jackdmp: Jack server for multi-processor machines

S.Letz, D.Fober, Y.Orlarey
Internal technical report 11-28-05
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 Linux and OSX.

1 Introduction

The new design and implementation aims in removing limitations of the current version. This has been done by isolating the "heart" of the system and simplifying the implementation:

- removing of the sequential client activation limitation using a new graph activation scheme based on a data-flow model, that will naturally take profit of multi-processor machines.
- more robust architecture based on *lock-free* programming techniques allowing 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 changes in the internal design to simplify portability.

2 Multi-processing

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.

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.

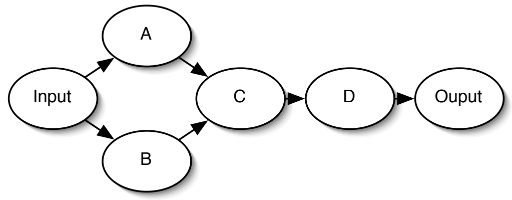

Figure 1: *Client graph: Client A and B could be executed at the same time, C must wait for A and B end, D must wait for C end.*

3.2 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. [1]

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 [2] 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 strategy that avoid this problem will possibly be tested in the future.

3.2.2 Complete graph

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, a 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 of. After being activated, a 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).

3.3 Lock-free programming

In classic lock-based programming, access to shared data needs to be serialized. 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 will check the mutex and possibly suspend their execution until the mutex becomes unlocked. Lock based programming is classically sensitive to prioriy inversion problems (when a high priority thread waits for a ressource owned by a lower priority thread) 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.

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

[2] Discussed on the jack-dev mailing list

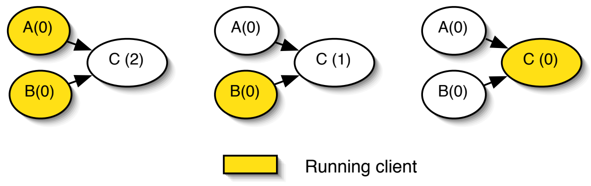

Figure 2: *Example of graph activation: C is activated by the last running of its A and B input.*

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 needs also to access the graph lock. If the graph is already locked and to avoid waiting an arbitrary long time, the RT thread will generate an empty buffer for the given audio cycle, causing an anoying interruption in the audio stream.

A lock-free implementation aims in removing 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.* [3] As we will see 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 will update the graph state. RT threads from the server and clients have to see a *coherent* state during a given audio cycle. Non RT threads from clients may also access the graph state at

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

any moment. 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 server thread at the beginning on a cycle. Clients RT threads will use the same state during the entire cycle. All state management operations are implemented thanks to the CAS operation and are described more deeply 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 interrupting the audio stream.

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 can be started in two different modes:

- *synchronous* mode: as in the previous version, the server waits for the client graph execution end before it can produce the output audio buffers. Thus a client that does not run during one cycle will cause the complete failure of the system.
- *asynchronous* mode: the server no longer waits for the graph 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 client 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.

Each client uses an inter-process *suspend/resume* primitive associated with an *activation counter.* For *synchronous* execution mode, the server cycle consists of:

- read audio input buffers
- for each client in client list, reset the activation counter to its initial value
- activate all clients connected to the audio driver and freewheel driver
- write output audio buffers
- suspend until the next cycle

For *asynchronous* execution mode, the server cycle consists of:

- 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 connected to the audio driver and freewheel driver
- suspend until the next cycle

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

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

3.4.2 Latency

This asynchronous model for engine cycle activation adds a *one buffer more latency in the system* [4]. But according to the needs, it is 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

Using pointers in shared memory 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 multi-thread 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]The additional latency can probably be lowered by adjusting the output advance used at the driver level

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 be automatically allocated in shared memory. The **GetShmIndex** method retrieves the corresponding index to be transfered and used on the client 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 [5] of the corresponding shared memory object. Shared memory objects will be accessed using a standard pointer on the server side and an JackShmMem pointer on the client side.

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, using the **JackAtomicState**.

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.

4.5 Client activation

On each platform, an efficient synchronization primitive must be found to implement the suspend/resume operation. On Linux, Fifo and POSIX named semaphores have been tested and POSIX named semaphores are the fastest. On OSX, Fifo, POSIX named semaphores and Mach semaphores have been tested and Mach semaphores are the fastest. They are allocated and published by the server in a global namespace (using the mach bootstrap service mechanism).

Synchronization primitives are described in the **JackSynchro** class. Specialized versions are implemented in **JackFifo**, **JackPosixSemaphore** and **JackMachSemaphore** classes.

When a new client appears in the server, running clients are notified and will access the corresponding synchronization primitive.

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).[6] A typical update method would be written the following way:

[5]Only non virtual methods

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

```
void ServerUpdate(...)
{
    State* next_state;
    next_state = WriteNextStateStart();
    ...
    < update next_state >
    ...
    WriteNextStateStop();
}
```

- the RT server thread switch to the new state using the **SwitchState** method:

```
void ServerRTCode(...)
{
    State* current_state;
    current_state = TrySwitchState();
    ...
    < use current_state >
    ...
}
```

- other RT threads read the current state, valid during the entire audio cycle using the **ReadCurrentState** method:

```
void ClientRTCode(...)
{
    State* current_state;
    current_state = ReadCurrentState();
    ...
    < use current_state >
    ...
}
```

- non RT threads read the current state using the **ReadCurrentState** method and can 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;
    do {
       cur_index = GetCurrentIndex();
       current_state = ReadCurrentState();
       ...
       < copy current_state >
       ...
       next_index = GetCurrentIndex();
     } while (cur_index != next_index);
}
```

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 notify client. Notifications are separated in two sets:

- *synchronous notifications* where the server waits for the client answer: they are used for add/remove client, buffersize change and freewheel events.
- *asynchronous notifications* where the server waits for the client answer: they are used for other kind of notifications like graph reorder, port registration and xrun events.

Running clients can also detect that the server no more runs when they input suspend primitive fails. (Fig 3) Communication channels are defined in a set of abstract classes: **JackClientChannelInterface**, **JackNotifyChannelInterface**, **JackServerChannelInterface** and **JackServerNotifyChannelInterface** which are implemented on each platform. On Linux, communication channels are based on socket. On MacOSX, we use MIG (Mach Interface Generator), a very convenient way to define new Remote procedure Calls (RPC) between the server and clients. [7]

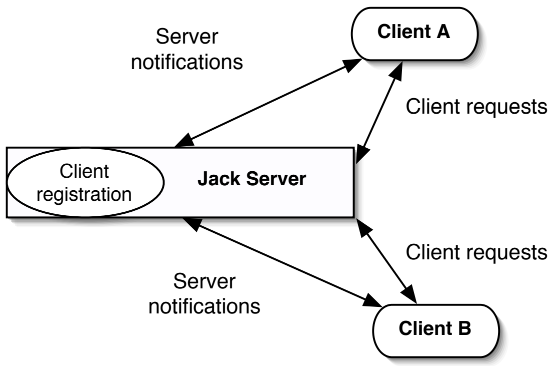

Figure 3: *The server defines a public "client registration" channel. Each client is linked with the server through two "request "and "notification" channels.*

4.8 Server

The Jack server contains the global client registration channel, the drivers, the engine, the en-

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

gine shared control (**JackEngineControl** object), the synchronization table, and a graph manager. It receives requests from the global channel, handle some of them (BufferSize change, Freewheel mode..) and other one are managed by the engine.

4.8.1 Engine

The engine contains the list of running client and timing related components. It does the following:

- 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
- activate 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 runs in the server process space[8] 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 notification (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 issues, 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 entire machine.

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

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

[8]Drivers are a special type of internal clients

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).

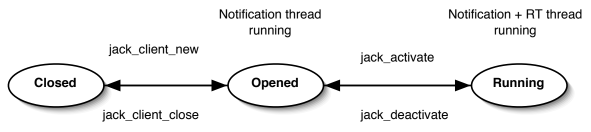

Figure 4: *Client life cycle*

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 object shared with the server like the graph manager or the server global state. **JackInternalClient** objects and **JackLibClient** objects share part of their implementation in a common superclass **JackClient**.

4.8.4 Drivers

Drivers are needed to activate the client graph. When several drivers need to be used, one of them is called the "master" and will update 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.

An experimental **JackLoopbackDriver** has been implemented. It allows to manually pipeline two applications connected in sequence by adding an additional one buffer latency in the connection, thus allowing parallel activation in a multi-processor context.

When the graph is synchronized to the audio card, the audio driver will be the "master" and the freewheel driver will be 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 will be 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

In synchronous mode, abnormal situations (too long cycle) will usually be detected by the driver. In asynchronous mode, the server can detect abnormal situations by checking if all clients have been executed during the previous cycle and notify the faulty clients with an *XRun* event.

On Linux, XRun detected by the ALSA driver are reported. [9] 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.

4.10 API semantic changes

Due to changes in the internal design, some functions in the external API have slightly changed semantic:

- with the 2 threads model on the client side, notification callbacks are now concurrent to the audio process. Applications may need to be adapted.
- **jack-port-connected** now works for all ports, not only the ones registered by the client.
- **jack-connect/jack-disconnect** are now asynchronous: the effective change will be seen at the next server cycle.

[9]Xrun reported at the engine level are currently not handled

5 Conclusion

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

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

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

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/