 Library Client

On the client side, the current Jack version uses a one thread model: real-time code and notifications (graph reorder event, xrun event...) are treated in a unique thread. Indeed the server stops audio processing while notifications are handled on the client side. This has some advantages: a much simpler model for synchronization, but also some problematic consequences: since notifications are handled in a thread with real-time behaviour, a non real-time safe notification may disturb the whole machine.

Because the server audio thread is not interrupted anymore, most of server notifications will typically be delivered while the client audio thread is also running. A two threads model for client has to be used:

- a real-time thread dedicated to the audio process
- a standard thread for notifications

The client notification thread is started in **jack-client-new** call. Thus clients can already receive notifications when they are in the *opened* state. The client real-time thread is started in **jack-activate** call. A connection manager client for example does not need to be activated to be able to receive *graphreorder*, or *portregistration* like notifications (Fig 4).

[10]Drivers are a special subclass of internal clients

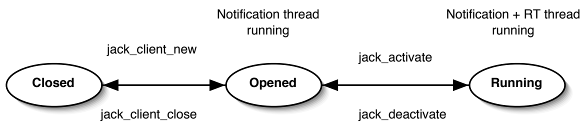

Figure 4: *Client life cycle*

This two threads model will possibly have some consequences for existing Jack applications: they may have to be adapted to allow a notification to be called while the audio thread is running.

The library client (a **JackLibClient** object) redirects the external Jack API to the Jack server. It contains a **JackClientChannel** object that implements both the request and notification channels, local client side resources as well as access to objects shared with the server like the graph manager or the server global state.

4.8.4 Drivers

Drivers are needed to activate the client graph. Graph state changes (new connections, port, client...) are done by the server RT thread. When several drivers need to be used, one of them is called the *master* and updates the graph. Other one are considered as *slaves.*

The **JackDriver** class implements common behaviour for drivers. Those that use a blocking audio interface (like the **JackALSADriver** driver) are subclasses of the **JackThreadedDriver** class. A special **JackFreewheelDriver** (subclass of JackThreadedDriver) is used to activate clients without inputs and to implement the freewheel mode (see 4.8.5). The **JackAudioDriver** class implements common code for audio drivers, like the management of audio ports. Callback based drivers (like the **JackCoreAudioDriver** driver, a subclass of JackAudioDriver) can directly trigger the Jack engine.

When the graph is synchronized to the audio card, the audio driver is the master and the freewheel driver is a slave.

4.8.5 Freewheel mode

In freewheel mode, Jack no longer waits for any external event to begin the start of the next process cycle thus allowing faster than real-time execution of Jack graph. Freewheel mode is implemented by switching from the audio and freewheel driver synchronization mode to the freewheel driver only:

- the global connection state is saved
- all audio driver ports are deconnected, thus there is no more dependancies with the audio driver
- the freewheel driver is synchronized with the end of graph execution: all clients are connected to the freewheel driver
- the freewheel driver becomes the master

Normal mode is restored with the connections state valid before freewheel mode was done. Thus one consider that no graph state change can be done during freewheel mode.

4.9 XRun detection

Two kind of XRun can be detected:

- XRun reported by the driver
- XRun detected by the server when a client has not be executed the previous cycle: this typically correspond to abnormal scheduler latencies

On MacOSX, the CoreAudio HAL system already contains a XRun detection mechanism: a *kAudioDeviceProcessorOverload* notification is triggered when the HAL detects an XRun. The notification will be redirected to all running clients. All clients that have not been executed the previous cycle will be notified individually.

5 Performances

The multi-processor version has been tested 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.

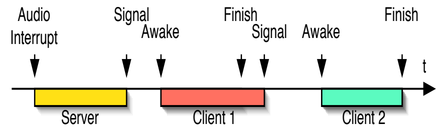

Figure 5: *Timing diagram for a two clients in sequence example*

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), which mix the output buffers in the kernel driver (offset between the first client *signal* date and its *awake* date (Fig 5)). The first client is then resumed.

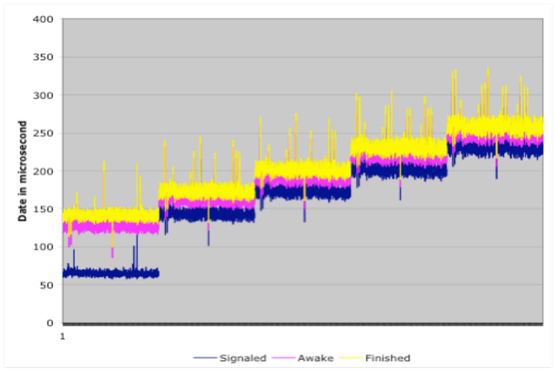

Figure 6: *Mono G5, clients connected in sequence. For a server cycle: signal (blue), awake (pink) and finish (yellow) date. End date is about 250 microsecond on average.*

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 6) and dual machine (Fig 7).

In the second benchmark, all clients are 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 [11]. Measures have been done on the mono (Fig 8) and dual (Fig 9) machine. When parallel clients are executed on the dual

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

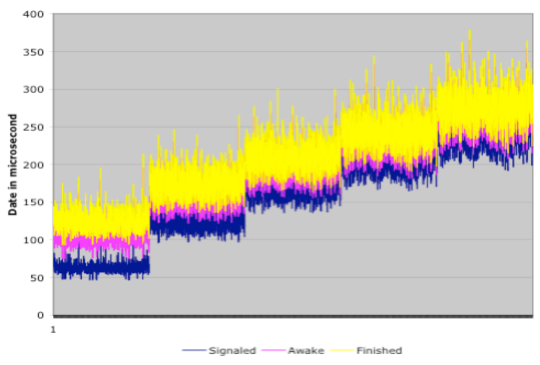

Figure 7: *Dual G5. Since clients are connected in sequence, computations are also serialized, but client 1 can start earlier on the second processor. End date is about 250 microsecond on average.*

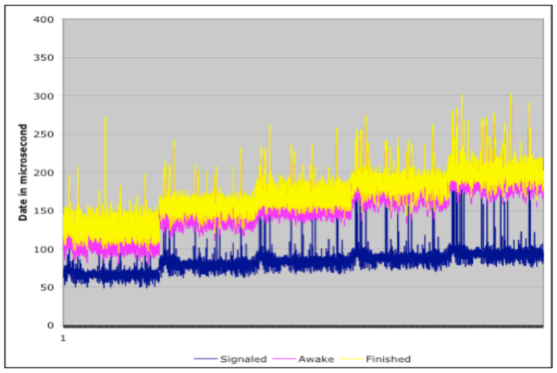

Figure 9: *Parallel clients on a dual G5. Client 1 can start earlier on the second processor before all clients have been signalled. Computations are done in parallel. End date is about 200 microsecond on average.*

machine, one see clearly that computations are done at the same time on the 2 processors and the end date is thus lowered.

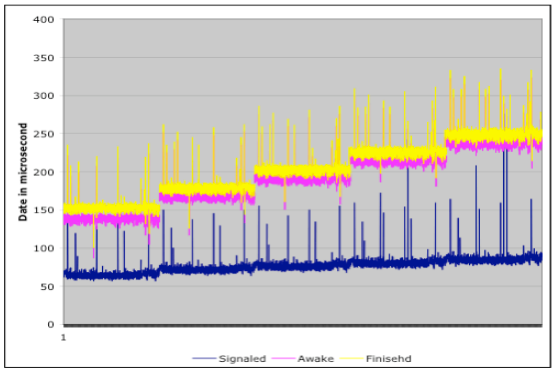

Figure 8: *Parallel clients on a mono G5. Although the graph can potentially be parallelized, computations are still serialized. End date is about 250 microsecond on average.*

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.

6 Conclusion

With the development of multi-processor machines, adapted architectures have to be developed. The Jack model is particularly suited to this requirement: 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.

By adopting a data flow model for client activation, it is possible to let the scheduler naturally distribute parallel Jack clients on available processors, and this model works for the benefit of all kind of client aggregation, like *internal clients in the Jack server*, or *multiple Jack clients in an external process.*

A Linux version has to be completed with an adapted primitive for inter process synchronization as well as socket based communication channels between the server and clients. The multi-processor 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.

References

D.Fober, S.Letz, Y.Orlarey "Lock-Free Techniques for Concurrent Access to Shared Objects", *Actes des Journes d'Informatique Musicale JIM2002, Marseille, pages 143–150*

S.Letz, D.Fober, Y.Orlarey, P.Davis "Jack Audio Server: MacOSX port and multi-processor version, *Proceedings of the first Sound and Music Computing conference - SMC'04", pages 177–183*

Vehmanen Kai, Wingo Andy and Davis Paul "Jack Design Documentation", *http://jackit.sourceforge.net/docs/design/*