jackdmp: Jack server for multi-processor machines

Stéphane Letz, Yann Orlarey, Dominique Fober
Grame, centre de création musicale
Lyon, France

Main objectives

- **To take advantage of multi-processor architectures: better use of available CPUs**

- **To have a more "robust" server:**

  - no more interruption of the audio stream

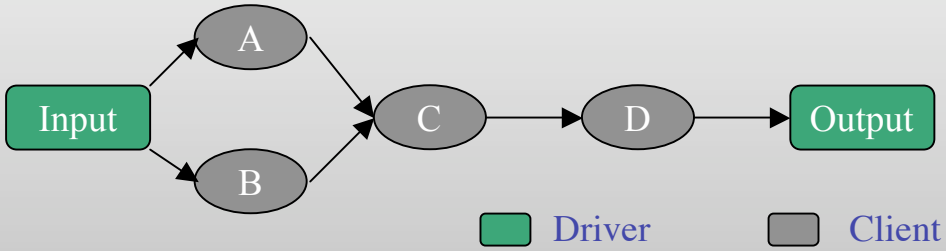
  - better client failure handling

→ **improved user experience: "glitch free" connections/disconnections…**

How ?

- **Using a “data-flow” model for client graph execution**
- **Using “lock-free” programming methods**
- **Redesigning of some internal parts: client threading model…**

Graph execution model

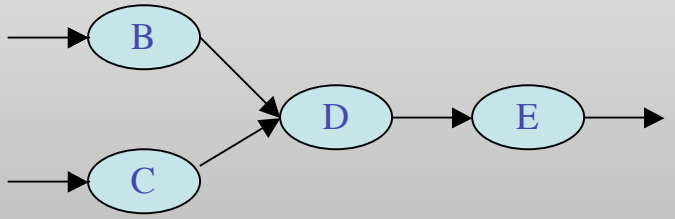


- The current version does a “topological sort” to find an activation order (A, B, C, D or B, A, C, D here)

- There is a natural source of parallelism when clients have the same input dependencies and can be executed concurrently

Data-flow model

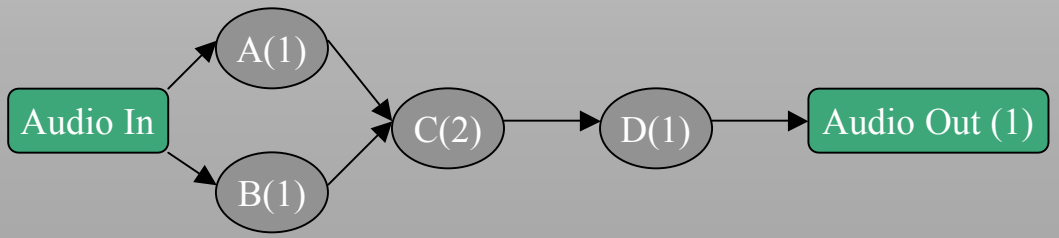
- Data-flow models are used to express parallelism
- Defined by “nodes” and “connections”
- Connections have properties



- The availability of the needed data determines the execution of the processes
- Execution can “data-driven” (in => out) or “demand-driven” (out => in)

“Semi” data-flow model

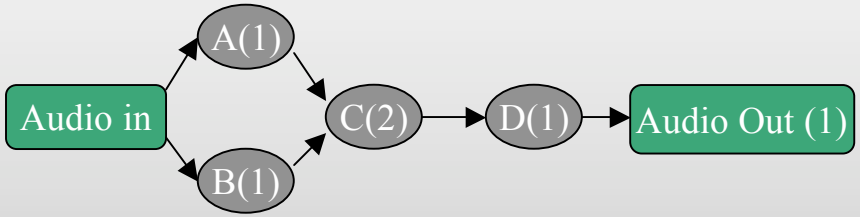
- **Currently tested: “semi” data-flow model where activation go in one direction only until *all* nodes have been executed**
- **Execution is synchronized by the audio cycle**
- **Activation counters are used to describe data dependencies**
- **A synchronization primitive is built using the activation counter and an “inter-process semaphore”**



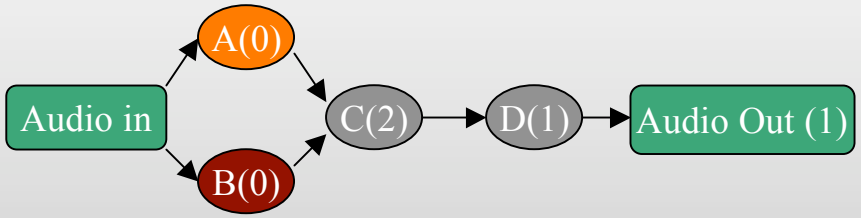
Graph execution

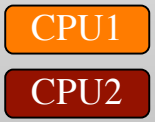
- Graph activation state is re-initialized at the beginning of each cycle
- The server initiates the graph execution by activating input drivers
- Activation is “propagated” by clients themselves until all clients have been executed

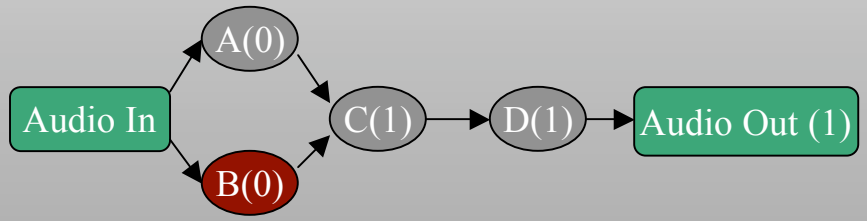
State 1



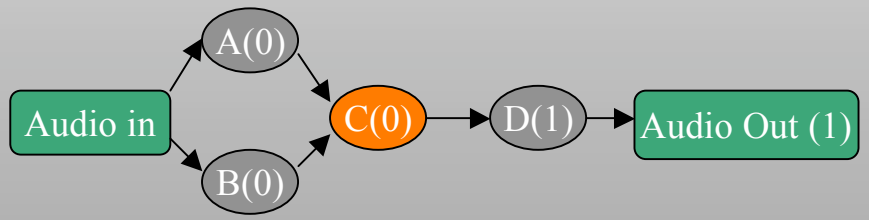
State 2







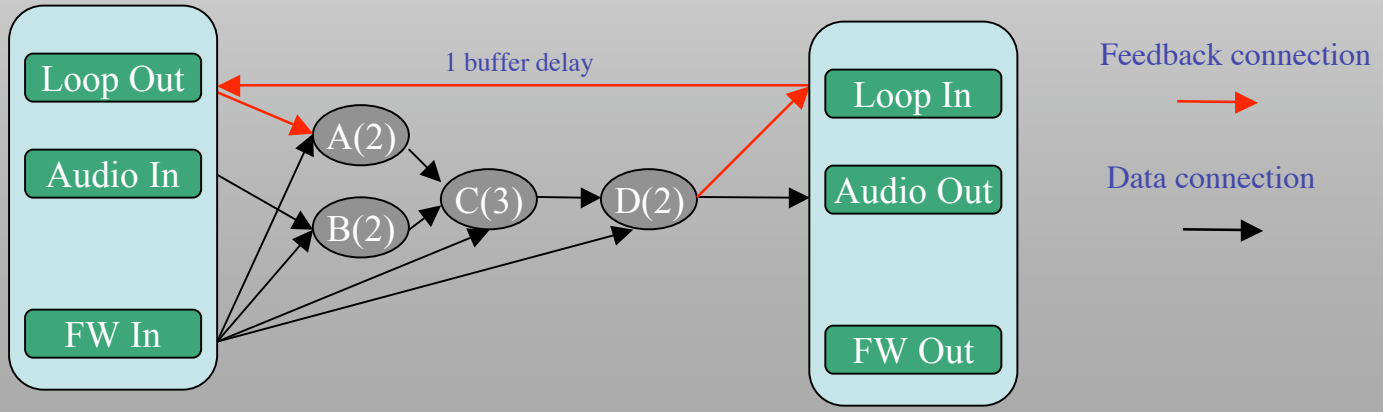
State 3



State 4

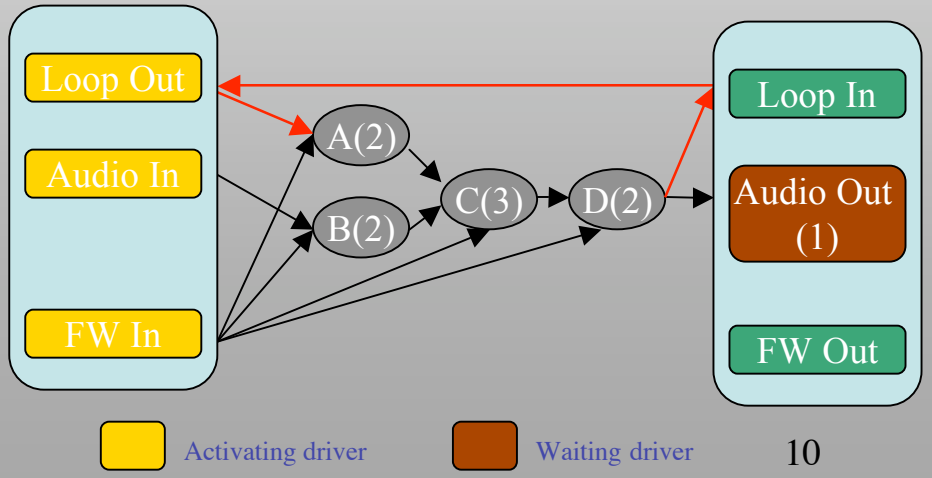
Complete graph

- **Some clients do not have audio inputs**
- **A “Freewheel” driver is connected to all clients**
- **Loops are detected and closed with a “Loop” driver**



Engine cycle: synchronous mode

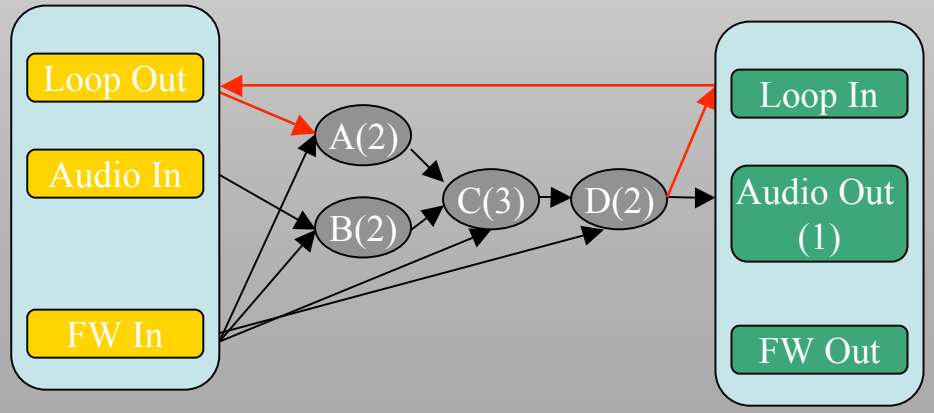
- **Read Input buffers**
- **Activate graph:** reset activation, timing…
- **Activate drivers: Audio,FW,Loop**
- **Wait for graph execution end**
- **Write output buffers**



Engine cycle: asynchronous mode

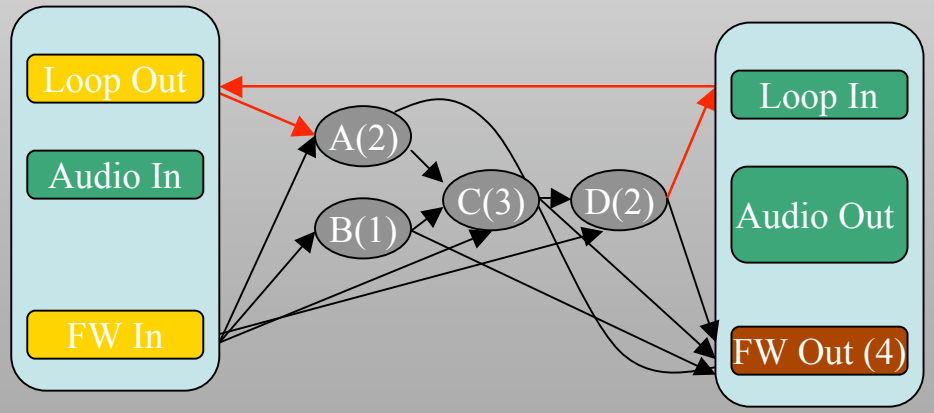
- **Read Input buffers**
- **Write output buffers from the previous cycle**
- **Activate graph:** reset activation, timing…
- **Activate drivers: Audio,FW,Loop**

one buffer more latency
one less context switch



Engine cycle : freewheel mode

- **Disconnect audio driver from the clients, connect to FW out**
- **The freewheel driver switches to a non-RT scheduling mode**
- **Activate graph at “full speed” in synchronous mode**



“Lock-based” graph state management

- The graph is “locked” whenever a read/write operation access it
- If the RT audio thread access the graph, it can not afford to wait for the lock
- A “null” cycle (silent buffer) is generated instead
- The reason for audio glitches when connecting/disconnecting

“Lock-free” graph state management

- **Avoid to lock the graph**
- **The audio stream is never stopped for “normal” operations**
- **Only interrupted during important changes (buffer size…) or failure cases**

What is “lock-free” programming?

- **Avoid mutual exclusion when several threads access a data structure**
- **Avoid deadlocks, priority inversion, convoying….**
- **“Lock-free” and “Wait-free” (stronger)**
- **Need processor specific instructions:**
  - **CompareAndSwap (CAS) : Intel**
  - **LoadReserve/StoreConditionnal: PPC**

Example

- **Implementing AtomicAdd using CAS:**

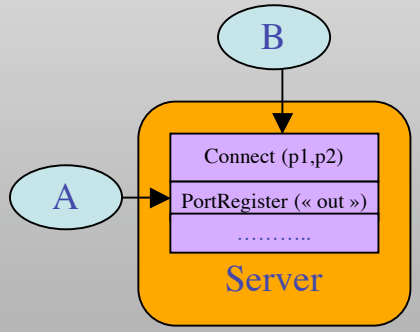
```
int AtomicAdd(int* value, int amount)
{
        int oldValue;
        int newValue;
        do {
                oldValue = * value;
                newValue = oldValue + amount;
        } while ( ! CAS(oldValue, newValue, value));
        return oldValue;
}
```

Lock-free graph state management (1)

- **Graph state (typically port connections) is shared between the server and clients**

- **Only one writer thread in the server: client access is serialized**

- **Multiple readers:**
  - **RT threads in server and clients**
  - **Non RT thread in clients**

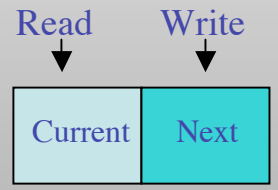


- **All RT readers must see the *same* (activation) state during a cycle**

Lock-free graph state management (2)

- **Using two separated graph states**

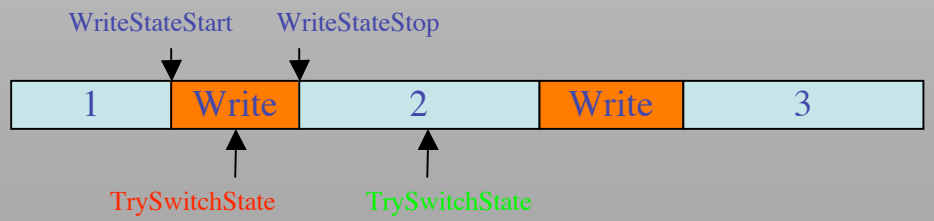
- **Switching from current to next state can be done:**
  - **when there is no more RT readers**
  - **if no write operation is currently done**



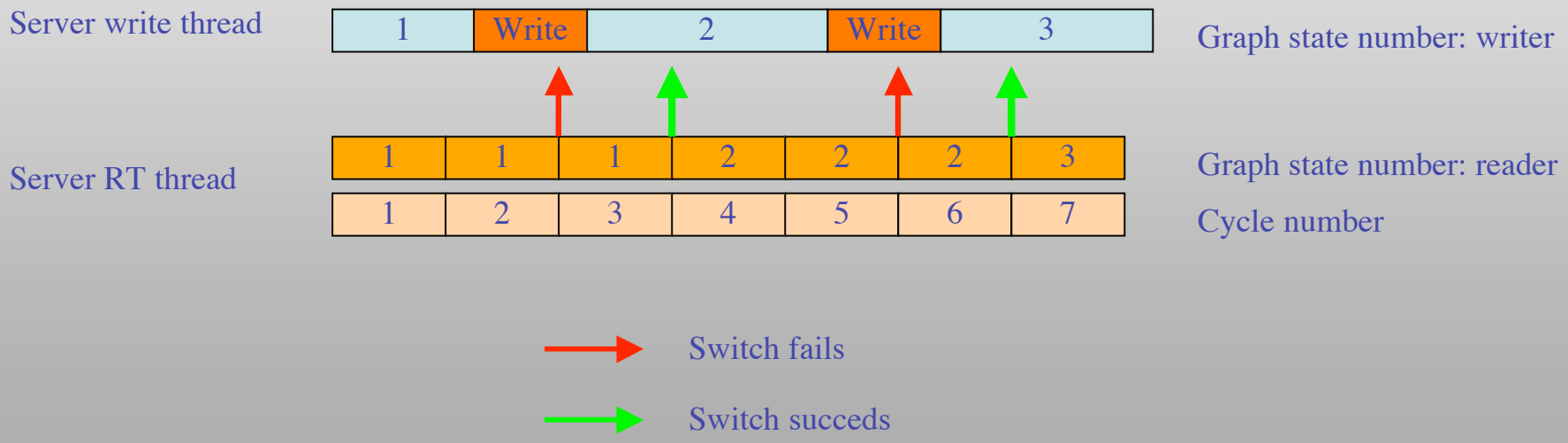
- **Switching states is done by the RT server thread at beginning of the cycle**

Lock-free graph state management (3)

- **Write operations are “protected” using WriteStateStart and WriteStateStop**
- **Switching is done using TrySwitchState**
  - **TrySwitchState returns the current state if called in the WriteStateStart/WriteStartStop window**
  - **Atomically switch from current to next state otherwise**
  - **Further write operations will copy the “new current state” and continue**
- **Other RT threads use ReadState to access the current state**



Lock-free graph state management (4)



Lock-free graph state management (5)

- **Programming model similar to the use of Lock/Unlock/Trylock primitives**
- **Non RT readers use ReadState in a “retryloop” to check state consistency**
- **Consequences:**
  - **write operations appear as “asynchronous” for clients**
  - **if needed, they have to be made synchronous by “waiting” for the effective graph state change (typically needed before notifying a “graph state change”)**

Client threading model (1)

- **Current situation:**
  - **a single thread is used for RT code and “notifications” (like graph order change…)**
  - **this thread is RT even when executing notifications…**
- **Since the server audio RT thread is never stopped anymore, notifications need to be executed concurrently with the audio process code**

Client threading model (2)

→ **A two threads model for clients:**

- **RT thread for audio process code**
- **Standard thread for notification code**
- **Is this model compatible with the way current client work?**
- **Possibly need client adaptation…**

Client failure handling

What happens when clients fail?

- **try to keep a “synchronicity” property: avoid to have client loose some cycle (other strategies are possible)**
- **possibly avoid completely stopping the audio stream**
- **let the system possibly recover during a “time-out” value**

Recovery strategy: a two step process

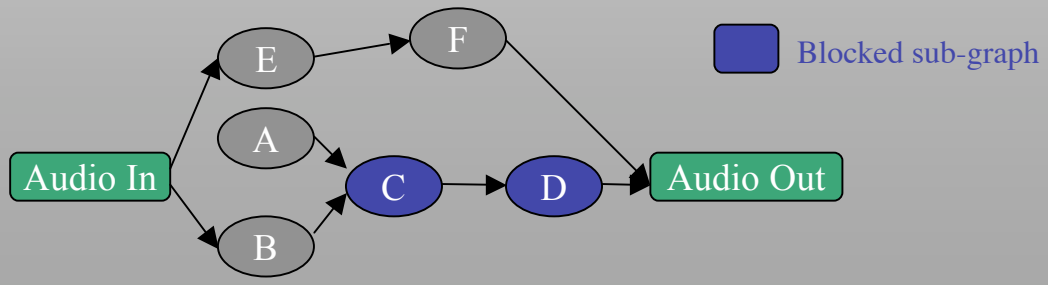
- If the graph has not been completely executed, RT thread may still access the current state, thus *do not* switch states

- Allow a client to “catch-up” if it fails only for some cycles
  - happens typically when abnormal system/scheduler latencies cause a client to be late
  - synchronization primitives “accumulate” activation signal
  - a client can possibly execute the pending cycles to catch up
  - but data may be lost

- Remove the failing client from the graph and switch to new graph state

Recovery strategy: asynchronous mode

- **"Sub-graph execution" is still possible : a "partial" output buffer can be produced, the audio stream is not interrupted**
- **Hope the client can start again and catch up during the time out**
- **Otherwise the failing client is disconnected: C in this example**



Recovery strategy: synchronous mode

- The “wait for graph end” semaphore uses the time out
- When one client is blocked, the whole graph is blocked
- The audio stream will be interrupted during the time out
- Hope the client can start again and catch up during the time out
- Otherwise the failing client is removed from the graph

XRun detection

- **Global XRun : typically driver latency, notified to all clients**
- **Asynchronous mode : individual client XRun, detected at the beginning of each cycle**
- **Synchronous mode : individual client failure will result in a complete cycle failure, thus only global XRun occur and are notified**

OSX version (1)

- C++ based version: recoded for easier experimentation
- Using mach semaphore for inter-process synchronisation
- Using MIG generated Remote Procedure Calls for server/client communications
- RT threads are “time-constraint” threads (period, computation, constraint)

OSX version (2)

- **Integration with CoreAudio : XRun detection…**
- **Asynchronous mode is preferred:**
  - **time-constraint threads are not supposed to be suspended during the cycle**
  - **actually it works much better (less XRun at small buffer size…)**
- **API not complete: transport API is still missing**
- **Tested on dual 1.8 Ghz G5**
- **Tested with “native” Jack clients: Ardour, SooperLooper, Hydrogen… as well as “Jackified” CoreAudio ones : iTunes, Max/MSP…**

Linux version

- **Using NPTL (Native POSIX Thread Library) futex based inter-process semaphore (tested by Fons Adriaensen)**
- **Recode socket based server/client communication**
- **Or possibly use the new D-BUS system ?**

Summary

. Semi data-flow model for graph execution

- Two execution mode:
  - synchronous : less latency but less robust
  - asynchronous : one buffer more latency but more robust
- New client threading model
- Client failure recovery strategy

Future ?

- Comments, feedback on the C++ version…
- What happens with the current C version ?
- What happens with pending Jack evolution: MIDI…. ?