Linux audio uncovered

Graham Morrison digs into the centre of the Linux kernel to uncover why sound can be so... unsound.

There's a problem with the state of Linux audio, and it's not that it doesn't always work. The issue is that it's overcomplicated. This soon becomes evident if you sit down with a piece of paper and try to draw the relationships between the technologies involved with taking audio from a music file to your speakers: the diagram soon turns into a plate of knotted spaghetti. This is a failure because there's nothing intrinsically more complicated about audio than any other technology. It enters your Linux box at one point and leaves at another.

If we were drawing the OSI model used to describe the networking framework that connects your machine to every other machine on the network, we'd find clear strata, each with its own domain of processes and functionality. There's very little overlap in layers, and you certainly don't find end-user processes in layer seven messing with the electrical impulses of the raw bitstreams in layer one.

"It's more like the Earth's crust than the network model."

Yet this is exactly what can happen with the Linux audio framework. There isn't even a clearly defined bottom level, with several audio technologies messing around with the kernel and your hardware independently. Linux's audio architecture is more like the layers of the Earth's crust than the network model, with lower levels occasionally erupting on to the surface, causing confusion and distress, and upper layers moving to displace the underlying technology that was originally hidden. The Open Sound Protocol, for example, used to be found at the kernel level talking to your hardware directly, but it's now a compatibility layer that sits on top of *ALSA*. *ALSA* itself has a kernel level stack and a higher API for programmers to use, mixing drivers and hardware properties with the ability to play back surround sound or an MP3 codec. When most distributions stick *PulseAudio* and *GStreamer* on top, you end up with a melting pot of instability with as much potential for destruction as the San Andreas fault.

ALSA

INPUTS: *PulseAudio*, Jack, *GStreamer*, *Xine*, *SDL*, *ESD*
OUTPUTS: Hardware, OSS

As Maria von Trapp said, "Let's start at the very beginning." When it comes to modern Linux audio, the beginning is the *Advanced Linux Sound Architecture*, or *ALSA*. This connects to the Linux kernel and provides audio functionality to the rest of the system. But it's also far more ambitious than a normal kernel driver; it can mix, provide compatibility with other layers, create an API for programmers and work at such a low and stable latency that it can compete with the ASIO and CoreAudio equivalents on the Windows and OS X platforms.

ALSA was designed to replace OSS. However, OSS isn't really dead, thanks to a compatibility layer in *ALSA* designed to enable older, OSS-only applications to run. It's easiest to think of *ALSA* as the device driver layer of the Linux sound system. Your audio hardware needs a corresponding kernel module, prefixed with **snd_**, and this needs to be loaded and running for anything to happen. This is why you need an *ALSA* kernel driver for any sound to be heard on your system, and why your laptop was mute for so long before someone thought of creating a driver for it. Fortunately, most distros will configure your devices and modules automatically.

ALSA is responsible for translating your audio hardware's capabilities into a software API that the rest of your system uses to manipulate sound. It was designed to tackle many of the shortcomings of OSS (and most other sound drivers at the time), the most notable of which was that only one application could access the hardware at a time. This is why a software component in *ALSA* needs to manages audio requests and understand your hardware's capabilities.

If you want to play a game while listening to music from *Amarok*, for example, *ALSA* needs to be able to take both of these audio streams and mix them together in software, or use a hardware mixer on your soundcard to the same effect. *ALSA* can also manage up to eight audio devices and sometimes access the MIDI functionality on hardware, although this depends on the specifications of your hardware's audio driver and is becoming less important as computers get more powerful.

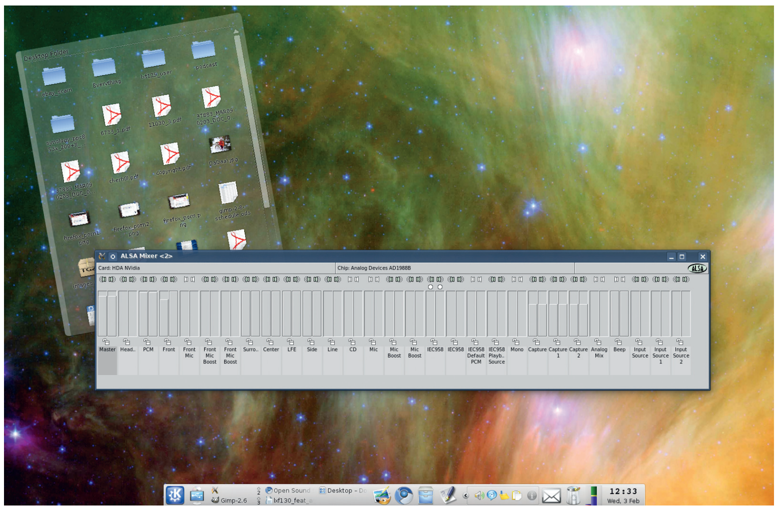
Where *ALSA* does differ from the typical kernel module/device driver is in the way it's partly user-configurable. This is where the complexity in Linux audio starts to appear, because you can alter almost anything about your *ALSA* configuration by creating your own config file – from how streams of audio are mixed together and which outputs they leave your system from, to the sample rate, bit-depth and real-time effects.

ALSA's relative transparency, efficiency and flexibility have helped to make it the standard for Linux audio, and the layer that almost every other audio framework has to go through in order to communicate with the audio hardware.

PulseAudio

INPUTS: *GStreamer*, *Xine*, *ALSA*
OUTPUTS: *ALSA*, Jack, *ESD*, OSS

If you're thinking that things are going to get easier with *ALSA* safely behind us, you're sadly mistaken. *ALSA* covers most of the nuts and bolts of getting audio into and out of your machine, but you must navigate another layer of complexity. This is the domain of *PulseAudio* – an attempt to bridge the gap between hardware and software capabilities, local and remote machines, and the contents of audio streams. It does for networked audio what *ALSA* does for multiple soundcards, and has become something of a standard across many Linux distros because of its flexibility.



› **This image highlights exactly what's wrong with Linux audio. This is the hideously complicated default view from the *ALSA* mixer for a typical sound device.**

As with *ALSA*, this flexibility brings complexity, but the problem is compounded by *PulseAudio* because it's more user-facing. This means normal users are more likely to get tangled in its web. Most distros keep its configuration at arm's length; with the latest release of Ubuntu, for example, you might not even notice that *PulseAudio* is installed. If you click on the mixer applet to adjust your soundcard's audio level, you get the *ALSA* panel, but what you're really seeing is *ALSA* going to *PulseAudio*, then back to *ALSA* – a virtual device.

At first glance, *PulseAudio* doesn't appear to add anything new to Linux audio, which is why it faces so much hostility. It doesn't simplify what we have already or make audio more robust, but it does add several important features. It's also the catch-all layer for Linux audio applications, regardless of their individual capabilities or the specification of your hardware.

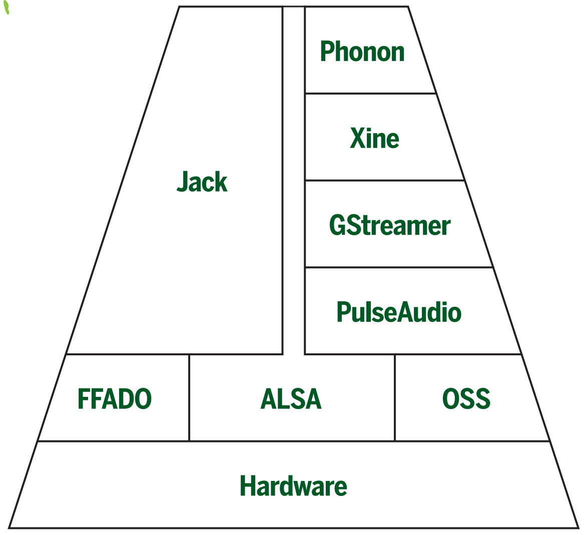
If all applications used *PulseAudio*, things would be simple. Developers wouldn't need to worry about the complexities of other systems, because *PulseAudio* brings cross-platform compatibility. But this is one of the main reasons why there are so many other audio solutions. Unlike *ALSA*, *PulseAudio* can run on multiple operating systems, including other POSIX platforms and Microsoft Windows. This means that if you build an application to use *PulseAudio* rather than *ALSA*, porting that application to a different platform should be easy.

"You can alter almost anything about your ALSA configuration."

But there's a symbiotic relationship between *ALSA* and *PulseAudio* because, on Linux systems, the latter needs the former to survive. *PulseAudio* configures itself as a virtual device connected to *ALSA*, like any other piece of hardware. This makes *PulseAudio* more like Jack, because it sits between *ALSA* and the desktop, piping data back and forth transparently. It also has its own terminology. Sinks, for instance, are the final destination. These could be another machine on the network or the audio outputs on your soundcard courtesy of the virtual *ALSA*. The parts of *PulseAudio* that fill these sinks are called 'sources' – typically audio-generating applications on your system, audio inputs from your soundcard, or a network audio stream being sent from another *PulseAudio* machine.

Unlike Jack, applications aren't directly responsible for adding and removing sources, and you get a finer degree of

»



› **Here's a simplified view of the audio layers typically used in Linux. The deeper the layer, the closer to the hardware it is.**

control over each stream. Using the *PulseAudio* mixer, for example, you can adjust the relative volume of every application passing through *PulseAudio*, regardless of whether that application features its own slider or not. This is a great way of curtailing noisy websites.

GStreamer

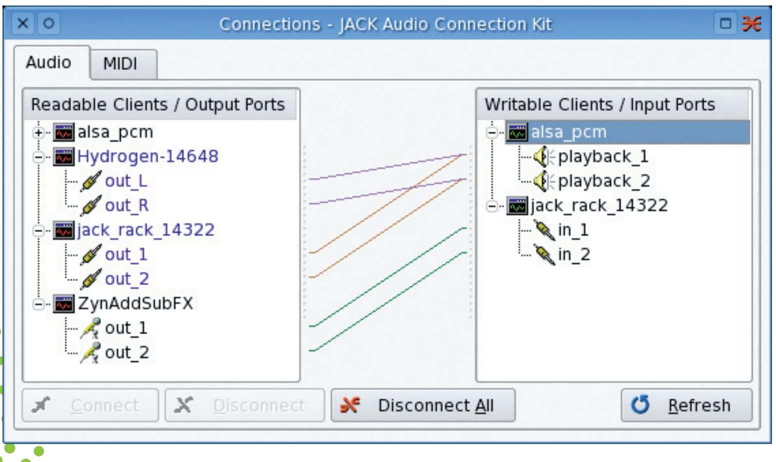
INPUTS: *Phonon*
OUTPUTS: *ALSA*, *PulseAudio*, Jack, *ESD*

With *GStreamer*, Linux audio starts to look even more confusing. This is because, like *PulseAudio*, *GStreamer* doesn't seem to add anything new to the mix. It's another multimedia framework and gained a reasonable following of developers in the years before *PulseAudio*, especially on the Gnome desktop. It's one of the few ways to install and use proprietary codecs easily on the Linux desktop. It's also the audio framework of choice for *GTK* developers, and you can even find a version handling audio on the Palm Pre.

GStreamer slots into the audio layers above *PulseAudio* (which it uses for sound output on most distributions), but below the application level. *GStreamer* is unique because it's not designed solely for audio – it supports several formats of streaming media, including video, through the use of plugins.

MP3 playback, for example, is normally added to your system through an additional codec download that attaches itself as a *GStreamer* plugin. The commercial *Fluendo* MP3 decoder, one of the only officially licenced codecs available for Linux, is supplied as a *GStreamer* plugin, as are its other proprietary codecs, including MPEG-2, H.264 and MPEG.

› **With Jack, you can connect the audio output from applications to the audio input of others manually – just like in a real recording studio.**



Jack

INPUTS: *PulseAudio*, *GStreamer*, *ALSA*,
OUTPUTS: OSS, FFADO, *ALSA*

Despite the advantages of open configurations such as *PulseAudio*, they all pipe audio between applications with the assumption that it will proceed directly to the outputs. Jack is the middle layer – the audio equivalent of remote procedure calls in programming, enabling audio applications to be built from a variety of components.

The best example is a virtual recording studio, where one application is responsible for grabbing the audio data and another for processing the audio with effects, before finally sending the resulting stream through a mastering processor to be readied for release. A real recording studio might use a web of cables, sometimes known as jacks, to build these connections. Jack does the same in software.

Jack is an acronym for 'Jack Audio Connection Kit'. It's built to be low-latency, which means there's no undue processing performed on the audio that might impede its progress. But for Jack to be useful, an audio application has to be specifically designed to handle Jack connections. As a result, it's not a simple replacement for the likes of *ALSA* and *PulseAudio*, and needs to be run on top of another system that will generate the sound and provide the physical inputs.

With most Jack-compatible applications, you're free to route the audio and inputs in whichever way you please. You could take the output from *VLC*, for example, and pipe it directly into *Audacity* to record the stream as it plays back. Or you could send it through *JackRack*, an application that enables you to build a tower of real-time effects, including pinging delays, cavernous reverb and voluptuous vocoding.

This versatility is fantastic for digital audio workstations. *Ardour* uses Jack for internal and external connections, for instance, and the *Jamin* mastering processor can only be used as part of a chain of Jack processes. It's the equivalent of having full control over how your studio is wired. Its implementation has been so successful on the Linux desktop that you can find Jack being put to similar use on OS X.

FFADO

INPUTS: Jack
OUTPUTS: Audio hardware

In the world of professional and semi-professional audio, many audio interfaces connect to their host machine using a FireWire port. This approach has many advantages. FireWire is fast and devices can be bus powered. Many laptop and desktop machines have FireWire ports without any further modification, and the standard is stable and mostly mature. You can also take FireWire devices on the road for remote recording with a laptop and plug them back into your desktop machine when you get back to the studio.

But unlike USB, where there's a standard for handling audio without additional drivers, FireWire audio interfaces need their own drivers. The complexities of the FireWire protocol mean these can't easily create an *ALSA* interface, so they need their own layer. Originally, this work fell to a project called FreeBOB. This took advantage of the fact that many FireWire audio devices were based on the same hardware.

FFADO is the successor to FreeBOB, and opens the driver platform to many other types of FireWire audio interface. Version 2 was released at the end of 2009, and includes support for many units from the likes of Alesis, Apogee, ART, CME, Echo, Edirol, Focusrite, M-Audio, Mackie, Phonic and Terratec. Which devices do and don't work is rather random, so you need to check before investing in one, but many of these manufacturers have helped driver development by providing devices for the developers to use and test.

Another neat feature in FFADO is that some the DSP mixing features of the hardware have been integrated into the driver, complete with a graphical mixer for controlling the balance of the various inputs and outputs. This is different to the ALSA mixer, because it means audio streams can be controlled on the hardware with zero latency, which is exactly what you need if you're recording a live performance.

Unlike other audio layers, FFADO will only shuffle audio between Jack and your audio hardware. There's no back door to *PulseAudio* or *GStreamer*, unless you run those against Jack. This means you can't use FFADO as a general audio layer for music playback or movies unless you're prepared to mess around with installation and Jack. But it also means that the driver isn't overwhelmed by support for various different protocols, especially because most serious audio applications include Jack support by default. This makes it one of the best choices for a studio environment.

Xine

INPUTS: *Phonon*
OUTPUTS: *PulseAudio*, *ALSA*, ESD

We're starting to get into the niche geology of Linux audio. *Xine* is a little like the chalk downs; it's what's left after many other audio layers have been washed away. Most users will recognise the name from the very capable DVD movie and media player that most distributions still bundle, despite its age, and this is the key to *Xine's* longevity.

When *Xine* was created, the developers split it into a back-end library to handle the media, and a front-end application for user interaction. It's the library that's persisted, thanks to its ability to play numerous containers, including AVI, Matroska and Ogg, and dozens of the formats they contain, such as AAC, Flac, MP3, Vorbis and WMA. It does this by harnessing the powers of many other libraries. As a result, *Xine* can act as a catch-all framework for developers who want to offer the best range of file compatibility without worrying about the legality of proprietary codecs and patents.

Xine can talk to *ALSA* and *PulseAudio* for its output, and there are still many applications that can talk to *Xine* directly. The most popular are the *Gxine* front-end and *Totem*, but *Xine* is also the default back-end for KDE's *Phonon*, so you can find it locked to everything from *Amarok* to *Kaffeine*.

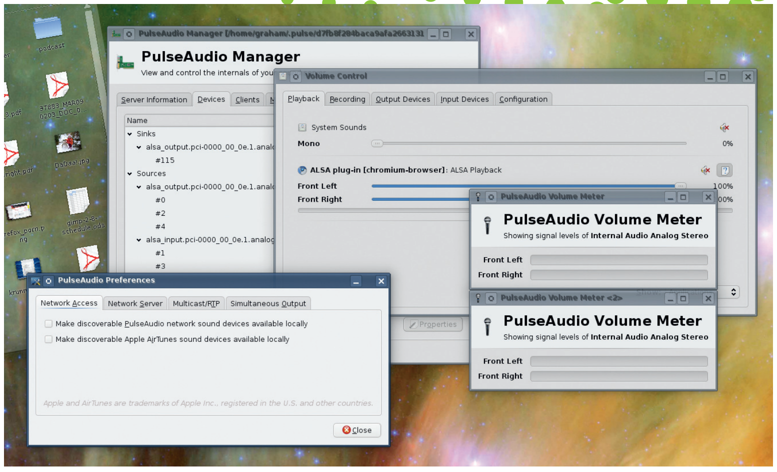
Phonon

INPUTS: *Qt* and KDE applications
OUTPUTS: *GStreamer*, *Xine*

Phonon was designed to make life easier for developers and users by removing some of the system's increasing complexity. It started life as another level of audio abstraction for KDE 4 applications, but it was considered such a good idea that *Qt* developers made it their own, pulling it directly into the *Qt* framework that KDE itself is based on.

This had great advantages for developers of cross-platform applications. It made it possible to write a music player on Linux with *Qt* and simply recompile it for OS X and Windows without worrying about how the music would be played back, the capabilities of the sound hardware being used, or how the destination operating system would handle audio. This was all done automatically by *Qt* and *Phonon*, passing the audio to the CoreAudio API on OS X, for example, or DirectSound on Windows. On the Linux platform (and unlike the original KDE version of *Phonon*), *Qt's Phonon* passes the audio to *GStreamer*, mostly for its transparent codec support.

Phonon support is being quietly dropped from the *Qt* framework. There have been many criticisms of the system, the most common being that it's too simplistic and offers nothing new, although it's likely that KDE will hold on to the framework for the duration of the KDE 4 lifecycle.



Make discoverable PulseAudio network sound devices available locally
Make discoverable Apple AirTunes sound devices available locally
Apple and AirTunes are trademarks of Apple Inc., registered in the U.S. and other countries.

› ***PulseAudio* is powerful, but often derided for making Linux audio even more complicated.**

The rest of the bunch

There are many other audio technologies, including *ESD*, *SDL* and *PortAudio*. *ESD* is the *Enlightenment Sound Daemon*, and for a long time it was the default sound server for the Gnome desktop. Eventually, Gnome was ported to use *libcanberra* (which itself talks to *ALSA*, *GStreamer*, OSS and *PulseAudio*) and *ESD* was dropped as a requirement in April 2009. Then there's *Arts*, the KDE equivalent of *ESD*, although it wasn't as widely supported and seemed to cause more problems than it solved. Most people have now moved to KDE 4, so it's no longer an issue.

"SDL supports plenty of features, and is both mature and stable."

SDL, on the other hand, is still thriving as the audio output component in the *SDL* library, which is used to create hundreds of cross-platform games. It supports plenty of features, and is both mature and stable.

PortAudio is another cross-platform audio library that adds SGI, Unix and Beos to the mix of possible destinations. The most notable application to use *PortAudio* is the *Audacity* audio editor, which may explain its sometimes unpredictable sound output and the quality of its Jack support.

And then there's OSS, the Open Sound System. It hasn't been a core Linux audio technology since version 2.4 of the kernel, but there's just no shaking it. This is partly because so many older applications are dependent on it and, unlike *ALSA*, it works on systems other than Linux. There's even a FreeBSD version. It was a good system for 1992, but *ALSA* is nearly always recommended as a replacement.

OSS defined how audio would work on Linux, and in particular, the way audio devices are accessed through the ioctl tree, as with **/dev/dsp**, for example. *ALSA* features an OSS compatibility layer to enable older applications to stick to OSS without abandoning the current *ALSA* standard.

The OSS project has experimented with open source and proprietary development, and is still being actively developed as a commercial endeavour by 4 Front Technologies. Build 2002 of OSS 4.2 was released in November 2009. LXF