DAW Design & Implementation #8

Integrating Control Data
Model-View-Controller #1

Agenda

- The problem with control data
- Two different designs
- Dealing with specific protocols
- Model-View-Controller (MVC) design
- History
- Basics

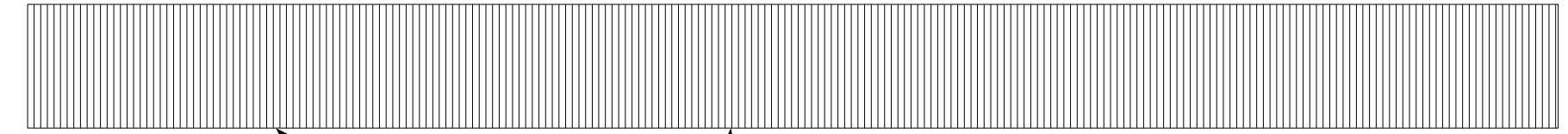
What is control data?

- Protocol: MIDI, OSC, SKINI, whatever ...
- Content: performance data, transport control, state control (e.g. solo/mute/rec-enable)
- Intent: to change/specify the behaviour of the program (notes, sounds, recording or not, etc.)

The Problem With Control Data

- Performance data is intended to influence real time thread behaviour
- Its very nature is inherently real time: “play this note starting now”
- OK, not always (e.g. SKINI, CSound scores)
- Other types of data are intended to change program behaviour in a broader sense
- May involve changes that cannot be implemented in real time

Audio & MIDI



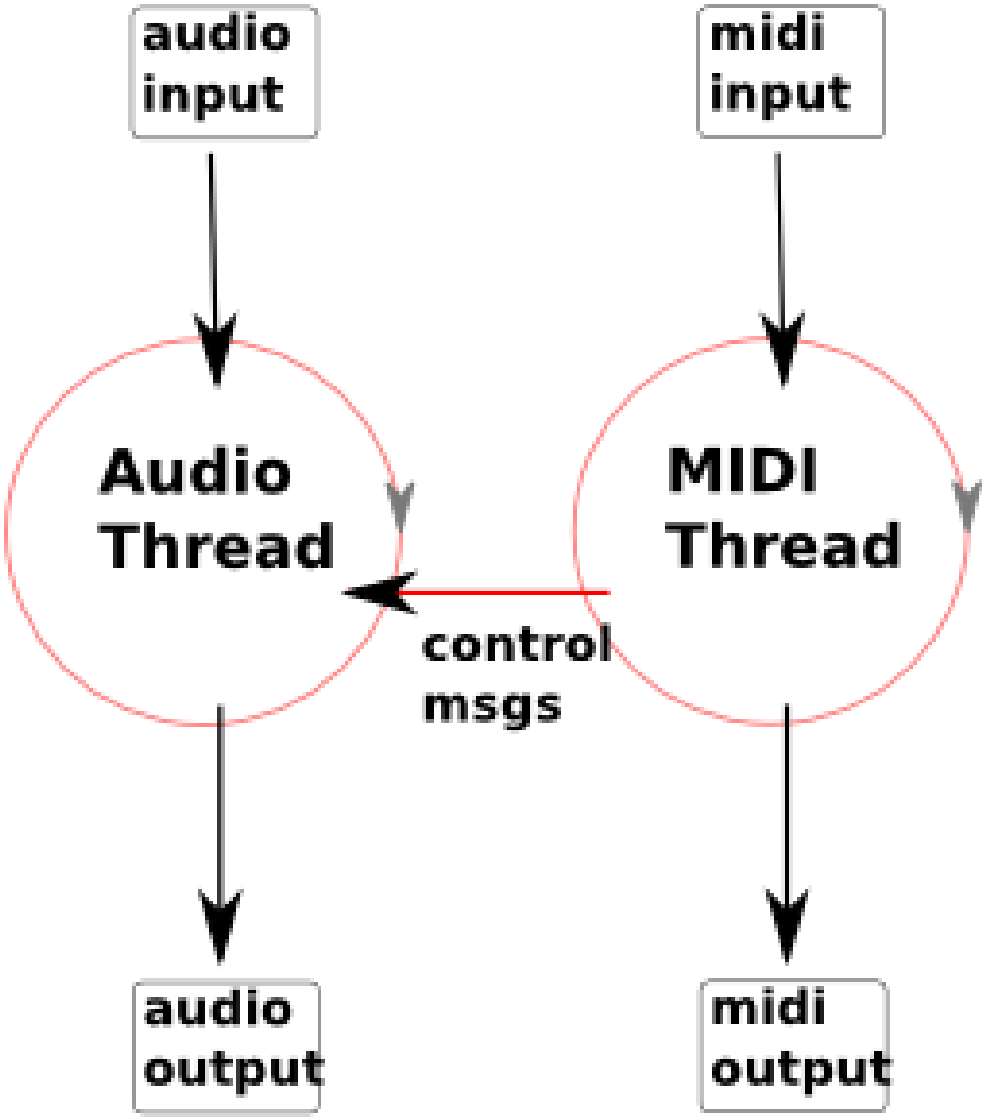
t1

t2

Audio Data (samples)

NoteOn (65,98,t1)
NoteOff (65,87,t1+80)
Controller(23,89,t1+242)
.
.
NoteOn(116,24,t1+1018)

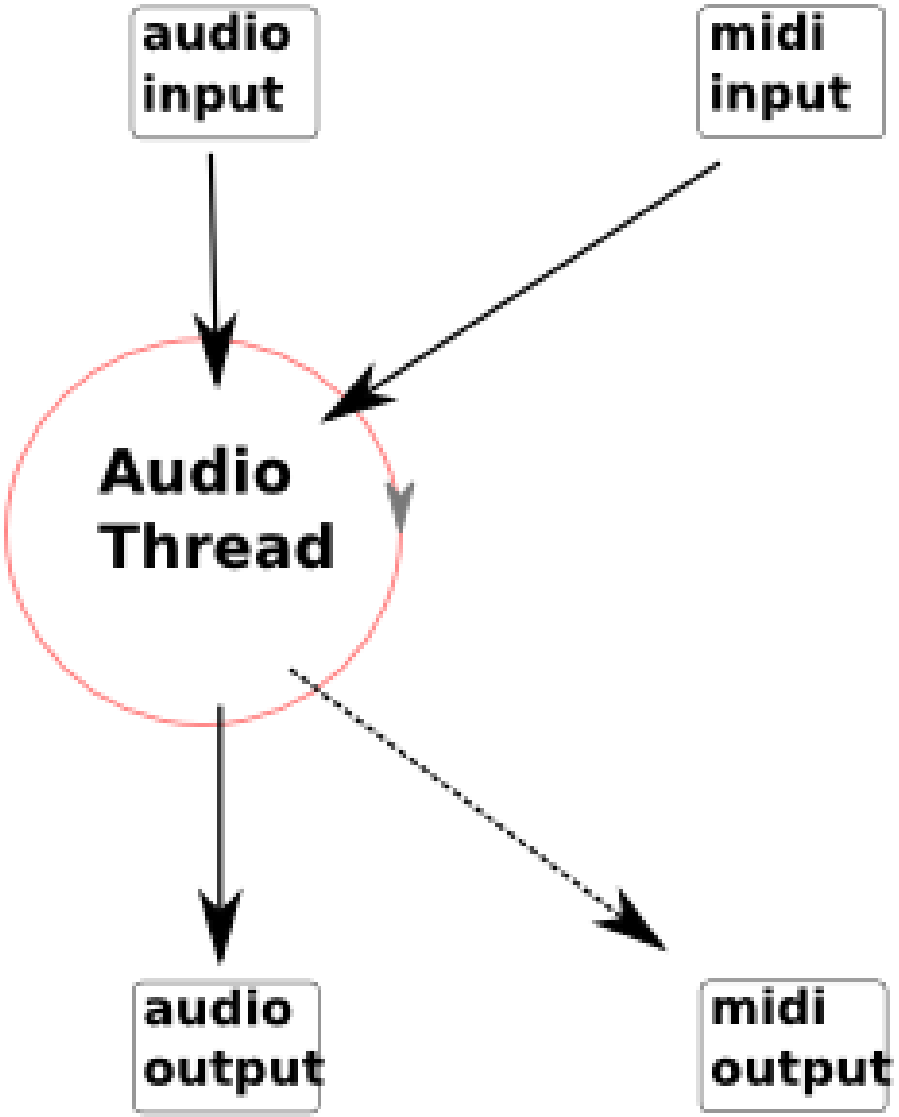
Two Scenarios: #1



That's the old way

- Pros: no MIDI data handling in the audio thread
- Cons: all control is cross-thread
- Timing becomes an issue
- No clean separation of thread function
- Welcome JACK MIDI
- All MIDI data arrives in the “RT” thread (audio)

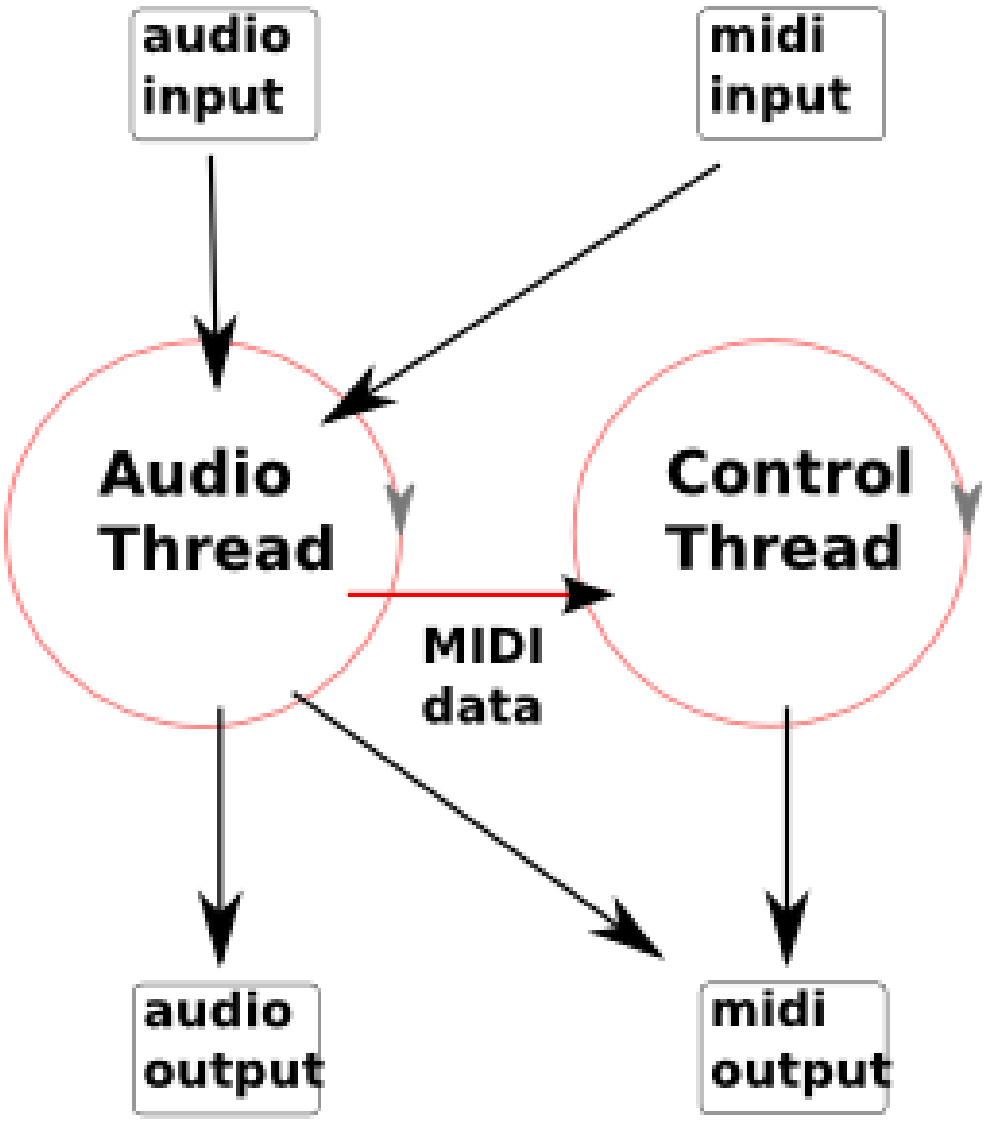
Two Scenarios: #2



Is this the new way?

- Pros: performance data is now arriving in the right thread
- Cons: other control data is arriving in the wrong thread
- We still need another thread...

The Answer (???)


audio input
midi input
Audio Thread
Control Thread
MIDI data
audio output
midi output

Model View Controller Design

- A way to design programs
- Leads to good design
- Doesn't guarantee good design
- Invented/Discovered by Trygve Reenskaug
- First applied to Smalltalk (hello, supercollider)

MVC Basics

- Divide the program into 3 parts
- Model: represents (or is) the thing the user will observe and manipulate
- View: a way for the user to see some or all of the current state of the model
- Controller: a way for the user to change the current state of the model

MVC/DAW example

- Model: audio data, playlists, signal routing, gain, plugins, panning, parameters
- View: buttons, faders, switches, text display, graphs
- Controller: buttons, faders, switches, text entry, draggable graphs
- LESSON: in real apps, the View and the Controller are often hard to separate

A More Detailed Example: mute

- Trivial implementation: there is a button; when the user presses it, we mute the track, and the button shows the status
  - So far, so good
  - Now add MIDI/OSC control over mute
- What happens to the button when a MIDI control surface sends a “mute” message?
  - Q1: where does the mute message actually arrive?

A More Detailed Example: mute

- Trivial implementation: there is a button; when the user presses it, we mute the track, and the button shows the status
  - So far, so good
  - Now add MIDI/OSC control over mute
- What happens to the button when a MIDI control surface sends a “mute” message?
  - Q1: where does the mute message actually arrive?
  - Q2: does the GUI only modify the state, or must it really just “show” it?

Mute #2

- In reality, the GUI is a View and a Controller
- There may be others (MIDI, OSC, or even multiple GUIs)
- Press a button: send a message to the Model to mute a track (Controller function)
- When model changes state, change the appearance of the button (View function)

MVC helpers

- Good MVC design requires a good way to notify Views of changes in the state of the Model
- Good notification systems will not understand the Model or the View (i.e. they have no semantics that are specific to the Model or View)
  - Ideal: “anonymous notification”
  - Model doesn't know who is listening
  - View doesn't know anything except the signals to listen to and how to get the new state (e.g. muted or not muted)

MVC in Ardour

- libardour is the Model: contains the data structures (objects) for everything that Ardour actually does.
- libardour doesn't know anything about any user interfaces (GUI or otherwise)
- gtk2_ardour is the View/Controller
- Connections between the two are made using libsigc++
- Note: this is code-level separation, not process-level separation (e.g. linuxsampler, sooperlooper)

Anonymous Notification

- Goal: a way to say “Something has changed” and have arbitrary code executed as a result
  - Way to say it: a “signal”
  - Arbitrary code: a “callback”
- When the “signal” is “emitted”, the callback is executed
- Pretty simple: the signal is just a list of pointers to functions.
  - Done!

Not So Fast!

- First, what happens if we want to know what object the change affected? (e.g. which track was muted)
- OK, add an argument that is passed to every function called
- Hmm, now we have type-safety issues (in C at least)
- Second, what happens if the View providing the callback wants to supply other information to be used when the callback is invoked?

Closures

- A very simple idea from Computer Science (aren't all the best ones?)
- A closure is just a packaging of a function with whatever other information is needed to call it (i.e. arguments)

Simple closure

```
typedef struct {
    void (*function)(int,int);
    int argument1;
    int argument2;
} closureForFunctionWith2IntArguments;

closureForFunctionWith2IntArguments c;
c.function = my_cool_function;
c.argument1 = 12;
c.argument2 = 0;
```

So ...

- Just add a 1 or more to a list
- When “emitting” the signal, go through the list
- For each closure, call the function
- Hmm ... type problems
- Forget it and use libsigc++

libsigc++

```
sigc::signal<void> aSignal;


aSignal.connect (ptr_fun (a_function));
```

Libsigc++: adding arguments

```
sigc::signal<void> aSignal;

aSignal.connect (bind (ptr_fun (a_function), 12);
```

When “a_function” is called, it will be invoked as:

```
    a_function (12);
```

Libsigc++: signals with args

```
sigc::signal<void,float> anotherSignal;
```

```
anotherSignal.connect (ptr_fun (another_function));
```

When “another_function” is called, it will be invoked as:

```
another_function (some floating point value);
```

Libsigc++: both kinds of args

```
sigc::signal<void,float> anotherSignal;
```

```
anotherSignal.connect(bind (ptr_fun(another_function), 1);
```

When “another_function” is called, it will be invoked as:

```
another_function (some floating point value, 1);
```

Actual MVC within Ardour

- Objects have setter/getter functions to access model state
- They also have 1 or more sigc++ signals that will notify any connected objects about changes in state (e.g. mute, gain, rec-enable etc. etc)
- View uses getter functions to find out what to display
- Controller uses setter functions to change state
- View hears of the change via a signal

Key Ideas

- Model has no knowledge of Views or Controllers
- Controllers change state of a model
- Views update to reflect new state
- Allows multiple views – all will update when model changes state
- Allows multiple controllers
- Maximal (?) encapsulation of object behaviour
- Model internals can be changed without View or Controller being modified

Possible Alternatives

- MVC is good
- Would process separation be an improvement?
- Model in one process, VC in another
- Replace setter/getter methods with (e.g) OSC
- Replace sigc++ with (e.g) OSC
- Multiple GUIs
- Move GUI from machine to machine

Next Week

The Tourist Guide to Ardour Data Structures