Applications

Contents

This document is intended to give a high-level overview of application handling by Apertis. Topics handled include the storage of applications and related data on the device, how they're integrated into the system, and how the system manages them at run-time. Topics related to the development of applications are covered by several other designs.

Unfortunately, the term "application"has seen a lot of misuse in recent times. While many mobile devices have an "application store"that distributes "application packages", what is actually in one of those packages may not fit any sensible definition of an application –as an example, on the Nokia N9 one can download a package from the application store that adds MSN Messenger capabilities to the existing chat application.

To avoid ambiguity, this document will avoid using "application"as a jargon term. Instead, we use two distinct terms for separate concepts that could informally be referred to as applications: *graphical programs*, and *application bundles*. See Terminology.

Apertis is a multiuser system; see themultiuser[1] design document for more on the specifics of the multiuser experience and the division of responsibilities between middleware and HMI elements.

Traditional package managers are unfit for applications

Apertis relies heavily on a traditional packaging system to compose the base OS. However, it does not rely on it to distribute the composed system as it is not a good fit for the use-cases Apertis addresses, seesystem updates and rollback[2] for more details. Similarly, a traditional packaging system is not a good fit for applications in Apertis since:

- Apertis relies on a immutable base OS to implement a robust update mechanism, seesystem updates and rollback[3] for more details. This means that a traditional package manager is not used to distribute updates on the field and that the writable application storage should be kept separate from the read-only base OS.
- Application bundles don't depend on each other –this simplifies dependency management in modern package management systems specializing in support for applications.
- Much of the complexity in application bundle handling (DRM, rollbacks, communicating security "permissions"to the user) is not part of traditional package management tools, and is not interesting to the upstream tool maintainers.
- Applications can have conflicting dependencies which can't be shipped as part of the base OS and should be somehow bundled with the application itself.

Thus, Apertis has chosen Flatpak[4] as the system for installing and managing application bundles (see application framework[5]).

Terminology

Graphical program

A ***graphical program*** is a program with its own UI drawing surface, managed by the system's window manager. This matches the sense with which "application"is traditionally used on desktop/laptop operating systems, for instance referring to Notepad or to Microsoft Word.

[1] https://www.apertis.org/concepts/multiuser/
[2] https://www.apertis.org/concepts/system-updates-and-rollback/
[3] https://www.apertis.org/concepts/system-updates-and-rollback/
[4] https://flatpak.org/
[5] https://www.apertis.org/concepts/application-framework/

Bundle

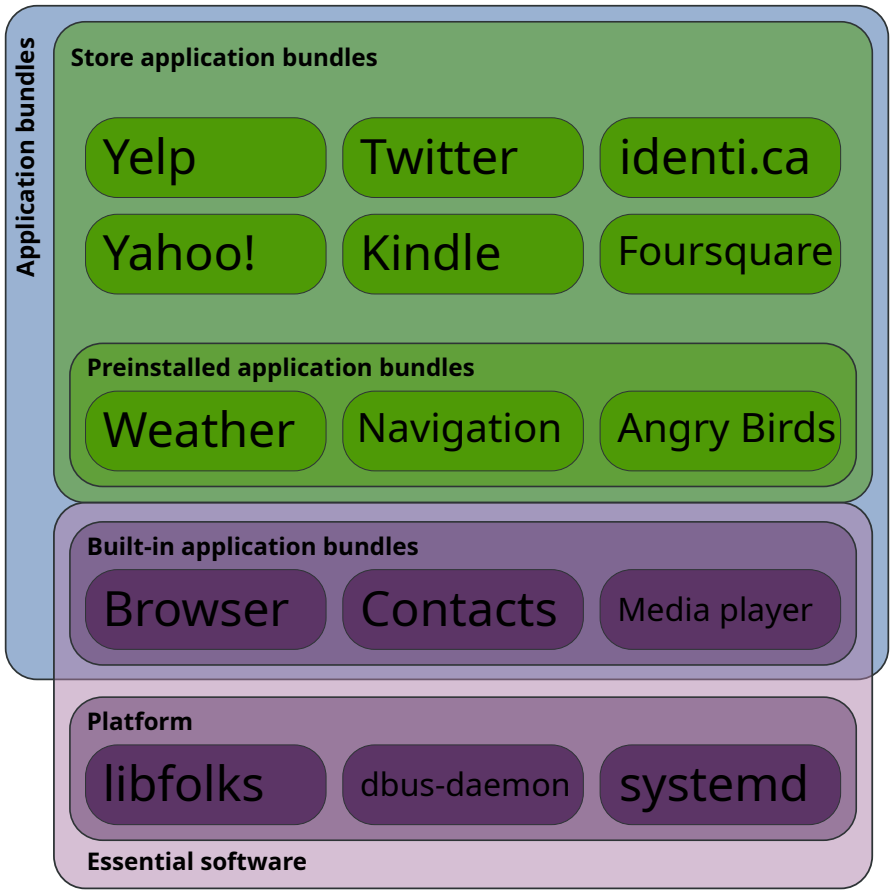
A ***bundle*** or ***application bundle*** is a group of functionally related components (be they services, data, or programs), installed as a unit. This matches the sense with which "app"is typically used on mobile platforms such as Android and iOS; for example, we would say that an Android .apk file contains a bundle. Some systems refer to this concept as a ***package***, but that term is strongly associated with dpkg/apt (.deb) packages in Debian-derived systems, so we have avoided that term in this document.

Store account

The Digital rights management section discusses ***store accounts***, anticipated to have a role analogous to Google Play accounts on Android or Apple Store accounts on iOS. If these accounts exist, we recommend against using the term "user"for them, since that would be easily confused with the users found in the Multiuser design document; it is not necessarily true that every user has access to a store account, or that every store account corresponds to only one user.

Software Categories

The software in a Apertis device can be divided into three categories: *platform*, *built-in application bundles* and *store application bundles*. Of these categories, some store application bundles may be *preinstalled*.



The ***platform*** is comprised of all the facilities used to boot up the device and perform basic system checks and restorations. It also includes the infrastructural services on which the applications rely, such as the session manager, window manager, message bus and configuration storage service, and the software libraries shared between components.

Built-in application bundles are components that have a structure analogous to that of an application bundle from the application store, but can only be upgraded as part of an operating system upgrade, not separately. This should include all software laid on top of the platform that is on the critical path of user-facing basic functionality, and hence cannot be removed or upgraded except by installing a new operating system; this might include basic software such as the browser, email reader and various settings management applications.

The platform and built-in applications combine to make up ***essential software***: the bare minimum Apertis will always have installed. Essential software has strict requirements both in terms of reliability and security.

Store application bundles are application bundles developed by third-parties to be used as add-ons to the system: they are not part of the system image and are made available for installation through the application store instead. While they may be important to the user, their presence is not required to operate the device properly.

It is important to note that store application bundles can be shipped pre-installed on the device, which provides OEMs with a flexible way of providing differentiation or a more complete user experience by default.

Pre-installed Applications

On most software platforms there are two kinds of applications that come pre-installed on the device: what we call built-in application bundles and regular store application bundles. The difference between built-in application bundles and regular store application bundles that just happen to come pre-installed is essentially that the former are considered part of the system's basic functionality, are updated along with the system and cannot be removed.

Taking Apple's iPad as an example, we can see that approach being applied: Safari, Weather, Mail, Camera and so on are built into the system.

> See http://www.apple.com/ipad/built-in-apps/ for a list

They cannot be removed and they are updated through system updates. Apple doesn't seem to include any store applications pre-installed, though.

The Android approach is very similar: applications such as the browser are not removable and are updated with the system, but it's much more common to have store applications be pre-installed, including Google applications such as Gmail, Google Maps, and so on.

The reason why browsers, mail readers, contacts applications are built-in software that come with the system is they are considered integral parts of the core user experience. If one of these applications were to be removed the user would not be able to utilize the device at all or would have a lot of trouble doing so: listening to music, browsing the web and reading email are basic expectations for any mobile consumer device.

A second reason which is also important is that these applications often provide basic services for other applications to call upon. The classic example here is the contacts application that manages contacts used by text messaging, instant Internet messaging, email, and several other use cases.

Case Study: a navigation application, how would it work? The navigation application was singled out as a case that has requirements and features that intersect those of built-in applications and those of store applications. On the one hand, the navigation application is core functionality, which means it should be part of the system. On the other hand, it should be possible to make

the application extensible or upgradable, enabling the selling of updated maps, for instance.

Apertis believes that the best way to solve this duality is to separate program and data, and to follow the lead of other platforms and their app stores in providing support for in-app purchases. This functionality is used often by games to provide additional characters, scenarios, weapons and such, but also used by applications to provide content for consumption through the application, such as magazine issues and also maps.

For such a feature to work, it needs to be provided as an API that applications can use to talk to the app store to place orders and to verify which data sets the user should be allowed to download. The actual data should be hosted at the app store for downloading post-validation. The disposition of the data, such as whether it should be made available as a single file or several, whether the file or files are compressed or not, should be left for the application author to decide on based on what makes more sense for the application.

Responsibilities of the Application Store

The application store will be responsible for hosting a developer's signed application bundles. Special "SDK"system images will provide software development tools and allow the installation of unsigned packages, but the normal "target" system image will not allow the installation of packages that don't contain a valid store signature.

The owner of the store, via the signing authority of the application store, will have the ability to accept or reject any application to be run on Apertis. By disallowing any form of "self publication"by application developers, the store owner can ensure a consistent look and feel across all applications, screen applications for malicious behavior, and enforce rigorous quality standards.

However, pre-publication screening of applications will represent a significant time commitment, as even minor changes to applications must undergo thorough testing. High priority security fixes from developers may need to be given a higher priority for review and publication, and the priority of application updates may need to be considered individually. System updates will correspond to the busiest periods for both internal and external developers, and the application store will experience significant pressure at these times.

Identifying applications

During the design of other Apertis components, it has become clear that several areas of the system design would benefit from a consistent way to identify and label application bundles and programs. In particular, the ability to provide a security boundary where inter-process communication is used relies on being able to identify the peer, in a way that ensures it cannot be impersonated.

An application has several strings that might reasonably act as its machine-readable name in the system:

- the name of the application bundle, being the Flatpak `app-id`[6] or the name discussed in Application bundle metadata[7]
- the D-Bus well-known name or names taken by the program(s) in the bundle, for instance via GLib's GApplication interface
- the name(s) of the freedesktop.org .desktop file(s) associated with the program(s), if they have them
- the name of the systemd user service (.service file) associated with the program(s), if they have them

Flatpak aligns these according to the following system[8]:

- The *bundle ID* is a case-sensitive string matching the syntactic rules for a D-Bus interface name, i.e. two or more components separated by dots, with each component being a traditional C identifier (one or more ASCII letters, digits, or underscores, starting with a non-digit).

  This scheme makes every bundle ID a valid D-Bus well-known name, but excludes certain D-Bus well-known names (those containing the hyphen/minus). This allows hyphen/minus to be used in filenames without ambiguity, and facilitates the common convention in which a D-Bus service's main interface has the same name as its well-known name.
- Application authors should be strongly encouraged to use a DNS name that they control, with its components reversed (and adjusted to follow the syntactic rules if necessary), as the initial components of the bundle ID. For instance, the owners of collabora.com and 7-zip.org might choose to publish `com.collabora.MyUtility` and `org._7_zip.Decompressor`, respectively. This convention originated in the Java world and is also used for Android application packages, Tizen applications, D-Bus names, GNOME applications and so on.
- App-store curators should not allow the publication of a bundle whose name is a prefix of a bundle by a different developer, or a bundle that is in the essential software set. App-store curators do not necessarily need to verify domain name ownership in advance, but if a dispute arises, the app-store curator should resolve it in favour of the owner of the relevant domain name.
- Well-known namespaces used by platform components (such as `apertis.org`, `freedesktop.org`, `gnome.org`, `gtk.org`) should be restricted to app bundles associated with the relevant projects. Example projects provided

[6] http://docs.flatpak.org/en/latest/using-flatpak.html#identifiers
[7] https://www.apertis.org/concepts/application-bundle-metadata/
[8] http://docs.flatpak.org/en/latest/using-flatpak.html#identifiers

in SDK documentation should use the names that are reserved for examples (see RFC2606[9]), such as example.com, but app-store curators should not publish bundles that use such names.

- Programs in a bundle may use the D-Bus well-known name corresponding to the bundle ID, or any D-Bus well-known name for which the bundle ID is a prefix. For instance, the `org.apertis.MyUtility` bundle could include programs that take the bus names `org.apertis.MyUtility`, `org.apertis.MyUtility.UI` and/or `org.apertis.MyUtility.Agent`.
- If a program has a freedesktop.org .desktop file, its name should be the program's D-Bus well-known name followed by `.desktop`, for example `org.apertis.MyUtility.UI.desktop`.

A library available to platform services should provide a recommended implementation of this algorithm.

Application Releasing Process

Once application testing is complete and an application is ready to be distributed, the application releasing process should contain at least the following steps:

- Verify that the application's bundle ID does not collide with any bundle by a different publisher (in the sense that neither is a prefix of the other).
- Make the application available at the store.

Application Installation Tracking

The System Updates and Rollback design describes a method of migrating settings and data from an existing Apertis system to another one. To work properly, the application store would need to have a list of applications installed on a specific Apertis device.

If the application store keeps a database of vehicle IDs and the applications purchased for them, this will help in order to facilitate software updates and to simplify software re-installation after a system wipe.

The application store can only know which applications have been downloaded for use in a specific vehicle –with no guarantee of a persistent Internet connection, the store has no way to know whether the application has really been installed or subsequently uninstalled. The store also can't reliably track what version of an application is installed.

If an application is downloaded on a computer with a web browser (presumably for installation via external media), the store shouldn't assume it was actually installed anywhere. Only applications installed directly to the device should be logged as installed. When the user logs in to the store (or the device logs into

[9] http://www.rfc-editor.org/info/rfc2606

the store with the users credentials to check for updates), the list of installed packages can be synchronized.

If an application is installed from a USB storage device the application manager could write a synchronization file back to the device that could subsequently be uploaded back to the application store from a web browser. Care should be taken to ensure these files can't be used by malicious users to steal applications –the store should check that the applications listed in the synchronization file have been legitimately purchased by the user and the file's contents should be discarded if they have not.

To perform a migration for a device that hasn't had a consistent Internet connection, the device could be logged into the store to synchronize its application list prior to beginning the migration process.

Digital Rights Management

Details of how DRM is to be used in Apertis are not finalized yet, but some options are presented here.

The store is in a convenient position to enforce access control methods for applications. When an application is purchased, the application store can generate the downloadable bundle with installation criteria built in.

The installation could be locked in the following ways:

- Locked to a specific device ID –it will only install on a specific Apertis unit.
- Locked to a specific ID of a larger system containing the device running Apertis. For instance, in the automotive use case, this could be locked to a specific vehicle ID. The Apertis unit will refuse to install the application if the vehicle ID does not match the ID embedded in the downloaded application package.
- Locked to a customer ID –It will only install for a specific person, as represented by their store account - presumably a store account must be present and logged in for this to work. The store account is assumed to be analogous to an Apple Store or Google Play account: as noted in Terminology, we recommend avoiding the term "user"here, since a store account does not necessarily correspond 1:1 to the "users"discussed in the Multiuser design document.

Any "and"combination of these 3 locks could also be used. For example, an application bundle may only be installable to a specific device in a specific vehicle (in other words, locked to vehicle ID and device ID) –if the Apertis unit is placed in another vehicle, or the vehicle's Apertis unit is replaced, the application bundle would not be installable.

Conversely, rights could also be combined with the "or"operator, such as allowing an application bundle to be installed if either the correct Apertis unit is used, or the correct vehicle. Collabora recommends these combinations not be implemented. Most of the combinations provided by "or"aren't obviously useful.

It might also be useful to distribute some packages in an unlocked form –free software, ad sponsored software, or demo software may not require any locking at all. Ultimately, this is a policy decision, not a technical one, as they could just as easily be locked to the downloader's account.

Note that these are all install time checks, and if a device is moved to another vehicle after successfully installing a bundle, it may result in running an app somewhere that an application developer or OEM didn't intend it to be run. In order to prevent this from happening, it would be more reliable to do launch-time testing of the applications.

The store would generate a file to be bundled with the application that listed the launch criteria, and the application manager would check those criteria before launching the application for use.

It should be considered that launch time testing would require a user to be logged in to the store in some way if the applications are to be keyed to a store account. This would make it impossible to launch certain applications when Apertis is without network connectivity, and could be a source of frustration for end users.

Permissions

Applications can perform many functions on a variety of user data. They may access interfaces that read data (such as contacts, network state, or the users location), write data, or perform actions that can cost the user money (like sending SMS). As an example, the Android operating system has a comprehensive manifest[10] that govern access to a wide array of functionality.

Some users may wish to have fine grained control over which applications have access to specific device capabilities, and even those that don't should likely be informed when an application has access to their data and services.

See the Permissions concept design[11] for further details.

Ideally, users would be able to accept a subset of permission, but there are some difficulties in allowing this:

- A huge testing burden is placed on the application developer if they can' t rely on the requested permissions. They must test their applications in all possible configurations.

[10] http://developer.android.com/reference/android/Manifest.permission.html
[11] https://www.apertis.org/concepts/permissions/

- The permissions may be required for the application developer's business model –be that network permissions for displaying advertising, or GPS information for crowd sourcing traffic information. Allowing the user to restrict permissions in these situations would be unfair to the developer.

To mitigate some of these problems, an application author may assume that the permissions listed in the Flatpak manifests[12] will be available. More sensitive permissions are available via runtime requests to the external XDG portals[13] running on the host system; the user may reject access to these permissions at their discretion.

Some permissions may prove to be more of an annoyance than helpful to the user. It may be worth considering having some permission acceptance governed by system settings, and only directly query the user if a permission is "important" (such as sending SMS).

Data Storage

Applications will have access to several types of writable application storage, as per the official Flatpak guidance on XDG base directories[14]. In addition, an application can request permission to access general storage locations such as the user's home or documents folder.

Extending Storage Capabilities

It may be desirable for some Apertis devices to allow the user to install an SD card to increase storage capacity. Since SD cards are removable –possibly even at runtime –they present some problems that need to be addressed:

- Allowing applications to be run from SD cards makes it more difficult to prevent software piracy.
- An SD card should be properly unmounted by the system before being physically removed from the device.

It is recommended that SD card storage not be used for the installation of applications or any manner of system software, as this could give users a way to run untrusted code, or tamper with application settings or data in ways the developers haven't anticipated. Media files are obvious candidates for placement on this type of removable storage, as they don't provide key system functionality, and are not trusted data.

If it is critical that applications (or other trusted data, such as navigation maps) be run off of removable storage, allowing the system to "format"the device before use, deleting all data already on the card and replacing it with an encrypted

[12] https://docs.flatpak.org/en/latest/manifests.html
[13] https://flatpak.github.io/xdg-desktop-portal/portal-docs.html
[14] https://docs.flatpak.org/en/latest/conventions.html#xdg-base-directories

BTRFS filesystem would allow a secure method of placing application storage on the device.

The dm-crypt LUKS[15] system would be used to encrypt the storage device using a random binary key file. These key files would be generated at the time the external storage device is formatted and stored along with the device serial number. One way to generate a key file would be to read an appropriate number of bytes (such as 32) from `/dev/random`.

The key store will be in a directory in the var subvolume (but not in `/var/lib`) as the var subvolume is not tracked by system rollbacks. If the key files were in a volume subject to rollbacks, they would disappear and render external storage unreadable after a system rollback that crossed their creation date.

It is imperative that the key store not be accessible to a user as it would allow them to directly access their removable storage device on another computer and potentially copy and distribute applications.

The device could be recognized by its label as reported by the blkid command, and added to the startup application scan in Boot time procedures.

If this is extended to multiple SD cards, difficulties arise in deciding which storage device to install an application to. Either configuration options will need to be added to control this, or the device with the greatest free space at the time of installation can be selected.

Many embedded devices require some manner of disassembly to remove an SD card, preventing the user from removing it while the system is in operation (such as a mobile phone that hides the SD card behind the battery). If an approach such as this is used, there is no need for special "eject"procedures for the SD storage. If this is not possible however, some manner of interface will need to be provided so the user can safely unmount the SD card before removal.

If it's physically possible for a user to remove the SD card while the system is running, the operating system and applications may be exposed to difficult to recover from situations and poorly tested code paths. These sorts of SD card sockets should probably not be used for cards using the BTRFS filesystem. Instead, the better tested FAT-32 filesystem should be used.

Application Management

Applications will be distributed by the application store as "application bundles" containing programs and services that can be launched in a variety of ways.

> All communication with the application store will take place over a secure HTTPS connection.

[15] https://gitlab.com/cryptsetup/cryptsetup

The metadata in this bundle provides information about the application such as it's user friendly name, services it needs from the system (such as querying the GPS) and the permissions it needs from the user.

Store Applications

Acquisition Applications will be made available through the application store's corresponding Flatpak repository[16].

Since Apertis may have limited or no Internet connectivity, it must be possible to download an application elsewhere and install it from a USB storage device. Even if Internet connectivity is available the download process must be reliable –it must be possible to resume a partially completed application download if the connection is broken or Apertis is shut down before the download completes.

Installation If an application is being installed directly from the store, an icon will be displayed in the launcher while the download and installation takes place will now be acquired from the application store.

Displaying an accurate progress indicator while installing an application is non-trivial. One simple option is to include the full decompressed size of the application in its metadata and send an update to the user interface occasionally based on the amount of bytes written.

This assumes that "number of bytes left to install"directly correlates to "amount of time left to completion", and suffers from a couple of common problems:

- Eventually storage caches are filled and begin writing out causing a dramatic slowdown in apparent installation speed for larger applications.
- Decompression speed may vary for different OSTree objects that are part of the same application.
- Some of the required files to be downloaded may already be available locally, but this will not be known until the file's digest is determined.

However, users are unlikely to notice even moderate inaccuracies in an installation percentage indicator, so this may be adequate without requiring complicated development that may not solve these problems anyway.

Upgrades If configured with a suitable Internet connection, the system will periodically check whether upgrades are available for any store applications that have been installed. Apertis will provide its vehicle ID to the application store and the application store will reply with a list of the most recent versions of the applications authorized for the vehicle. If Apertis has had software installed or removed without an Internet connection, the list of installed applications will be synchronized with the store at this time.

[16] https://docs.flatpak.org/en/latest/repositories.html

Some users may voice concerns over the store's tracking of all the installed packages on their Apertis. It may be worth mentioning in a "privacy policy" exactly what the data will be used for.

If no Internet connection is available, the user can still supply a newer version of an application on a USB device to start an upgrade. They can acquire application bundles from the store web page, which will provide the latest version of applications for download. Old application versions will not be available through the store.

Since the application store attempts to track installed applications, it could notify a user by e-mail when updates are available, or show a list of updated application when the user logs in to the store.

Removal When a user removes the application, any personal settings and caches required by the application will be automatically removed –files the application has stored in general storage will be left behind.

Removing a third-party music player shouldn't delete the user's music collection, but it should delete any configuration information specific to that player. For this to work properly, application developers need to be careful to store data in the appropriate locations.

Roll-back Apertis may utilize Flatpak functionality[17] to implement a per-application rollback system that allows an end user to revert to the last installed version of an application (that is, a single previously installed version will be kept when an upgrade is performed).

This rollback paradigm has some interesting quirks:

- If a user rolls back an application installed system-wide, all other users of that application will also be rolled back.
- As some software updates may contain critical security fixes, an ever growing blacklist will have to be maintained to prevent a user from rolling back to potentially dangerous versions.
- A rollback will not modify the application's data. Thus, if the data has been modified by the newer version, it would then be up to the application to determine how to handle the changes when loading the data in the previous version.
- Developers will have no control over what software versions their customers are using, making long term support very difficult. They may receive bug reports for bugs already fixed in newer versions of the software.

[17] https://docs.flatpak.org/en/latest/tips-and-tricks.html#downgrading

- Old versions of applications may break if they interact with online services that changed their protocols, or if Apertis APIs are deprecated.
- The effect of a system rollback on installed applications is unclear. If an application has been upgraded twice since the last system update and a full system rollback occurs it is possible for applications to have no launchable version installed.
- In some cases an application rollback may not even be possible if the old version of the application is not capable of running on the current version of the system.

After application rollback, launching the application now will use the previously installed version, with all user data left untouched, in identical state as before the rollback.

License Agreements

> Collabora does not have legal expertise in these matters, and any authoritative information –especially if financial damages may be involved –should be supplied by the appropriate legal advisers.

Each application may have its own license agreements, privacy policies, or other stipulations a user must accept before they can use the application. Different OEMs may have different requirements, and the legal requirements governing the contents of these documents may vary from country to country.

Such licenses generally disclose information regarding the use of data collected by an application or related services, define acceptable usage of the application or services by a user, or discuss the warranty and culpability of the application provider.

Regardless of content, Apertis should make all reasonable efforts to ensure a user has agreed to the appropriate agreements before they may use an application. The first step to accomplishing this goal is to require a user accept the license agreement before downloading an application from the store.

As this only requires a single user to accept the agreement, and does nothing for built-in applications, it is an incomplete solution. Requiring acceptance of the license terms when an application is installed, or when it is enabled for a user's account, would increase the likelihood that a user has agreed to the appropriate license.

If license terms change between releases, it might be advisable to ask users to accept the license terms on the first launch after an application update or rollback as well.

Ultimately, there is no guarantee that the person using a Apertis account is the person that agreed to an application's license.

Some licenses, such as the GPL, inform the user of their rights to obtain a copy of the source code of the software. Licenses like this should be made available to the user, but don't necessarily need to be displayed to the user unless the user explicitly requests the information.

Application Run Time Life-Cycle

The middleware will assist UI components in launching and managing applications on Apertis. Application bundles can provide executable files (programs) to be launched via different mechanisms, some of them user visible (perhaps as icons on the desktop or home screen that will launch a graphical program), and some of them implicit (like a connection manager for the Telepathy framework, or a graphical program that does not appear in menus but is launched in order to handle a particular request).

On a traditional Linux desktop, a graphical program doesn't generally make a distinction between foreground and background operation, though it may watch for certain events (input focus, window occlusion) that could be used to monitor that status. Some mobile operating systems (Android, iOS) hide the details of background operation from the user, some (WebOS, Meego) allow the user to interact with background applications more directly.

The approach will be similar to that traditional desktop Linux; an application is fully closed upon restart or when requested, and resuming state is entirely within its own responsibility.

Start

There are multiple ways in which a program associated with an application bundle, whether graphical or not, can be started by Apertis:

- Direct launch –application bundles may contain an image to be displayed in the application launcher, which will launch a suitable graphical program. The name and icon shown in the application launcher is part of the entry point metadata[18].
- By data type association - The content-type (MIME type) of data is used to select the appropriate application to handle the request. Applications will provide a list of content-types (if any) that they handle in the entry point metadata[19]; activating the application with the corresponding content type will launch the corresponding graphical program.
- An application can request the ability to launch persistent non GUI processes that provide a background component for applications. These can also be launched automatically at boot time. The ability for an application to request this is conditional upon the user's choice, and the application

[18] https://www.apertis.org/concepts/application-entry-points/
[19] https://www.apertis.org/concepts/application-entry-points/

is responsible the situation that would arise if background or auto-start support were rejected.

We refer to the programs that are launched in these ways as *entry points*.

Another method of launching processes is present –D-Bus activation. If a D-Bus client attempts to use a known-name for a service that isn't currently running, D-Bus will search its configuration files for an appropriate handler to launch. This sort of activation is more useful for system level developers, and won't be used to launch graphical programs.

During pre-publication review by the app store, careful attention should be paid to application bundles that wish to use background services, and the resource consumption of the services. The concept does not scale –it creates a system where the number of installed application bundles can dramatically affect run-time performance as well as system boot-up time.

Background Operation

More than one graphical program may be running at the same time, but the user can only directly interact with a limited number of graphical programs at any instant. For example, 1/3 of the screen may be giving driving directions while the other 2/3 of the screen displays an e-mail application. Concurrently, in the background, a music player may be running while several RSS feed readers are periodically updating.

Background tasks may also be performed by long running background processes. These run for the duration of the user's session, and are never explicily terminated unless the user revokes the background permissions or the system is low on resources.

Graphical programs will be notified by the compositor when they lose focus and are relegated to background status –the response to this notification is application dependent. If it has no need to perform processing in the background, It may save its current state and self-terminate, or it may remain idle until re-focused. Some graphical programs will continue to operate in the background –for example, a navigation application might remain active in the background and continue to give turn-by-turn instructions.

Graphical programs that need to perform tasks in the background will have to request for permissions[20]. Ideally they should be designed with a split between foreground and background components (perhaps using a graphical program for the user interface and a background service for such work) instead.

If a background graphical program wishes to be focused, it can use the standard method for requesting that a window be brought to the foreground.

[20] https://www.apertis.org/concepts/permissions/

End

Applications written for Apertis have persistent state, so from a user's perspective they never end. Apertis still needs to be able to terminate applications to manage resources, perform user switching, or prepare for shutdown.

Programs –either graphical or not –may be sent a signal by the middleware at any time requesting that they save their state and exit. Even if the application bundle has permission to run in the background, its processes may still be signaled to save its state in the case of a system shut-down or a user switch.

To prevent an application that doesn't respond to the state saving request from delaying a system shutdown or interfering with the system's ability to manage memory, processes will be given a limited amount of time (5 seconds) to save their state before termination. Applications that don't need to save state should simply exit in response to this signal.

It should be noted that state saving is difficult to implement, and much of the work is the responsibility of the application writer. While Apertis can provide functions for handling the incoming signal and storing state data, the hardest part is determining exactly what application state needs to be saved in order for the application to exit and restart in exactly the same way it had been previously running.

There is no standard Linux API for saving application state. POSIX defines `SIGSTOP` and `SIGCONT` signals for pausing and resuming programs, but these signals don't remove applications from memory or provide any sort of persistence over a system restart. Since they're unblockable by applications, the application may be interrupted at any time with no opportunity to do any sort of clean-up.

However, some applications may react to changes in system state –such as network connectivity. One method of preventing applications from reacting to D-Bus messages, system state changes, and other signaling is to use `SIGSTOP` to halt an application's processing. The application becomes responsible for handling whatever arises after `SIGCONT` causes it to resume processing (such as a flood of events or network timeouts).

Automatically saving the complete state of an application is essentially impossible - even if the entire memory contents are saved, the application may have open files, or open connections on remote servers, or it may have configured hardware like the GPU or a Bluetooth device.

For a web browser the state might be as simple as a URL and display position within the page, and the page will be reloaded and redisplayed when the browser is re-launched. However, if the user was in the middle of watching a streaming video from a service that requires a log-in, the amount of information that needs to be retained is larger and has potential security ramifications.

It's possible that a viewer application may exit and the file it was viewing be deleted before the application's next start, making it impossible to completely

restore the previous application state. Applications will be responsible for handling such situations gracefully.

Resource Usage

To make better use of the available memory, it's recommended that applications listen to the cgroup notification memory.usage_in_bytes[21] and when it gets close to the limit for applications, start reducing the size of any caches they hold in main memory. It may be good to do this inside the SDK and provide applications with a GLib object[22] that will notify them.

In order to reduce the chances that the system will find itself in a situation where lack of disk space is problematic, it is recommended that available disk space is monitored and applications notified so they can react and modify their behavior accordingly. Applications may chose to delete unused files, delete or reduce cache files or purge old data from their databases.

The recommended mechanism for monitoring available disk space is for a daemon running in the user session to call `statvfs (2)` periodically on each mount point and notify applications with a D-Bus signal. Example code can be found in the GNOME project[23] which uses a similar approach (polling every 60 seconds).

In order to make sure that malfunctioning applications cannot cause disruption by filling filesystems, it would be required that each application writes to a separate filesystem.

Applications not Written for *Apertis*

It may be desirable to run applications (such as Google Earth) that were not written for Apertis. These applications won't understand any custom signals or APIs that Apertis provides, providing yet another reason to minimize those and stick to upstream solutions as much as possible.

References

This document references the following external sources of information:

- XDG Base Directory Specification[24]
- Apertis System Updates and Rollback[25] design
- Apertis Multiuser[26] design

[21] https://www.kernel.org/doc/Documentation/cgroup-v1/memory.txt
[22] https://gitlab.gnome.org/GNOME/glib/merge_requests/1005
[23] http://git.gnome.org/browse/gnome-settings-daemon/tree/plugins/housekeeping/gsd-disk-space.c#n693
[24] https://specifications.freedesktop.org/basedir-spec/latest/
[25] https://www.apertis.org/concepts/system-updates-and-rollback/
[26] https://www.apertis.org/concepts/multiuser/

- Apertis Supported API[27] design
- Apertis Preferences and Persistence[28] design
- Eastlake 3rd, D. and A. Panitz, "Reserved Top Level DNS Names", BCP 32, RFC 2606, DOI 10.17487/RFC2606, June 1999 (*http://www.rfc-editor.org/info/rfc2606*)

[27] https://www.apertis.org/concepts/supported-api/
[28] https://www.apertis.org/concepts/preferences-and-persistence/