Audio management

Contents

Apertis audio management was previously built around PulseAudio but with the move to the Flatpak-based application framework PipeWire[1] offers a better match for the use-cases below. Compared to PulseAudio, PipeWire natively supports containerized applications and keeps policy management separate from the core routing system, making it much easier to tailor for specific products.

Applications can use PipeWire through its native API[2], as the final layer to access sound features. This does not mean that applications have to deal directly with PipeWire: applications can still make use of their preferred sound APIs as intermediate layers for manipulating audio streams, with support being available for the PulseAudio API, for GStreamer or for the ALSA API.

In an analogous manner, applications can capture sound for various purposes. For instance, speech recognition or voice recorder applications may need to capture input from the microphone. The sound will be captured from PipeWire. ALSA users can use `pcm_pipewire`. GStreamer users can use `pipewiresrc`.

Terminology and concepts

See also the Apertis glossary[3] for background information on terminology.

Standalone setup

A standalone setup is an installation of Apertis which has full control of the audio driver. Apertis running in a virtual machine is an example of a standalone

[1] https://pipewire.org/
[2] https://pipewire.github.io/pipewire/
[3] https://martyn.pages.apertis.org/apertis-website/glossary/

setup.

Hybrid setup

A hybrid setup is an installation of Apertis in which the audio driver is not fully controlled by Apertis. An example of this is when Apertis is running under an hypervisor or using an external audio router component such as GENIVI audio manager[4]. In this case, the Apertis system can be referred to as Consumer Electronics domain (CE), and the other domain can be referred to as Automotive Domain (AD).

Different audio sources for each domain

Among others, a standalone Apertis system can generate the following sounds:

- Application sounds
- Bluetooth sounds, for example music streamed from a phone or voice call sent from a handsfree car kit
- Any kind of other event sounds, for example somebody using the SDK can generate event sounds using an appropriate command line

A hybrid Apertis system can generate the same sounds as a standalone system, plus some additional sounds not always visible to Apertis. For example, hardware sources further down the audio pipeline such as:

- FM Radio
- CD Player
- Driver assistance systems

In this case, some interfaces should be provided to interact with the additional sound sources.

Mixing, corking, ducking

Mixing is the action of playing simultaneously from several sound sources.

Corking is a request from the audio router to pause an application.

Ducking is the action of lowering the volume of a background source, while mixing it with a foreground source at normal volume.

Use cases

The following section lists examples of usages requiring audio management. It is not an exhaustive list, unlimited combinations exists. Discussion points will be highlighted at the end of some use cases.

[4] http://docs.projects.genivi.org/AudioManager/

Application developer

An application developer uses the SDK in a virtual machine to develop an application. He needs to play sounds. He may also need to record sounds or test their application on a reference platform. This is a typical standalone setup.

Car audio system

In a car, Apertis is running in a hypervisor sharing the processor with a real time operating system controlling the car operations. Apertis is only used for applications and web browsing. A sophisticated Hi-Fi system in installed under a seat and accessible via a network interface. This is a hybrid setup.

Different types of sources

Some systems classify application sound sources in categories. It's important to note that no standard exists for those categories.

Both standalone and hybrid systems can generate different sound categories.

Example 1

In one system of interest, sounds are classified as *main sources*, and *interrupt sources*. Main sources will generally represent long duration sound sources. The most common case are media players, but it could be sound sources emanating from web radio, or games. As a rule of thumb, the following can be used: when two main sources are playing at the same time, neither is intelligible. Those will often require an action from the user to start playing, should it be turn ignition on, press a play button on the steering wheel or the touchscreen. As a consequence, only policy mechanisms ensure that only one main source can be heard at a time.

Interrupt sources will generally represent short duration sound sources, they are emitted when an unsolicited event occurs. This could be when a message is received in any application or email service.

Example 2

In another system of interest, sounds are classified as *main sources*, *interrupt sources* and *chimes*. Unlike the first example, in this system, a source is considered a main source if it is an infinite or loopable source, which can only be interrupted by another main source such FM radio or CD player. Interrupt sources are informational sources such as navigation instructions, and chimes are unsolicited events of short duration. Each of these sound sources is not necessarily generated by an application. It could come from a system service instead.

Navigation instruction

While some music from FM Radio is playing, a new navigation instruction has to be given to the driver: the navigation instructions should be mixed with the music.

Traffic bulletin

Many audio sources can be paused. For example, a CD player can be paused, as can media files played from local storage (including USB mass storage), and some network media such as Spotify.

While some music from one of these sources is playing, a new traffic bulletin is issued: the music could be paused and the traffic bulletin should be heard. When it is finished, the music can continue from the point where the playback was paused.

By their nature, some sound sources cannot be paused. For example, FM or DAB radio cannot be paused.

While some music from a FM or DAB radio is playing, a new traffic bulletin is issued. Because the music cannot be paused, it should be silenced and the traffic bulletin should be heard. When it is finished, the music can be heard again.

Bluetooth can be used when playing a game or watching live TV. As with the radio use-case, Bluetooth cannot be paused.

USB drive

While some music from the radio is playing, a new USB drive is inserted. If setting *automatic playback from USB drive* is enabled, the Radio sound stops and the USB playback begins.

Rear sensor sound

While some music from the radio is playing, the driver selects rear gear, the rear sensor sound can be heard mixed with the music.

Blind spot sensor

While some music from Bluetooth is playing, a car passes through the driver's blind spot: a short notification sound can be mixed with the music.

Seat belt

While some music from the CD drive is playing, the passenger removes their seat belt: a short alarm sound can be heard mixed with the music.

Phone call

While some music from the CD drive is playing, a phone call is received: the music should be paused to hear the phone ringing and being able to answer the conversation. In this case, another possibility could be to notify the phone call using a ring sound, mixed in the music, and then pause the music only if the call is answered.

Resume music

If music playback has been interrupted by a phone call and the phone call has ended, music playback can be resumed.

VoIP

The driver wishes to use internet telephony/VoIP without noticing any difference due to being in a car.

Emergency call priority

While a phone call to emergency services is ongoing, an app-bundle process attempts to initiate lower-priority audio playback, for example playing music. The lower-priority audio must not be heard. The application receives the information that it cannot play.

Mute

The user can press a mute hard-key[5]. In this case, and according to OEM-specific rules, all sources of a specific category could be muted. For example, all *main* sources could be muted. The OEM might require that some sources are never muted even if the user pressed such a hard-key.

Audio recording

Some apps might want to initiate speech recognition. They need to capture input from a microphone.

Microphone mute

If the user presses a "mute microphone" button (sometimes referred to as a "secrecy" button) during a phone call, the sound coming from the microphone should be muted. If the user presses this button in an application during a video conference call, the sound coming from the microphone should be muted.

[5] https://martyn.pages.apertis.org/apertis-website/concepts/hardkeys/

Application crash

The Internet Radio application is playing music. It encounters a problem and crashes. The audio manager should know that the application no longer exists. In an hybrid use case, the other audio routers could be informed that the audio route is now free. It is then possible to fall back to a default source.

Web applications

Web applications should be able to play a stream or record a stream.

Control malicious application

An application should not be able to use an audio role for which it does not have permission. For example, a malicious application could try to simulate a phone call and deliver advertising.

Multiple roles

Some applications can receive both a standard media stream and traffic information.

External audio router

In order to decide priorities, an external audio router can be involved. In this case, Apertis would only be providing a subset of the possible audio streams, and an external audio router could take policy decisions, to which Apertis could only conform.

Non-use-cases

Automatic actions on streams

It is not the purpose of this document to discuss the action taken on a media when it is preempted by another media. Deciding whether to cork or silence a stream is a user interface decision. As such it is OEM dependent.

Streams' priorities

The audio management framework defined by this document is intended to provide mechanism, not policy: it does not impose a particular policy, but instead provides a mechanism by which OEMs can impose their chosen policies.

Multiple independent systems

Some luxury cars may have multiple IVI touchscreens and/or sound systems, sometimes referred to as multi-seat[6] (please note that this jargon term comes

[6] https://martyn.pages.apertis.org/apertis-website/concepts/multiuser/#multi-seat-logind-seats

from desktop computing, and one of these "seats" does not necessarily correspond to a space where a passenger could sit). We will assume that each of these "seats" is a separate container, virtual machine or physical device, running a distinct instance of the Apertis CE domain.

Requirements

Standalone operation

The audio manager must support standalone operation, in which it accesses audio hardware directly (Application developer).

Integrated operation

The audio manager must support integrated operation, in which it cannot access the audio hardware directly, but must instead send requests and audio streams to the hybrid system. (Different types of sources, External audio router).

Priority rules

It must be possible to implement OEM-specific priority rules, in which it is possible to consider one stream to be higher priority than another.

When a lower-priority stream is pre-empted by a higher-priority stream, it must be possible for the OEM-specific rules to choose between at least these actions:

- silence the lower-priority stream, with a notification to the application so that it can pause or otherwise minimise its resource use (corking)
- leave the lower-priority stream playing, possibly with reduced volume (ducking)
- terminate the lower-priority stream altogether

It must be possible for the audio manager to lose the ability to play audio (audio resource deallocation). In this situation, the audio manager must notify the application with a meaningful error.

When an application attempts to play audio and the audio manager is unable to allocate a necessary audio resource (for example because a higher-priority stream is already playing), the audio manager must inform the application using an appropriate error message. (Emergency call priority)

Multiple sound outputs

The audio manager should be able to route sounds to several sound outputs. (Different types of sources).

Remember preempted source

It should be possible for an audio source that was preempted to be remembered in order to resume it after interruption. This is not a necessity for all types of streams. Some OEM-specific code could select those streams based on their roles. (Traffic bulletin, Resume music)

Audio recording

App-bundles must be able to record audio if given appropriate permission. (Audio recording)

Latency

The telephony latency must be as low as possible. The user must be able to hold a conversation on the phone or in a VoIP application without noticing any form of latency. (VoIP)

Security

The audio manager should not trust applications for managing audio. If some faulty or malicious application tries to play or record an audio stream for which permission wasn't granted, the proposed audio management design should not allow it. (Application crash, Control malicious application)

Muting output streams

During the time an audio source is preempted, the audio framework must notify the application that is providing it, so that the application can make an attempt to reduce its resource usage. For example, a DAB radio application might stop decoding the received DAB data. (Mute, Traffic bulletin)

Muting input streams

The audio framework should be able to mute capture streams. During that time, the audio framework must notify the application that are using it, so that the application can update user interface and reduce its resource usage. (Microphone mute)

Control source activity

Audio management should be able to set each audio source to the playing, stopped or paused state based on priority. (Resume music)

Per stream priority

We might want to mix and send multiple streams from one application to the automotive domain. An application might want to send different types of alert. For instance, a new message notification may have higher priority than 'some contact published a new photo'. (Multiple roles)

GStreamer support

GStreamer should be supported.

Approach

PulseAudio embeds a default audio policy so, for instance, if you plug an headset on your laptop aux slot, it silences the laptop speakers. PipeWire has no embedded logic to do that, and relies on something else to control it, which suites the needs for Apertis better since it also targets special use-cases that don't really match the desktop ones, and this separation brings more flexibility.

WirePlumber[7] is a service that provides the policy logic for PipeWire. It's where the default policy like the one above is implemented, but unlike PulseAudio is explicitly designed to let people customize it. PipeWire and WirePlumber is what AGL has used to replace their previous audio manager in their latest Happy Halibut 8.0.0 release.

The overall approach is to adopt WirePlumber as the reference solution, but the separation between audio management and audio policy means that product teams can replace it with a completely different implementation with ease.

Streams metadata in applications

PipeWire provides the ability to attach metadata to a stream. The function `pw_fill_stream_properties()`[8] is used to attach metadata to a stream during creation. The current convention in usage is to use a metadata named `media.role`, which can be set to values describing the nature of the stream, such as `Movie`, `Music`, `Camera`, `Notification`, and other defined by PipeWire.

See also [GStreamer support].

Requesting permission to use audio in applications

Each audio role is associated with a permission. Before an application can start playback a stream, the audio manager will check whether it has the permission to do so. See [Identification of applications]. Application bundle metadata[9] describes how to manage the permissions requested by an application. The application can also use bundle metadata to store the default role used by all streams in the application if this is not specified at the stream level.

Audio routing principles

The request to open an audio route is emitted in two cases:

[7] https://gitlab.freedesktop.org/pipewire/wireplumber

[8] https://pipewire.github.io/pipewire/classpw___pipewire.html#a841dbb7608dc9cdda4a320d33fbbd39a

[9] https://martyn.pages.apertis.org/apertis-website/concepts/application-bundle-metadata/

- when a new stream is created
- before a stream changes state from Paused to Playing (uncork)

In both cases, before starting playback, the audio manager must check the priority against the business rules, or request the appropriate priority to the external audio router. If the authorization is not granted, the application should stop trying to request the stream and notify the user that an undesirable event occured.

If an application stops playback, the audio manager will be informed. It will in turn notify the external audio router of the new situation, or handle it according to business rules.

An application that has playback can be requested to pause by the audio manager, for example if a higher priority sound must be heard.

Applications can use the PipeWire event API to subscribe to events. In particular, applications can be notified about their mute status. If an event occurs, such as mute or unmute, the callback will be executed. For example, an application playing media from a source such as a CD or USB storage would typically respond to the mute event by pausing playback, so that it can later resume from the same place. An application playing a live source such as on-air FM radio cannot pause in a way that can later be resumed from the same place, but would typically respond to the mute event by ceasing to decode the source, so that it does not waste CPU cycles by decoding audio that the user will not hear.

Standalone routing module maps streams metadata to priority

An internal priority module can be written. This module would associate a priority to all differents streams' metadata. It is loaded statically from the config file. See Routing data structure example for an example of data structure.

Hybrid routing module maps stream metadata to external audio router calls

In the hybrid setup, the audio routing functions could be implemented in a separate module that maps audio events to automotive domain calls. However this module does not perform the priority checks. Those are executed in the automotive domain because they can involve a different feature set.

Identification of applications

Flatpak applications are wrapped in containers and are identified by an unique app-id which can be used by the policy manager. Such app-id is encoded in the name of the transient systemd scope wrapping each application instance[10] and can be retrieved easily.

[10] https://github.com/flatpak/flatpak/wiki/Sandbox#the-current-flatpak-sandbox

If AppArmor support is added to Flatpak, AppArmor profiles could also be used to securely identify applications.

Web application support

Web applications are just like any other application. However, the web engine JavaScript API does not provide a way to select the media role. All streams emanating from the same web application bundle would thus have the same role. Since each web application is running in its own process, AppArmor can be used to differentiate them. Web application support for corking depends on the underlying engine. WebKitGTK+ has the necessary support. See changeset 145811[11].

Implementation of priority within streams

The policy manager should be able to cork streams: when a new stream with a certain role is started, all other streams within a user defined list of roles will get corked.

Corking streams

Depending on the audio routing policy, audio streams might be “corked”, “ducked” or simply silenced (moved to a null sink).

As long as the role is properly defined, the application developer does not have to worry about what happens to the stream except corking. Corking is part of PipeWire API and can happen at any time. Corking *should* be supported by applications. It is even possible that a stream is corked before being started.

If an application is not able to cork itself, the audio manager should enforce corking by muting the stream as soon as possible. However, this has the side effect that the stream between the corking request and the effective corking in the application will be lost. A threshold delay can be implemented to give an application enough time to cork itself. The policy of the external audio manager must also be considered: if this audio manager has already closed the audio route when notifying the user, then the data will already be discarded. If the audio manager synchronously requests pause, then the application can take appropriate time to shutdown.

Ensuring a process does not overrides its priorities

Additionally to request a stream to cork, a stream could be muted so any data still being received would be silenced.

[11] https://trac.webkit.org/changeset/145811

GStreamer support

GStreamer support is straightforward. `pipewiresink` support the `stream-properties` parameter. This parameter can be used to specify the `media.role`. The GStreamer pipeline states already changes from `GST_STATE_PLAYING` to `GST_STATE_PAUSED` when corking is requested.

Remembering the previously playing stream

If a stream was playing and has been preempted, it may be desirable to switch back to this stream after the higher priority stream is terminated. To that effect, when a new stream start playing, a pointer to the stream that was currently playing (or an id) could be stored in a stack. The termination of a playing stream could restore playback of the previously suspended stream.

Using different sinks

A specific `media.role` metadata value should be associated to a priority and a target sink. This allows to implement requirements of a sink per stream category. For example, one sink for main streams and another sink for interrupt streams. The default behavior is to mix together all streams sent to the same sink.

Routing data structure example

The following table document routing data for defining a A-IVI inspired stream routing. This is an example, and in an OEM variant of Apertis it would be replaced with the business rules that would fulfill the OEM's requirements

App-bundle metadata defines whether the application is allowed to use this audio role, if not defined, the application is not allowed to use the role. From the role, priorities between stream could be defined as follows:

In a standalone setup:

role	priority	sink	action
music	0 (lowest)	main_sink	cork
phone	7 (highest)	main_sink	cork
ringtone	7 (highest)	alert_sink	mix
customringtone	7 (highest)	main_sink	cork
new_email	1	alert_sink	mix
traffic_info	6	alert_sink	mix
gps	5	main_sink	duck

In a hybrid setup, the priority would be expressed in a data understandable by the automotive domain. The action meaning would be only internal to CE domain. Since the CE domain do not know what is happening in the automotive domain.

role	priority	sink	action
music	MAIN_APP1	main_sink	cork
phone	MAIN_APP2	main_sink	cork
ringtone	MAIN_APP3	alert_sink	mix
customringtone	MAIN_APP3	main_sink	cork
new_email	ALERT1	alert_sink	mix
traffic_info	INFO1	alert_sink	mix
gps	INFO2	main_sink	mix

WirePlumber policy samples

All the policies in WirePlumber are completely scriptable and written in Lua. The Lua API Documentation can be found here[12].

The default roles, priorities and related actions are defined in `/etc/wireconfig/policy.lua.d/50-endpoints-config.lua` and can be re-written to support the standalone setup defined in Routing data structure example:

```
default_policy.policy.roles = {
  -- main sink
 ["Multimedia"]   = { ["priority"] = 0, ["action.default"] = "cork", ["alias"] = { "Movie", "Music", "Game" }, }
  ["GPS"]            = { ["priority"] = 5, ["action.default"] = "duck", },
 ["Phone"]       = { ["priority"] = 7, ["action.default"] = "cork", ["alias"] = { "CustomRingtone" }, },

  -- alert sink
  ["New_email"]      = { ["priority"] = 1, ["action.default"] = "mix", },
  ["Traffic_info"]   = { ["priority"] = 6, ["action.default"] = "mix", },
  ["Ringtone"]       = { ["priority"] = 7, ["action.default"] = "mix", },
}

default_policy.endpoints = {
 ["endpoint.multimedia"]   = { ["media.class"] = "Audio/Sink", ["role"] = "Multimedia", },
 ["endpoint.gps"]       = { ["media.class"] = "Audio/Sink", ["role"] = "GPS", },
 ["endpoint.phone"]      = { ["media.class"] = "Audio/Sink", ["role"] = "Phone", },
 ["endpoint.ringtone"]    = { ["media.class"] = "Audio/Sink", ["role"] = "Ringtone", },
 ["endpoint.new_email"]   = { ["media.class"] = "Audio/Sink", ["role"] = "New_email", },
 ["endpoint.traffic_info"]  = { ["media.class"] = "Audio/Sink", ["role"] = "Traffic_info", },
}
```

And, for example, a policy to automatically switch Bluetooth from A2DP to HSP/HFP profile when a specific application starts, e.g. Zoom, could be defined like:

[12] https://pipewire.pages.freedesktop.org/wireplumber/lua_api.html

```
#!/usr/bin/wpexec
--
-- WirePlumber
--
-- Copyright © 2021 Collabora Ltd.
--    @author George Kiagiadakis <george.kiagiadakis@collabora.com>
--
-- SPDX-License-Identifier: MIT
--
-- This is an example of a standalone policy making script. It can be executed
-- either on top of another instance of wireplumber or pipewire-media-session,
-- as a standalone executable, or it can be placed in WirePlumber's scripts
-- directory and loaded together with other scripts.
--
-- The script basically watches for a client application called
-- "ZOOM VoiceEngine", and when it appears (i.e. Zoom starts), it switches
-- the profile of all connected bluetooth devices to the "headset-head-unit"
-- (a.k.a HSP Headset Audio) profile. When Zoom exits, it switches again the
-- profile of all bluetooth devices to A2DP Sink.
--
-- The script can be customized further to look for other clients and/or
-- change the profile of a specific device, by customizing the constraints.
---------------------------------------------------------------------------
-

devices_om = ObjectManager {
  Interest { type = "device",
    Constraint { "device.api", "=", "bluez5" },
  }
}

clients_om = ObjectManager {
  Interest { type = "client",
    Constraint { "application.name", "=", "ZOOM VoiceEngine" },
  }
}

function set_profile(profile_name)
  for device in devices_om:iterate() do
    local index = nil
    local desc = nil

    for profile in device:iterate_params("EnumProfile") do
      local p = profile:parse()
      if p.properties.name == profile_name then
        index = p.properties.index
```

```
        desc = p.properties.description
        break
      end
    end

    if index then
      local pod = Pod.Object {
        "Spa:Pod:Object:Param:Profile", "Profile",
        index = index
      }

      print("Setting profile of '"
            .. device.properties["device.description"]
            .. "' to: " .. desc)
      device:set_params("Profile", pod)
    end
  end
end

clients_om:connect("object-added", function (om, client)
  print("Client '" .. client.properties["application.name"] .. "' connected")
  set_profile("headset-head-unit")
end)

clients_om:connect("object-removed", function (om, client)
 print("Client '" .. client.properties["application.name"] .. "' disconnected")
  set_profile("a2dp-sink")
end)

devices_om:activate()
clients_om:activate()
```

Testability

The key point to keep in mind for testing is that several applications can execute in parallel and use PipeWire APIs (and the library API) concurrently. The testing should try to replicate this. However testing possibilities are limited because the testing result depends on the audio policy.

Application developer testing

The application developer is requested to implement corking and error path. Testing those features will depend on the policy in use.

Having a way to identify the *lowest* and *highest* priority definition in the policy could be enough for the application developer. Starting a stream with the lowest priority would not succeed if a stream is already running. Starting a stream with the highest priority would cork all running streams.

The developer may benefit from the possibility to customize the running policy.

Testing the complete design

Testability of the complete design must be exercised from application level. It consist of emulating several applications each creating independent connections with different priorities, and verifying that the interactions are reliable. The policy module could be provisionned with a dedicated test policy for which the results are already known.

Requirements

This design fullfill the following requirements:

- [Standalone operation] and [Integrated operation] are provided using separate sets of configuration files.
- [Priority rules] are provided by the policy manager.
- the audio manager library interface is aware of [Multiple sound outputs].
- [Remember preempted source] can be implemented in the policy manager.
- [Audio recording] will use the same mechanisms.
- [Latency] is implemented by not adding additional audio processing layer.
- [Security] is implemented by relying on the Flatpak containerization, which could be further hardened by adding AppArmor support.
- [Muting output streams] and [Control source activity] uses PipeWire corking infrastructure.
- [Per stream priority] uses the PipeWire API.
- [GStreamer support] is provided indirectly thanks to existing plugins.

Open questions

Roles

- Do we need to define roles that the application developer can use?

  It's not possible to guarantee that an OEM's policies will not nullify an audio role that is included in Apertis. However, if we do not provide some roles, there is no hope of ever having an application designed for one system work gracefully on another.

- Should we define roles for input?

  Probably, yes, speech recognition input could have a higher priority than phone call input. (Imagine the use case where someone is taking a call, is not currently talking on the call, and wants to change their navigation destination: they press the speech recognition hard-key, tell the navigation system to change destination, then input switches back to the phone call.)

- Should we define one or several audio roles not requiring permission for use?

  No, it is explicitely recommended that every audio role requires permission. An app-store curator from the OEM could still give permission to every application requesting a role.

Policies

- How can we ensure matching between the policy and application defined roles?

  Each permission in the permission set should be matched with a media role. The number of different permissions should be kept to a minimum.

- Should applications start stream corked?

  It must be done on both the application side and the audio manager side. Applications cannot be trusted. As soon as a stream opens, the PipeWire process must cork it - before the first sample comes out. Otherwise a malicious application could play undesirable sounds or noises while the audio manager is still thinking about what to do with that stream. The audio manager might be making this decision asynchronously, by asking for permission from the automotive domain. The audio manager can choose uncork, leave corked or kill, according to its policies. On the application side, it is only possible to *suggest* the best way for an application to behave in order to obtain the best user experience.

- Should we use `media.role` or define an apertis specific stream property?

Summary of recommendations

- PipeWire is adopted as audio router and WirePlumber as policy manager.
- Applications keep using the PulseAudio API or higher level APIs like GStreamer to be compatible with the legacy system.
- The default WirePlumber policy is extended to address the use-cases described here.
- Static sets of configuration files can implement different policies depending on hybrid setup or standalone setup.
- Each OEM must derive from those policies to implement their business rules.
- WirePlumber must be modified to check that the application have the permission to use the requested role and, if the `media.role` is not provided in the stream, it must check if a default value is provided in the application bundle metadata.
- If AppArmor support is made available in Flatpak, WirePlumber must be modified to check for AppArmor identity of client applications.
- The application bundle metadata contains a default audio role for all streams within an application.

- The application bundle metadata must contain a permission request for each audio role in use in an application.
- For each stream, an application can choose an audio role and communicate it to PipeWire at stream creation.
- The policy manager monitors creation and state changes of streams.
- Depending on business rules, the policy manager can request an application to cork or mute.
- GStreamer's `pipewiresink` support a `stream.properties` parameter.
- A tool for corking a stream could be implemented.