PIPEWIRE: A LOW-LEVEL MULTIMEDIA SUBSYSTEM

Wim Taymans *

Principal Software Engineer
Red Hat, Spain
wim.taymans@gmail.com

ABSTRACT

PipeWire is a low-level multimedia library and daemon that facilitates negotiation and low-latency transport of multimedia content between applications, filters and devices. It is built using modern Linux infrastructure and has both performance and security as its core design guidelines. The goal is to provide services such as JACK and PulseAudio on top of this common infrastructure. PipeWire is media agnostic and supports arbitrary compressed and uncompressed formats. A common audio infrastructure with backwards compatibility that can support Pro Audio and Desktop Audio use cases can potentially unify the currently fractured audio landscape on Linux desktops and workstations and give users and developers a much better audio experience.

1. INTRODUCTION

In recent years, a lot of effort has been put into improving the delivery method for applications on Linux. Both Flatpak [1] (backed by Red Hat and others) and snappy [2] (backed by Canonical) aim to improve application dependencies, delivery and security.

Due to the increased security policy of these sandboxed applications, no direct access to system devices is allowed. Access to devices needs to be mediated by a portal and controlled by a daemon.

Linux and other operating systems have traditionally used a daemon to control audio devices. For consumer audio, the Linux desktop has settled around PulseAudio [3] and for Pro-Audio it has adopted JACK [4].

The initial motivation for PipeWire [5] in 2017, came from a desire to support camera capture in a sandboxed environment such as Flatpak. PipeWire was initially conceived as a daemon to decouple access from the camera and the application, not very different from the existing audio daemons. Later on, the design solidified and with input from the LAD community it went through a couple of rewrites and gained audio functionality as well.

PipeWire provides a unified framework for accessing multimedia devices, implementing filters and sharing multimedia content between applications in an efficient and secure way. This framework can be used to implement various services such as Camera access from browsers, Screen sharing, audio server, etc.. The design allows to run PulseAudio and JACK applications on top of a common framework, essentially providing a way to unify the Linux Audio stack.

This paper will focus on the Audio infrastructure that PipeWire implements.

* This work was supported by Red Hat

2. LINUX AUDIO LANDSCAPE

Audio support on Linux first appeared with the Open Sound System (OSS) [6] and was until the 2.4 kernel the only audio API available on Linux. It was based around the standard Unix open/close/read-/write/ioctl system calls.

OSS was replaced by the Advanced Linux Sound Architecture (ALSA) [7]from Linux 2.5. ALSA improved on the OSS API and included a user space library that abstracted many of the hardware details. The ALSA user-space library also includes a plugin infrastructure that can be used to create new custom devices and plugins. Unfortunately, the plugin system is quite static and requires editing of configuration files.

OSS — and also ALSA — both suffer from the fact that only one application can open a device at a time. Some hardware can solve this by doing mixing in the audio card itself but most consumer cards or even pro audio cards don't have this functionality. ALSA implements a software mixer as a plugin (Dmix) but its implementation is lacking and its setup inflexible.

2.1. First sound servers

EsoundD (or ESD) was one of the first sound servers. It was developed for Enlightenment and was later adopted by GNOME. It received audio from multiple clients over a socket and mixed the samples before writing to the hardware device. Backend modules could be written for various sound APIs such as ALSA and OSS.

The first sound servers used TCP as a transport mechanism and were not designed to provide low-latency audio. Applications were supposed to send samples to the server at a reasonable speed with some limited feedback about the fill levels in the server.

BSD has another audio API called `sndio` [8]. This is a very simple audio API that can also handle midi. It is based on Unix pipes to transport audio and has, like ESD, no real support for low-latency audio.

2.2. Pro audio with JACK

The JACK Audio Connection Kit (JACK) was developed by Paul Davis in 2002 based on the audio engine in Ardour. It provides real-time and low-latency connections between applications for audio and midi.

JACK maintains a graph of applications (clients) that are connected using ports. In contrast to the previous audio servers, JACK will use the device interrupt to wake up each client in the graph in turn to process data. This makes it possible to keep the delay between processing and recording/playback very low.

There are 2 implementations of the JACK API with different features. Work is ongoing to bring the JACK2 implementation to the same level as JACK1, eventually rendering JACK1 obsolete.

JACK is missing features that are typically needed for regular desktop users such as format support, power saving, dynamic devices, volume control, security etc.

2.3. Consumer audio with PulseAudio

PulseAudio is a modern modular sound server. In contrast to other sound servers on Linux it handles the routing and setup of multiple devices automatically and dynamically. One can connect a bluetooth device and have the sound be routed to it automatically, for example.

It is possible to write rules into a policy module to perform various tasks based on events in the system. One can for example, lower or pause audio streams when an incoming call is received.

PulseAudio is optimized for power saving and does not handle low-latency audio very well, the code paths to wake up a client are in general too CPU hungry.

Most desktops nowadays install PulseAudio by default. And it is possible to let PulseAudio and JACK somewhat coexist. PulseAudio can automatically become a JACK client when needed although this will cause high CPU load with low-latency JACK setups.

3. PIPEWIRE MEDIA SERVER

When designing the audio infrastructure for PipeWire we need to build upon the lessons learned from JACK and PulseAudio. We will present the current design and how each part improves upon the JACK and PulseAudio design.

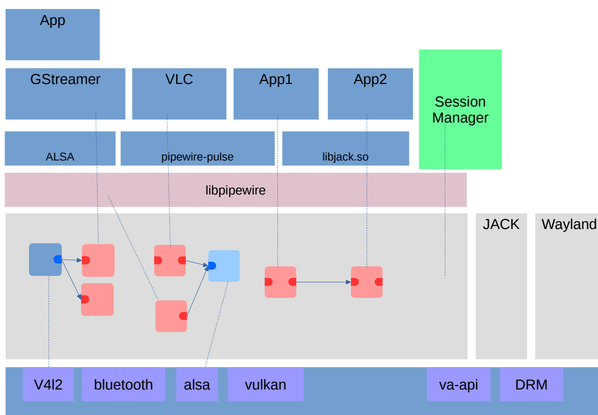

Figure 1: *Overview*

Figure 1 shows where PipeWire is situated in the software stack. It sits right between the user-space API to access the kernel drivers and the applications. It takes on a similar role as PulseAudio or JACK but also includes video devices, midi, Firewire and bluetooth. It allows application to exchange media with each other or with devices.

Applications are not supposed to directly access the devices but go through the PipeWire API. There is a replacement JACK and PulseAudio server that go through PipeWire to provide compatibility with existing applications. There is no urgent reason to port applications to PipeWire unless they would want to use newer features that cannot be implemented in the other APIs.

The core functionality of PipeWire is simple and consists of:

- Provide a set of core objects. This includes: Core, Client, Device, Node, Port, Module, Factory, Link along with some helper objects.
- Load modules to create factories, policy or other objects in the PipeWire daemon or client.
- Allow clients to connect and enforce per client permissions on objects.
- Allow clients to introspect objects in the daemon. This includes enumerating object properties.
- Allow clients to call methods on objects. New objects can be created from a factory. This includes creating a link between ports or creating client controlled objects.
- Manage the negotiation of formats and buffers between linked ports.
- Manage the dataflow in the graph.

An important piece of the infrastructure is the session manager. It includes all the specific configuration and policy for how the devices will be managed in the PipeWire graph. It is expected that this component will be use-case specific and that it will interface with the configuration system of the environment. PipeWire provides a modular example session manager and work is ongoing to create an alternative session manager called WirePlumber [9].

In the next subsections we cover the various requirements and how they are implemented in PipeWire.

3.1. IPC

A sound/media server needs to have an efficient and extensible IPC mechanism between client and server. The PipeWire IPC (inter process communication) system was inspired by Wayland [10]. It exposes a set of objects that can have properties, methods and that can emit events.

The protocol is completely asynchronous, this means that method calls do not have a return value but will trigger one or more events asynchronously later with a result. This also makes it possible to use the protocol over a network without blocking the application.

PipeWire has a set of core built-in interfaces, such as Node, Port and Link that can be mapped directly to JACK Client, Port and connections. It is also possible to define new interfaces and implement them into a module. This makes it possible to extend the number of interfaces and evolve the API as time goes by.

Extensibility of the protocol has been lacking in JACK and to some extend PulseAudio as well.

3.2. Configuration/Policy

Configuration of the Devices and nodes in the PipeWire daemon as well as the routing should be performed by a separate module or even an external session manager.

With PulseAudio, the setup and policy was loaded into the daemon with modules and requires editing of configuration files to change. Modules can also only be developed inside the PulseAudio repository, which makes them very inflexible and not adaptable to the specific desktop environment.

JACK has very limited setup, it can normally only load and configure 1 hardware device for capture and playback. It is up to other processes (session managers, control applications) to add extra devices dynamically (`netjack`, `zita-a2j`, `zita-n2j`, ...).

PipeWire chooses the external session manager setup, like JACK but makes it possible to choose what services to run in the daemon and which in the session manager. Devices, for example, can run inside the daemon for better performance or outside of the daemon for more flexibility (Bluetooth devices need encoding/decoding that is better run outside of the daemon).

The session manager can export any kind of device, including Bluetooth, Firewire, ALSA, video4linux and so on.

3.3. Security

JACK and PulseAudio make it possible for clients to interfere with other clients or even read and modify their data. PipeWire fixes this problem with a permission system that is enforced at the PipeWire core level.

When a client connects, it can be frozen until permissions are configured on it. This is usually done by an access module when it detects a sandboxed client. Usually a session manager will configure permissions on a client based on its stored permissions or based on user interaction.

PipeWire enforces that clients can't see or interact with objects for which they don't have READ permission. Client can't call methods on objects without EXECUTE permissions and WRITE permission is needed to change properties on an object.

It is also important that clients can only see the shared memory they need. This is implemented in PipeWire by only handing `memfd` file descriptors to clients that require the data. Seals are used to make sure that clients can't truncate or grow the memory in any way and cause other clients or the daemon to crash.

3.4. Format negotiation

PipeWire uses the same format description as used in GStreamer [11]. This allows it to express media formats with properties, ranges and enumerations. It is possible to easily find a common format between ports by doing a generic intersection of formats.

Format conversion, however, is not something that should be done often in a real-time, low-latency pipeline. It should typically only be done when writing to or reading from the actual hardware. The PipeWire audio processing graph uses a common single format between all the processing nodes. The format is not hard-coded into PipeWire but configured by the session manager and is currently the same format as used by JACK: Float 32 bits mono samples.

PipeWire uses a generic control format to transport midi and other control messages. This can include timed property updates, OSC or CV values.

Flexible format negotiation is a requirement to implement features like pass-through over HDMI or AAC decoding on the bluetooth devices. The session manager will usually define how this will work, for example, pass-through will require exclusive access to the device because mixing of encoded formats is not possible.

3.5. Dataflow

After a format is negotiated, PipeWire negotiates a set of buffers backed by memory in `memfd`. These buffers are then shared between nodes and ports that need them by passing the file descriptor around. `eventfd` is used to wakeup nodes when they need to process input buffers and produce output buffers.

`timerfd` is used to measure when a devices will be empty/-filled. The timeout is adjusted based on the fill level of the device and a DLL. By using a timer, we can also dynamically adjust the period size based on client requirements. It is also possible to write the device wakeup using the traditional IRQ based approach but that does not provide flexible period adjustments.

When a device needs more data (or has more data, in case of a source), the graph is woken up. PipeWire uses the same concepts as JACK2 to schedule the processing graph. It keeps track of dependencies between nodes and nodes are informed about the peer nodes they are linked to. When processing starts, all nodes without dependencies are scheduled (sources). When they complete, dependencies are satisfied on their peer nodes, which are then scheduled, and so on until the whole graph is completed. Nodes that complete can directly wake up their peers by signaling the `eventfd` without having to wake up the PipeWire daemon.

This allows for the same latency and complexity as JACK and significantly better performance than PulseAudio.

3.6. Automatic slaving

PipeWire will automatically manage the master/slave relationship between devices. For this it uses a priority property configured on the device node by the session manager.

Devices are only slaved to each other when their graphs are interconnected in some way. This is an improvement compared to JACK, which requires all devices to be slaved to one master, even if they don't need to be. It allows PipeWire to avoid resampling in many cases.

The clock slaving and resampling algorithm is inspired by zita-ajbridge [12]. It however runs in a single thread and uses a DLL to drive the resampler by matching its device fill level to the graph period size. This results in exceptionally good rate matching, far superior to what PulseAudio manages and with lower latency than what zita-ajbridge does.

3.7. Transport

PipeWire expands on the JACK transport feature with the following additions:

- multiple transports at the same time. Each driver has its own transport, when drivers are slaved, the transport of the master becomes the active one. This makes it possible to avoid slaving and resampling when the driver graphs are not linked in any way.
- Seeking is supported in other formats than audio samples. Seeking in beats or bars is possible.
- Clients can know about new position changes in advance. There is a queue of pending position changes that clients can look at.
- Sample accurate looping.

4. SESSION MANAGER

The PipeWire daemon is usually configured to start up with a minimal set of modules. All devices and policy are typically loaded and configured by an external session manager. This usually include a factory for devices and a factory for making links.

PipeWire includes a modular example session manager that can be used as a basis for a custom session manager.

The session manage usually also implement the session manager extension API that introduces concepts of Session/Endpoint/EndpointStream and EndpointLink. These interfaces are used to group and configure nodes in the graph and allows PipeWire/SessionManager to provide similar concepts to what PulseAudio uses.

5. API SUPPORT

Legacy application should run unmodified on a PipeWire system. Depending on the API, a plugin or a replacement library is used for this purpose.

5.1. ALSA

There is an ALSA plugin that interfaces directly with PipeWire to support older ALSA-only applications. See Figure-3 for an example of aplay streaming to PipeWire.

5.2. PulseAudio

PulseAudio support was initially implemented with a reimplementation of `libpulse.so` and some other pulse libraries that interfaces with PipeWire directly. This however proved to be more complicated and error prone than expected.

The lastest PipeWire version implements PulseAudio support with a minimal reimplementation of the PulseAudio protocol in a separate daemon. This provides excellent compatibility even for Flatpak applications and turns out to be considerably less complicated to implement.

See Figure 2 for a screenshot of `pavucontrol` running on top of the PipeWire PulseAudio replacement daemon.

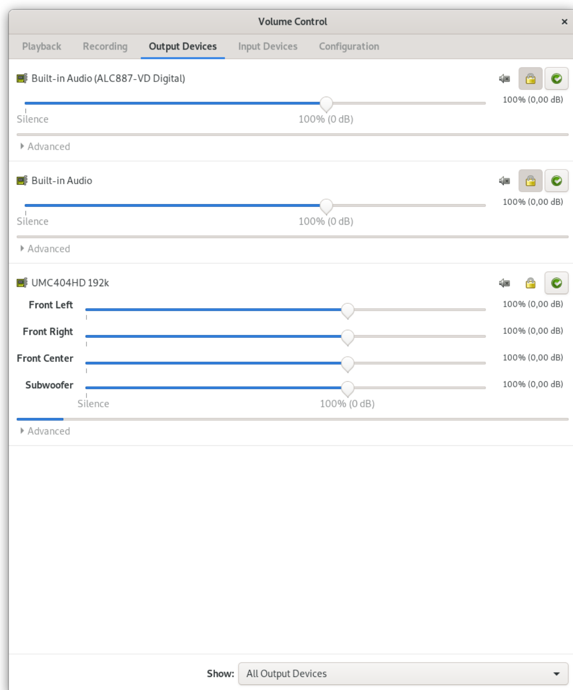

Figure 2: *`pavucontrol` on PipeWire*

5.3. JACK

JACK is supported with a custom `libjack.so` library that maps all jack method calls to equivalent PipeWire methods. See Figure-3 for an example of catia running on top of the PipeWire `libjack.so` replacement. The figure also shows how VLC (using the PulseAudio API), aplay (using the ALSA plugin) and paplay (using the PulseAudio API) can coexist with JACK applications.

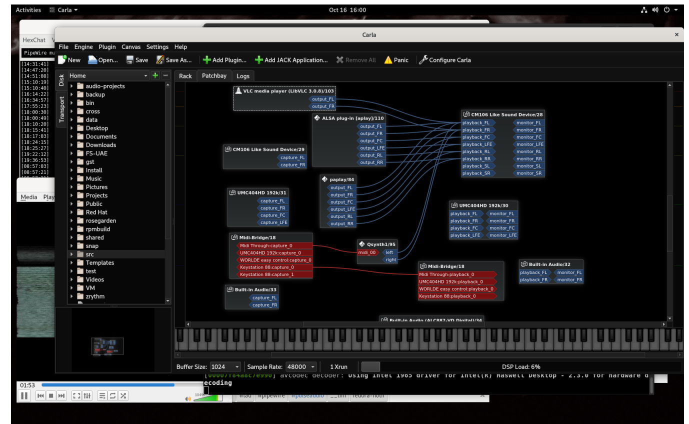

Figure 3: *Catia on PipeWire*

6. USE CASES

In addition to the audio use case that we covered in the previous section, in this section we briefly touch upon the other use cases that PipeWire handles.

6.1. Camera access

In sandboxed applications, it is not allowed to directly access the video camera. Browsers provide a custom dialog to mediate access to cameras but this task would be better handled by the lower layers in order to have a unified access control mechanism but also a common video processing graph.

PipeWire can provide a video4linux source that applications can use to capture video from the camera. This has many benefits such as:

- Access can be controlled by PipeWire. Revoking access is easy.
- Resolutions and format are managed by the session manager. Based on the profile and requirements of the apps using the camera.
- Filters can be applied.
- The camera can be shared between applications.

GNOME has created a portal DBus API [13] to negotiate access with the camera (what camera to use) and create a session with limited permissions for this stream.

Figure-4 shows Cheese and a GStreamer pipeline sharing the captured video of a `video4linux` camera served through PipeWire.

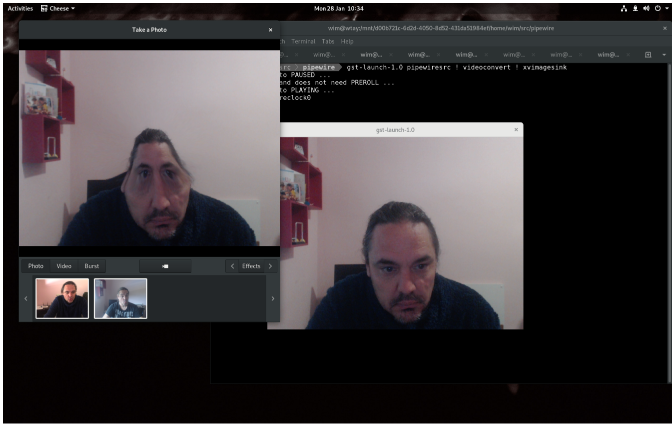

Figure 4: *Camera access and sharing*

6.2. Screen sharing

Under Wayland, it is for security reasons not possible to grab the contents of the screen. This makes it impossible to implement screen sharing or remote desktop on top of Wayland without some extra work.

GNOME has implemented a portal (DBus API) that can be used to request a PipeWire stream of the desktop. The portal will ask the user what kind of screen sharing to activate (windows, area or whole desktop along with what monitor etc) and will then set up a PipeWire session with the stream. The `fd` of the session is passed to the application. Using the PipeWire security model, only this stream is visible to the application and data can flow between the compositor and the application. See Figure-5 to see a GStreamer pipeline rendering the captured screen of a Wayland session.

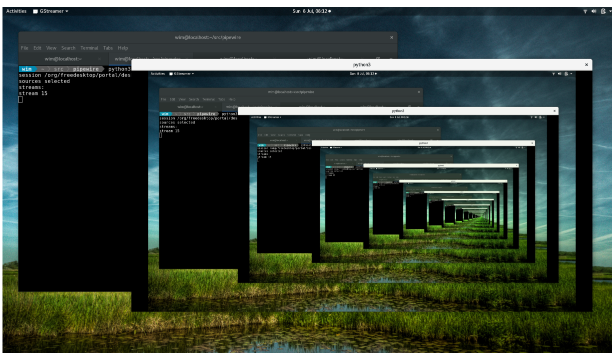

Figure 5: *Wayland screen sharing*

6.3. Video processing

Some effort has been put into the Video processing part of PipeWire. Currently there is a Vulkan compute source that can generate video in RGBA float 32 linear light format. We expect video filters to be made at a later stage, enabling the same kind of features JACK gives but on video streams.

6.4. Adoption

PipeWire has been in Fedora 27 since 2017 to implement Wayland screen sharing. FireFox, Chrome (WebRTC) have support to implement the screen sharing with native PipeWire API using the DBus portal.

Since Fedora 32 (early 2020), the redesigned version 3 with audio support has been shipped but not enabled by default.

On September 4th 2020 [14], a tech preview can be enabled in Fedora 32 and Fedora 33 to test out the audio functionality. This resulted in quite a few new features and bugfixes reported by early testers.

Currently, a plan is developing to try to enable PipeWire as the default Audio service in Fedora 34 (april 2021) and to phase out PulseAudio and JACK.

7. CONCLUSIONS

We showed how PipeWire provides a performant and secure multimedia subsystem in Linux. With lessons learned from existing consumer and pro audio solutions, PipeWire unifies the audio stack and provides a future proof foundation for all kinds of new exciting multimedia applications.

Future work will involve deploying PipeWire in distros and learning how to improve the design. More research and experience is needed for writing the session manager and how this will integrate with the desktop configuration.

More work is being done on experimenting with scripting languages to define the policy and routing in a flexible and reusable way.

8. ACKNOWLEDGEMENTS

Many thanks to the LAD community (and in particular Robin Gareus, Paul Davis, Len Ovens and Filipe Coelho) for letting me pick their brains and putting me on the right track.

Many thanks to my employer Red Hat, who sponsored the development of PipeWire.

9. REFERENCES

[1] Flatpak Comunity, “Flatpak,” `https://github.com/flatpak/flatpak`.

[2] Canonical, “Snappy,” `https://snapcraft.io`.

[3] Lennart Poettering et al., “Pulseaudio,” `https://pulseaudio.org/`.

[4] Paul Davis, “Jack audio connection kit,” `http://jackaudio.org/`, 2002.

[5] Wim Taymans, “Pipewire - multimedia processing,” `https://pipewire.org`, 2017, [Online].

[6] Hannu Savolainen, “Open sound system,” `http://www.opensound.com`.

[7] Jaroslav Kysela, “Advanced linux sound architecture,” `http://alsa-project.org`, 1998.

[8] Alexandre Ratchov and Jacob Meuser, “sndio: Openbsd sound system,” `http://www.sndio.org`, 2008.

[9] George Kiagiadakis and Julian Bouzas, "Wireplumber - session / policy manager implementation for pipewire," https://gitlab.freedesktop.org/pipewire/pipewire.

[10] Kristian Høgsberg, "Wayland," https://wayland.freedesktop.org/.

[11] The GStreamer Comunity, "Gstreamer api documentation," https://gstreamer.freedesktop.org/documentation/gstreamer/.

[12] Fons Adriaensen, "Zita-ajbridge," http://kokkinizita.linuxaudio.org/linuxaudio/zita-ajbridge-doc/quickguide.html, 2012.

[13] Freedesktop Comunity, "A portal frontend service for flatpak," https://github.com/flatpak/xdg-desktop-portal, 2016.

[14] Christian F.K. Schaller, "Pipewire late summer update 2020," https://blogs.gnome.org/uraeus/2020/09/04/pipewire-late-summer-update-2020/, 2020.

[15] PipeWire comunity, "Pipewire - gitlab freedesktop," https://gitlab.freedesktop.org/pipewire/pipewire.