JACK AUDIO SERVER FOR MULTI-PROCESSOR MACHINES

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

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 a new C++ version for multi-processor machines that aims at 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.

1. INTRODUCTION

Jack is a low-latency audio server that can connect a number of different applications to an audio device, as well as allowing them to share audio between themselves. The current code base, written in C, is available for GNU/Linux and MacOSX systems[3].

The system is now a fundamental part of the Linux audio world, where most of music-oriented audio applications are Jack compatible. On MacOSX, it has extended the CoreAudio architecture by adding low-latency inter-application audio routing capabilities in a transparent manner[2].

2. ARCHITECTURE

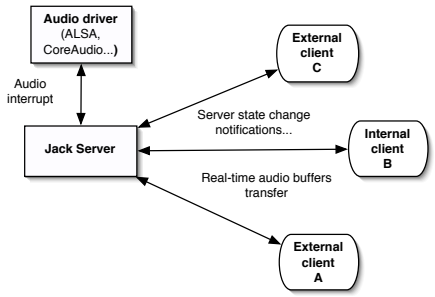

Figure 1. *Archictecture of Jack server/client system*

2.1. Server

Jack is based on a server/client model (Fig 1). The Jack server 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 of Jack clients, and each one has its *process* callback 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 must 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.

2.2. Driver

The whole graph is executed synchronously by a driver which interacts with the hardware, 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 represents a 3 ms duration).

2.3. 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 is awaken by the engine when it is time to operate on its data. 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, that is deterministic and that does 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 high kernel scheduling latencies or client graph process overloading is notified to the clients as *xruns*, and too slow clients during several consecutive audio cycles are usually removed from the graph.

3. GRAPH EXECUTION

3.1. Sequential 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 topological sorting step is done each time the graph state changes, for example when connections are done or removed. 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 2) 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. Multi-processor version

Multi-processor machines become more and more common (for example all high end Apple systems are dual processor machines and bi-Xeon hyperthreaded machines offer four CPU).

Taking profit of multi-processor architectures usually requires applications to be adapted. 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.

3.3. 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 2). 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.

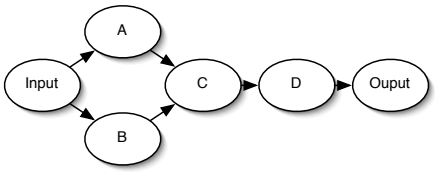

Figure 2. *Client graph: Client A and B could be executed at the same time, C must wait for A and B end, D must wait for C end.*

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]

3.4. Client activation

At each cycle, clients that only depend of the input driver and clients without inputs have to be activated first. At the beginning of the cycle, each client has its activation counter reseted to the number of input client it depends on. After being activated, a client decrements the activation counter of all its connected output. The *last* activated input client resumes the following clients in the graph (Fig 3). 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 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 that depends on the input
  driver client or without input
- suspend until the next cycle
```

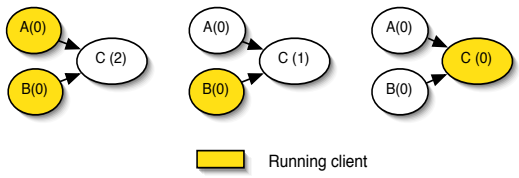

Figure 3. *Example of graph activation: C is activated by the last running of its A and B input.*

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

[1] 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.

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

On each platform, an efficient synchronization primitive must be found to implement the suspend/resume operation. Mach semaphores are used on MacOSX and Linux kernel 2.6 features the Fast User space mutEx (futex).

3.5. Lock-free implementation

In classic lock-based programming, access to shared data needs to be serialized using mutual exclusion. Update operations must appear as *atomic*. 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[1].

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 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 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*. [2]

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 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 audio server thread at the beginning of a cycle, and other clients RT threads will use the same state during the whole cycle. All state management operations are implemented using the CAS [3] operation.

Lock-free state update operations are implemented using a set of primitives operations:

- 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. [4]
- 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 whole 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);
}
```

4. PERFORMANCES

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

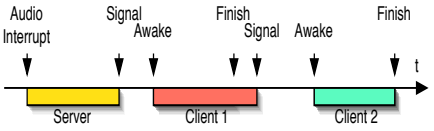

Figure 4. *Timing diagram for an example with two clients in sequence*

For a server cycle, the *signal date [blue]* (when the client resume semaphore is activated), the *awake date [pink]* (when the client actually wakes up) and the *finish date [yellow]* (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 audio driver, that internally mix the output buffers in the kernel (offset between the first client *signal* date and its *awake* date (Fig 5)). Then the first client will be resumed.

The measure is done during 5 seconds with all clients running. 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.

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

[3] 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.

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

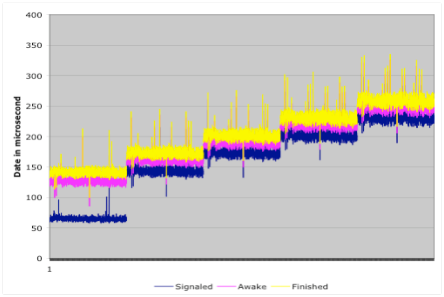

Figure 5. *Mono G5, clients connected in sequence. For a server cycle: signal (blue), awake (pink) and finish (yellow) date. End date is about 250 microsecond on average.*

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).

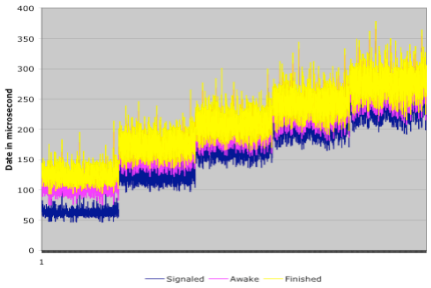

Figure 6. *Dual G5. Since clients are connected in sequence, computations are also serialized, but client 1 can start earlier on the second processor. End date is about 250 microsecond on average.*

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 [5]. 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.

Other benchmarks with different parallel/sequence graph 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.

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

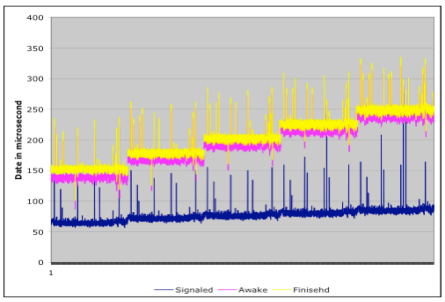

Figure 7. *Parallel clients on a mono G5. Although the graph can potentially be parallelized, computations are still serialized. End date is about 250 microsecond on average.*

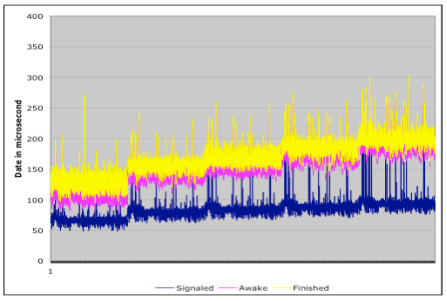

Figure 8. *Parallel clients on a dual G5. Client 1 can start earlier on the second processor before all clients have been signalled. Computations are done in parallel. End date is about 200 microsecond on average.*

5. CONCLUSION

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.

6. REFERENCES

[1] 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*

[2] 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*

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