Canterbury legacy application framework

Contents

The custom application framework based on the Canterbury app manager has been removed from `v2022dev2` in favor of upstream components like Flatpak, see the application framework[1] document for more details.

Flatpak and Canterbury cover the core tasks of an application framework:

- packaging
- distribution
- sandboxing

[1] https://em.pages.apertis.org/apertis-website/concepts/application-framework/

When Canterbury was designed Flatpak didn't exist and the available technologies were quite different from what is in today's usage, so it's now time to reconsider our approach.

Flatpak

- upstream, large community
- mature, proven on the field
- uses Linux containers to isolate the filesystem view from the application
- sandbox based on Linux containers and seccomp
- uses AppStream and .desktop files to encode metadata about the application
- backed by OSTree
- shared runtimes decouple libraries on the host from libraries depended by applications, changes on the host won't break applications
- deduplicates files across applications, runtimes and the host OSTree-based system
- SDK runtimes decouple development from the host
- growing IDE support (GNOME Builder, Eclipse)
- standardized D-Bus based portals for privileged operations
- transparent support for portals already available in the most widespread toolkits (Qt/GTK/etc.)
- large userbase
- available out-of-the-box on the most widespread distributions (Debian/Ubuntu/Fedora/Red Hat/Suse/etc.)
- well documented
- additional permissions are managed through high level entries in the application manifest
- sandboxed with seccomp
- mature OTA mechanism for applications
- user-facing app store available upstream
- the upstream app-store, FlatHub, can be deployed for Apertis, or the experimental Magento app-store could be adapted
- enables third-party applications (Sublime Text, Visual Studio Code, etc.) to be run on the SDK with no effort

Canterbury

- Apertis specific, no community
- not proven on the field
- pre-dates Linux containers availability, does not use them
- sandbox based on AppArmor
- uses AppStream and .desktop files to encode metadata about the application
- backed by OSTree
- applications use libraries from the host, no decoupling

- no concept of runtimes
- no deduplicaions
- limited IDE support (Eclipse)
- very sparsely documented
- security constraints expressed via low-level AppArmor profiles, no higher-level permission system
- no seccomp sandbox
- OTA mechanism for applications and agents at the prototype stage (Bosch-only, not available in Apertis)
- user-facing app store at the prototype stage (Bosch-only, not available in Apertis)
- there's an experimental Magento-based app-store, not currently available in Apertis

Comparison

Since Apertis is meant to adopt upstream solutions whenever possible it is natural for us to adopt Flatpak, but to do so the gaps that need to be filled must be evaluated.

The two systems are very different and for this reason no transparent compatibility can be provided, but thanks to the modular approach in Apertis Canterbury can be kept available in the repositories even if the reference setup will use Flatpak.

Since the two systems share many underlying technologies (D-Bus, OSTree, etc.) their performance are comparable. The additional use of control groups in Flatpak doesn't add any noticeable overhead. Flatpak consists of just an executable setting up the environment and does not require an always-running daemon as Canterbury does, so there may be a negligible memory saving.

Applications concept

The legacy Apertis application framework already defined the concept of application bundles. The new application framework defines the wanted format used for the bundle as being Flatpak.

Application layout

The application layout remains compatible with the legacy application framework, note that the layout is relative to the `/app/` folder inside of the Flatpak.

Application entry points

As the entry points[2] were defined using the standard specification from FreeDesktop.org, they remain compatible with the new Apertis application framework and are exposed by the flatpak executable to the system when necessary.

Desktop file should be updated to use Flatpak instead of Canterbury to launch the application, e.g. replacing

```
Exec=@bindir@/eye app-name @app_id@ play-mode stop url NULL
```

by

```
Exec=flatpak run app-name @app_id@ play-mode stop url NULL
```

Application metadata

The application metadata were specified using the AppStream FreeDesktop.org specification and remains the main metadata specification for Flatpak.

Bundle spec

The latest Canterbury application bundle specification has been largely based on the Flatpak one, in a initial effort to align Canterbury with recent upstream technologies:

- the binary format is the exactly same;
- in both cases AppStream is used for the bundle metadata;
- entrypoints are defined with `.desktop` files both in Canterbury and Flatpak;
- installation paths differ since Canterbury requires an unique installation path while Flatpak relies on containers to put different contents on the same path for each application, but from a practical point of view the difference is purely cosmetic.

Permissions

No high level support for application permission has been implemented in Canterbury, application access to resources was exclusively based on writing dedicated AppArmor profiles[3] for each applications and carefully reviewing them.

Flatpak instead lets application authors specify in the application manifest a set of special high-level permissions. The Flatpak approach has been analysed in more detail in the original permissions[4] document which already described the

[2] https://em.pages.apertis.org/apertis-website/architecture/bundle-spec/#entry-points

[3] https://em.pages.apertis.org/apertis-website/architecture/bundle-spec/#apparmor-profile

[4] https://em.pages.apertis.org/apertis-website/concepts/permissions/

use-cases for the permissions mechanism in the context of the Apertis application framework.

Preferences and persistence

The Apertis application framework satisfies the requirements of the legacy application framework. The only missing part is that application rollback is not able to revert the user-data to a previous state.

Containerisation

Canterbury pre-dates the maturity of containerization in Linux (cgroups and namespaces) and it does not make use of it.

Flatpak is instead heavily based on containers, providing much stronger isolation capabilities.

Large data sharing

The Apertis application framework allows to share data using the standard mechanisms as described by the FreeDesktop.org Desktop File specification. Any D-Bus enabled sharing service can be used when specifying the right interface in the Flatpak manifest. It is no more possible to register a service by putting a file into `/var/lib/apertis_extensions/applications` at installation time as the files are installed into a different path for each bundle.

Dialogs and notifications

The Apertis application framework is also using the Notification Specification[5] and allows to reuse the same interface without any breakage.

The dialog abstraction for the legacy application framework has never been implemented as its design is subject to many questions.

Launch applications and services

As Flatpak is well-integrated into existing environments and uses the same technology and protocols for its foundations, there is no expected problems with Flatpak here.

Launch pre-configured default apps at start-up (Launcher / Global popup / Status Bar)

The work has already beeing started as show by this upstream request[6] for this feature making it a small gap to fill.

[5] https://people.gnome.org/~mccann/docs/notification-spec/notification-spec-latest.html
[6] https://github.com/flatpak/flatpak/issues/118

AppArmor

Currently Apertis depends heavily on AppArmor to constrain services and applications: it is used to restrict filesystem access and mutually authenticate applications in a secure way when communicating over D-Bus.

AppArmor is currently used in Apertis for two different purposes:

- access constraints
- secure identification of D-Bus peers

While Flatpak has no support for AppArmor out of the box and adding it is not on the roadmap so far, the first use case is already covered by the use of Linux cgroups and namespaces which provide more flexibility than AppArmor. Flatpak also ships a D-Bus proxy to manage access policies at the D-Bus level, since that needs a finer control than cgroups and namespaces can provide.

The higher-level access constraints implemented by Flatpak are much easier and secure to be used by application authors than the low-level AppArmor policy language currently used by Apertis. In that sense, the adoption of Flatpak would be aligned to the plan to provide an higher-level access constraints mechanism to application authors and shield them from the AppArmor policy language.

Flatpak also includes the concept of "portals" to provide restricted access to resources to unprivileged applications, either by applying system-specific policies or by requiring user interaction. For instance, applications don't have access to user files, and file opening is handled via a privileged portal that ensure that applications can only access files users have given their consent to.

The second use of AppArmor is something very few applications at the moment use, and portals seem well suited to replace its known usages:

- Canterbury itself uses it to control applications: this is managed by Flatpak by using cgroups
- Newport (download manager) uses it to securely identify its clients: creating a dedicated Flatpak portal would address the use-case with no reliance on AppArmor
- Frome (magento app-store client) uses it to only let the `org.apertis.Mildenhall.Setting` system application talk to it: a dedicated Flatpak portal seem appropriate here as well
- Beckfoot (network management service) uses it to talk with `org.apertis.Mildenhall.StatusBar`, but Beckfoot itself has been declared obsolete long ago in {T3626} and the existing org.freedesktop.portal.Notification[7] could be used instead.

[7] https://flatpak.github.io/xdg-desktop-portal/portal-docs.html#gdbus-org.freedesktop.portal.Notification

Headless agents

Flatpak focuses on graphical application on the user session bus: nothing in its design prevents its usage for headless agents and some testing didn't show any significant issue, but some rough edges are expected.

Some one-time effort may be needed to consolidate this use-case in Flatpak.

System agents

Canterbury can only manage user-level applications and agents, and it doesn't currently have support for agents meant to be accessed on the system bus by different users.

Flatpak is not suited for system agents as well and focuses on the user session. Upstream explicitly considers system agents a non-usecase and working in this direction would produce a significant delta that would significantly impact the maintenance burden.

Flatpak apps run in an environment that can never exercise capabilities (`CAP_SYS_ADMIN`, `CAP_NET_ADMIN` etc.) or transition between uids, so some system services will not be possible to implement. System services that could run as an unprivileged system-level uid and don't do anything inherently privileged, like downloading files and putting them in a centralized location where all users can access them, should work. System services that need to be root to do inherently privileged things, like ConnMan/BlueZ, won't.

systemd "portable services", perhaps deployed using OSTree, might be a reasonable solution for system agents. They are very new and not yet considered stable, but are specifically meant for this purpose.

Multiple entry points

Canterbury supports multiple entry points in a single app-bundle, and Flatpak should support more than one desktop file which, as in Canterbury, are the implementation of entry points.

Application manager D-Bus interface

Canterbury exports an obsoleted D-Bus interface with a set of largely unrelated methods to:

- let application register themselves
- communicate to applications their new application state (show, hide, paused, off)
- hide global popups
- get the currently active application
- get the application that is currently using the audio source
- find out if the currently active application needs an Internet connection

Tracking the application that is currently "active" and hiding popups are tasks that should be handled by the compositor. The other interfaces are considered problematic as well.

Canterbury-core, the version of Canterbury for headless systems, already doesn't ship the application manager interface so there's no contingent need to reimplement it.

Audio management

The legacy application framework was built around PulseAudio.

Canterbury provides a custom audio manager which was already considered obsoleted and a different design[8] was proposed some time ago on top of PulseAudio.

With the need of more containment into the framework, the Apertis application framework is meant to use PipeWire as a replacement for PulseAudio. The intent for PipeWire is to be a drop-in replacement for PulseAudio during the transition period. PipeWire also provides a sink and source GStreamer element to replace their PulseAudio conterparts.

PipeWire is designed to let an external policy engine dictate how the audio should be routed and also provide proper security controls to restrict untrusted applications: for this reason AGL plans to use it as the foundation for their upcoming audio management solution, and Collabora is involved to ensure the embedded use-cases are covered.

An alternative which is largely in use is the GENIVI AudioManager, which can be used with Flatpak as well.

Canterbury-core, the version of Canterbury for headless systems, already doesn't ship the audio manager so there's no contingent need to reimplement it.

Hard Keys

Canterbury provides a D-Bus interface for handling hard-keys by communicating with the compositor over private interfaces. This is considered obsolete and hard-key handling should happen in the compositor directly.

Canterbury-core, the version of Canterbury for headless systems, already doesn't ship the hard key interface so there's no contingent need to reimplement it.

Preference application launching

Canterbury provides a D-Bus interface to let applications launch the preference manager to edit their preferences rather than providing their own interface.

[8] https://em.pages.apertis.org/apertis-website/concepts/audio-management/

This also requires support in the preference manager, which is not currently implemented.

Canterbury-core, the version of Canterbury for headless systems, already doesn't ship the preference launcher interface so there's no contingent need to reimplement it.

Out-of-memory handling

When memory pressure is detected Canterbury tries to kill applications not currently visible. The private API between Canterbury and the Mildenhall compositor and the implementation were already known to be problematic and were considered to be needing a significant rework in any case, possibly to move them to a dedicated module.

The module dedicated to the prioritization of applications in case of memory pressure can then be implemented to work with Flatpak applications seamlessy.

Bandwidth prioritization

Canterbury provides a experimental bandwidth prioritization system that is known to be problematic and has been considered obsoleted, see {T4043} for details. No similar mechanism is available in Flatpak.

App store

There's an experimental Magento-based app-store for Canterbury, but it is not yet available in Apertis. Flatpak has its own upstream app store, FlatHub, which is Open Source and can be self-hosted. It doesn't currently implement payments in any form. Possible options here are either publishing the Magento-based code and adapting it to work with Flatpak with a limited amount of changes but higher maintenance costs, or contribute on the implementation of payment methods on FlatHub, with an higher one-time cost but likely lower on-going maintenance requirements.

Manage launched application windows using the Window Manager

This was deprecated since Apertis 17.09. Canterbury uses private interfaces with the compositor to:

- show/hide splashscreens, but WM should be able to display splashscreens on its own without involving the application manager
- learn which application is being displayed to manage the "back" stack, but the WM is better positioned to handle the "back" stack on its own
- inform the WM that the Last User Mode is being set up, but it appears that the compositor takes no special action in that case

Notifies application whether they are in background or foreground

This is not part of canterbury-core and has been deprecated since Apertis 17.09. In a single fullscreen window scenario this can be handled by tracking whether the application has the focus or not. In the case multiple applications are visible at the same time, such as in the normal desktop case, the "background" status can be misleading since applications can still be partially visible. Wayland provides the frame clock to throttle the rendering of application windows which are not visible.

Maintain an application stack

Canterbury maintains a stack of applications to provide an Android-like back button. This feature should be implemented by the compositor to avoid layering violation. This is not part of canterbury-core as well and deprecated since Apertis 17.09.

Store Last User Mode (LUM) information periodically and restore LUM on start-up

This is not part of canterbury-core, and was deprecated since 17.09. Canterbury saves the currently running applications, "back" stack and the selected audio output in order to restore them on reboot.

The compositor should handle the saving and restoration of the application stack and the audio manager should save and restore the selected audio output without involving the application manager.

Conclusions

- No major gaps have been identified between Canterbury and Flatpak
- Flatpak has an very active upstream community and widespread adoption
- Most of the Canterbury APIs not related to app-management have been formally deprecated since Apertis 17.09
- Providing compatibility between the two would be a very big undertaking with unclear benefits, so it's actively discouraged and existing applications needs to be ported explicitly
- HMI applications will need to be reimplemented in any case as Mildenhall is not a viable solution for product teams
- The Canterbury application framework will remain available in Apertis as an option at least until the new application framework has matured enough and reference applications are available for it, and product teams will be able to choose one or the other depending on their specific needs