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, the format of the distributable application bundle files, 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.

Apertis has first shipped with a custom application framework to address the needs described in this document, seecanterbury legacy application framework[2]. The custom legacy framework is in the process of being replaced with an evolution based upstream components, seeapplication framework[3].

Traditional package managers are unfit for applications

Apertis relies heaviliy 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[4] 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[5] 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 makes creating a new special purpose package management solution much easier, and removes the main reason for customizing an existing solution to fit Apertis-specific needs.
- Much of the complexity in application bundle handling (DRM, rollbacks, communicating security "permissions" to the user) is not part of the existing 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.

[1] https://sjoerd.pages.apertis.org/apertis-website/concepts/multiuser/
[2] https://sjoerd.pages.apertis.org/apertis-website/architecture/canterbury-legacy-application-framework/
[3] https://sjoerd.pages.apertis.org/apertis-website/concepts/application-framework/
[4] https://sjoerd.pages.apertis.org/apertis-website/designs/system-updates-and-rollback/
[5] https://sjoerd.pages.apertis.org/apertis-website/designs/system-updates-and-rollback/

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.

Bundle

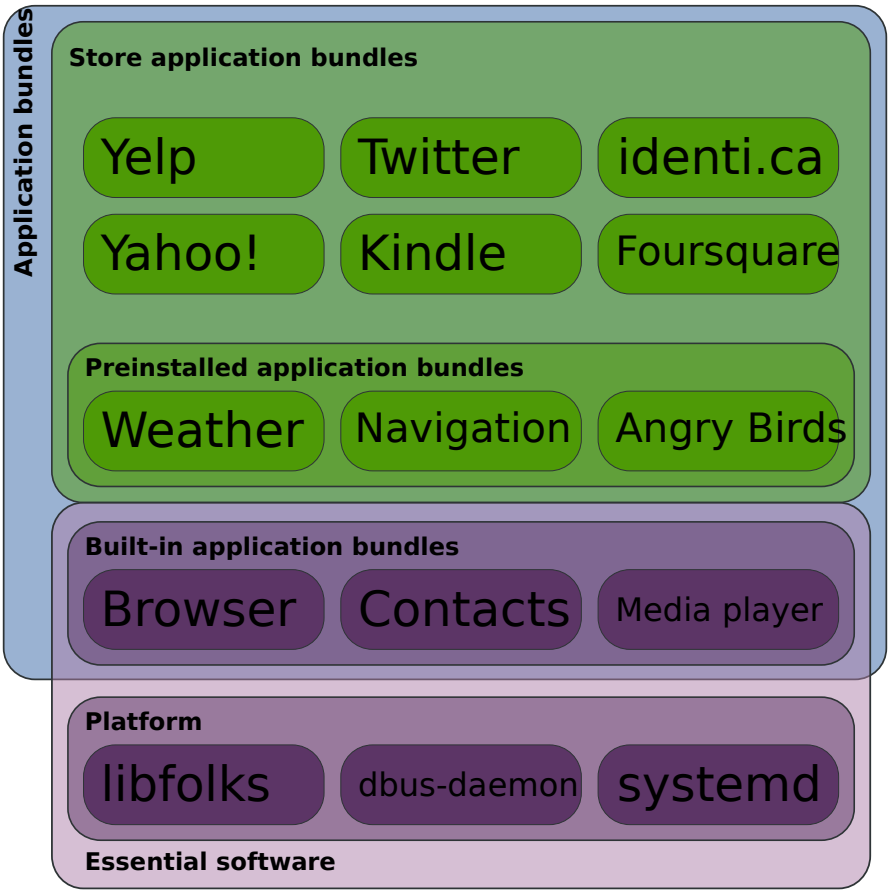
A ***bundle*** or ***application bundle*** is a group of functionally related components (be they services, data, or programs), installed as a unit. This matches the sense with which "app" is typically used on mobile platforms such as Android and iOS; for example, we would say that an Android .apk file 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.

Collabora 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 packaging a developer's store application bundle into a bundle file along with a "store directory"(see Store directory) that contains some generated metadata and a signature. 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.

In order for the the application update system to work properly, each new release of an application needs to have a version number greater than the previous release. The store may need to make changes to application metadata between the developer's releases of that application. To allow the store to increment the application version without interfering with the developer's version numbers, the store will maintain a "store version" number to be appended to the developer's

version number. The store version will start at 1 and be reset to 1 any time the developer increases the application version.

As an example, if a developer releases an application with a version of 2.5 for publication, the store will release this under the version 2.5-1.

> This approach closely resembles the versioning scheme used in dpkg and rpm packages, which combine an "upstream version" with a "packaging revision"

If the store ever needs to push an update to this application without waiting for the developer to create a new version, then the store version can be incremented from 1 to 2, and version 2.5-2 can be released without any intervention from the developer. This is expected to be an uncommon occurrence, but may be done to correct packaging problems, or even to disable an application if it's found to have critical security flaws and the developer isn't responsive.

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 of the AppArmor profile attached to the program(s) in the bundle, if they have them
- 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

We propose to align all of these as follows, matching the approach used by Flatpak for its application identifiers[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,

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

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.

- Application-specific filenames on disk should be based on the bundle name. For instance, `com.collabora.MyUtility` might have its program, libraries and data in appropriate subdirectories of `/Applications/com.collabora.MyUtility/`. Built-in applications should also use the bundle ID; for instance, the Frampton executable might be `/usr/Applications/org.apertis.Frampton/bin/frampton`.

- 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 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.Frampton` bundle could include programs that take the bus names `org.apertis.Frampton`, `org.apertis.Frampton.UI` and/or `org.apertis.Frampton.Agent`.

- Programs in a bundle all use the same AppArmor profile. As a result of the convention that AppArmor profile names are equal to on-disk filenames, its name must start with the installation location based on the bundle ID.

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

Further, to allow upgrade and rollback to be carried out without making the system insecure, we currently require that every store app-bundle's AppArmor profile is deterministically derived from the bundle ID, by being exactly `/Applications/${bundle_id}/**` where `${bundle_id}` represents the bundle ID. (See AppArmor profiles for rationale for this choice.)

For instance, all programs in the `org.apertis.Frampton` built-in app-bundle would run under a profile whose name starts with `/usr/Applications/org.apertis.Frampton/`, and all programs in the `com.example.ShoppingList` store app-bundle would run under a profile named `/Applications/com.example.ShoppingList/**`.

- If a program is a systemd user or system service, the service file should be the program's D-Bus well-known name followed by `.service`, for example `org.apertis.Frampton.Agent.service`. Similarly, 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.Frampton.UI.desktop`.

In particular, using the bundle ID in the AppArmor profile name makes it trivial for a D-Bus service to identify the application bundle to which a peer belongs:

- the service can learn the AppArmor profile name via the standard `GetConnectionCredentials` D-Bus method call
- if the profile starts with `/Applications/`, followed by a syntactically valid bundle ID, followed by either end-of-string or `/`, then the peer is a store app-bundle with the bundle ID that appears after the second `/`
- if the profile starts with `/usr/Applications/`, followed by a syntactically valid bundle ID, followed by either end-of-string or `/`, then the peer is a built-in app-bundle with the bundle ID that appears after the third `/`
- if the profile starts with one of the well-known executable directories for the platform (`/usr/`, `/bin/`, `/lib/` etc.) and does not start with `/usr/Applications`, or the profile has the special value `unconfined` indicating the absence of AppArmor confinement, then the peer is a platform component
- otherwise, the peer is in an unknown category and must not be given any special privileges

The same approach can be used across any other IPC channel on which a process can securely query the peer's LSM (Linux Security Module) context, such as Unix sockets or kdbus.

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

This was implemented in `libcanterbury-platform`.

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).
- Generate an AppArmor profile for the application based on its [permissions]
- Generate the application's Store directory.
- 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 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 vehicle ID – it will only install on a specific vehicle. 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 specific device ID – it will only install on a specific Apertis unit.
- 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.

Application developers will declare the permissions their application depends on in application bundle metadata[12], and Apertis will allow a user to approve a subset of an application's required permissions.

There are some difficulties in allowing users to accept only some of the permissions an application developer expected their software to have access to:

- Some of the permissions will be controlled by an AppArmor profile generated by the application store. The user is merely accepting the profile, actually changing it would not be trivial.
- AppArmor profiles are per-application, not per user. AppArmor profiles would need to be changed on user switch if different users required different permission configurations for the same applications.
- 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.
- 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, there must be two kinds of permissions: required and optional. Required permissions are those that can't be removed

[10] http://developer.android.com/reference/android/Manifest.permission.html
[11] permissions.md
[12] application-bundle-metadata.md

from an application – such as anything granted by the AppArmor profile. If a user chooses to deny a required permission, an application can not run.

Optional permissions are handled by higher level APIs in the SDK and may be influenced by system settings. Apertis-specific "wrapper" services that abstract the functionality of lower level libraries can provide access controls. These wrapper services would act based on the individual user's settings and preferences, so each user would have control over the applications they use. Because these services must act as a trust boundary between apps within the scope of a particular user's account, a privilege boundary must be imposed between the app bundle and the wrapper service: to provide this boundary, they must be implemented as a separate service process, rather than merely a library that is loaded by the application program.

Some permissions may prove to be more of an annoyance than helpful to the user. For example, if Start are employed by vast numbers of applications, users may not wish to be informed every time a new application requires one. 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. Care should be taken to select the appropriate area as different areas are handled differently if rollbacks (See Roll-back) occur. The storage types are:

- Application User – for settings and any other per-user files. In the event of an application rollback, files in this area are rolled back with their associated application.
- Application Everyone – for data that is rolled back with an application but isn't tied to a user account – such as voice samples or map data.
- Cache – for easily recreated data. If the system is low on storage space, it may reclaim cache space for applications that aren't currently running. Caches will be cleared on update and rollback instead of being stored by the rollback system.
- Shared – This area's contents are not touched by the application management framework for any reason. They are also not subject to any form of rollback. This area is intended for storage of videos, music, photos and other data in standard formats that aren't tied to a single application, analogous to `/sdcard` on Android devices. Since Apertis space may be limited, and since it is thought that users will usually want to share media between accounts, the data in this storage area will be accessible by all users. More details on this are available in themultiuser[13] design.

[13] https://sjoerd.pages.apertis.org/apertis-website/concepts/multiuser/

The XDG Base Directory Specification[14] environment variables will guide applications to find the appropriate locations for the different storage types.

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 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

[14] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html
[15] https://code.google.com/p/cryptsetup/

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 compressed "application bundles" containing programs and services that can be launched in a variety of ways – with the limitation that an application bundle can't contain more than one program launchable from the application launcher, or more than one agent.

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

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), permissions it needs from the user, and the versions of system APIs it depends on.

An application bundle may provide back-ends to existing system functionality and add new features to installed software without necessarily adding any new applications to the application manager. These are called "system extensions" and are detailed in System extensions.

Store Applications

Acquisition

Applications will be made available through the application store as compressed files.

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.

A background download service will be provided by Apertis (See Reliable download service). This service will continue downloads if they are interrupted or if the system is restarted. When the download is completed, or if the download is incapable of being completed, a callback will be made to the requester via D-Bus.

The user interface components will be able to query status from the download service in order to display status about the installation – including a completion percentage or a position in the download queue.

Installation

If an application is being installed directly from the store, the application bundle metadata will be downloaded and the user allowed to select a subset of [permissions] to allow, or cancel the installation. If the installation is to proceed, an icon to be displayed in the launcher while the download and installation takes place will now be acquired from the application store.

If the application is being provided from a USB device, the application bundle metadata and icon are extracted from the application bundle.

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 parts of the same archive.

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.

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.

In order to allow application rollbacks to rollback the user data associated with an application, all running instances of an application will have to be closed prior to starting an upgrade for data coherency reasons. The user will be unable to launch an instance of the application during the upgrade process. The system won't recognize services, handlers, or launchable components of the application until the final phase of installation is complete.

Removal

When a user removes the application, any personal settings and caches required by the application will be automatically removed along with any user specific data stored for rollback purposes – 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 have 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), with all their settings and data in exactly the state it had been the last time they used it.

This rollback paradigm has some interesting quirks:

- If a user rolls back an application, all other users of that application will also be rolled back. This allows one user to delete some of another user's settings and personal data.
- 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.
- 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.

- Old versions of applications may break if they interact with online services that changed their protocols, or if Apertis APIs are deprecated.
- Application developers have to think very carefully about what data goes into application storage (and is subject to rollbacks) and general storage (which isn't). In reality, application developers will likely pay very little attention to this distinction and the application store will carry this burden.
- 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.
- Settings management tools like GSettings directly manipulate application setting data and don't currently support the rollback system.

The settings problems can be mitigated by using the persistence API from the SDK when writing applications, allowing Apertis to hide the complexity from the application developer. Each application should have its own database of settings instead of using a single system-wide database.

After application rollback, launching the application now will use the previously installed version with all settings and private data in the state they were before the upgrade.

System Extensions

In the context of Apertis, system extensions may refer to themes and skins which provide global user interface changes, or plug-ins for existing frameworks that aren't intended for extension by regular application developers.

Generally speaking, these will be purchasable add-ons that don't fit into the category of "application", and are instead additions to basic system functionality. Examples include downloadable content that radically changes the visual appearance of all applications under Apertis, or a plug-in that integrates Skype with the contacts and communications software.

While these don't fit the standard role of an application, they are still made available as bundles through the application store, and their installation is still handled by the application manager.

The application bundle metadata will have a list of known extension "types", and extension components inside an application bundle will be handled differ-

ently based on the specified type. There is no comprehensive list of extension types, but "Telepathy connection manager" and "theme" will be the commonly used examples in this document.

System extensions, being outside the realm of regular application developers, are entitled to make assumptions about available libraries and frameworks that applications are not. This makes rolling them back independently complicated, and some simplifications are made by disallowing manual rollback of extensions, and only rolling them back automatically with a system rollback.

Installation

There will be no difference between an application bundle or a "system extension bundle" - and it may even be desirable to deploy an application with supporting system extensions from the same bundle.

Most of the process for installing a bundle with system extensions will be no different than the usual application installation process. However, the "application specific metadata" configuration will include exporting files in the system extension directories.

Depending on the extension, the newly installed extension may not be functional until daemons are restarted, or programs rescan plug-in directories.

Determining what needs to be restarted can be difficult, and could be different depending on what other system extensions have been installed. For simple add-ons like themes, or Telepathy connection managers, no restarts or re-loads should be required, so no special effort needs to be made.

For more invasive system extensions, the application manager can decide based on the extension type in the application bundle metadata whether the functionality requires that the system be restarted. The user should be informed during installation that new features will only be present next time they start their vehicle.

Upgrades

There may be additional steps required based on the extension type – for example, if a theme is being upgraded, the application manager should check if it is the theme currently being used to render GUI elements. If it is, the system may need to switch to a default theme before the upgrade begins, and switch back after the upgrade finishes.

Apart from any extension type specific steps performed by the application manager, the upgrade process will be exactly as described in Upgrades.

Removal

Once again, the process only deviates from Removal by performing any specific actions required by the extension type before following the standard procedure.

As an example, if the extension is a theme, the system should ensure it is not currently in use before beginning the usual removal process.

Rollback

Like regular applications, a system rollback will automatically rollback system extension components.

An intentional rollback will only need special steps at the start of the process, dependent on the type of extension being handled.

Since system extensions are likely to be low level components, it may be a good idea to disallow rolling them back in order to ensure important bug fixes can be deployed.

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 in the application manager 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 of Android and iOS – whether an application (graphical program) is actually running is hidden from the user. The user may either launch new applications or press the "back" button to return the last running application.

From the user's perspective, applications will be persistent. When a user comes back to an application they've previously used, it will be in the same state they left it – even if the vehicle has been turned off and restarted.

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 or widget 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[16].
- 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[17]; activating the application with the corresponding content type will launch the corresponding graphical program.
- Sharing back-ends – Applications may define sharing capabilities that allow other applications to launch them and send a receiver-limited amount

[16] application-entry-points.md
[17] application-entry-points.md

of data. Again, activating an application in this way would normally start a corresponding graphical program.

- Agents – persistent non GUI processes that provide a background component for applications. These will be launched automatically at boot time or immediately after application installation. An application bundle will contain at most a single agent, and the [permissions] will include an "agent" permission to allow users to know they're installing an application that uses one.

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

Collabora feels that the first three types of launch should be under the control of the application manager. See Responsibilities of the application manager

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 agents, and the resource consumption of the agents. 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.

When a program in an application bundle is started by the application manager, in certain circumstances the manager will need to take extra steps. For the first launch of a built-in application, or the first launch after one has been updated, a subvolume will need to be created for storing user data.

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 agents. Agents run for the duration of the user's session, and are only terminated if the system needs to unmount an application's subvolume - either to shut down the system or to upgrade or uninstall the application.

Graphical programs will be notified with a D-Bus message 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 refocused. 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 set the "background" permission in their [permissions]. Ideally they should be designed with a split between foreground and background components (perhaps using a graphical program for the user interface and an agent for the background part) 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.

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 the background permission, 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 (See State saving convenience APIs), 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.

State Saving Convenience APIs

As state saving is a difficult problem for developers, it seems appropriate for Apertis to provide API to assist in performing the task accurately.

A minimal C language API for state saving could be developed consisting of:

- A way to register a callback for a D-Bus signal that requests a save of state information.
- Functions to atomically serialize data structures to application storage.
- Functions to read previously serialized data structures into memory.
- Functions to clear previously saved state.
- Documentation and sample code for using the API.

This API's usage won't be mandatory for application developers.

If it is intended that users have control over which apps save state and which merely close on exit, this API could also provide the code to handle those configuration options.

Maemo provided, through libosso[18] a very simple state saving API. It expected relevant application state be contained within a single contiguous memory region, and provided a call that would write out this single memory area to some abstract storage area that persists across reboots. On startup, an application would attempt to re-read that memory, and if no pre-existing state was present, would start over.

[18] http://maemo.org/api_refs/5.0/5.0-final/libosso/group___Statesave.html

Frozen

Interest has been expressed in creating a state with less resource consumption than background-operation yet still having faster start-up times than ending the process and saving state.

This is a very difficult problem to solve without application intervention – simply dumping the memory contents of a process to long term storage won't be enough to restore the application. File descriptors and network connections are tracked by the kernel and would need to be re-established on process restart. The network connections are especially problematic as the remote end would be unaware of what was happening.

Having applications involved in the process may allow some form of task suspension that reduces the perceived start-up time of a "frozen" application. In response to a signal (presumably over D-Bus) from the application manager, a running application could free easily re-created data, close down network connections and remain dormant until the application manager gave it a signal to resume regular operation.

Resource Usage

To make better use of the available memory, it's recommended that applications listen to the cgroup notification memory.usage_in_bytes[19] 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[20] 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[21] which uses a similar approach (polling every 60 seconds).

In case applications cannot be trusted to properly delete non-essential files, a possibility would be for them to state in their application bundle metadata[22] where such files will be stored, so the system can delete them when needed.

[19] http://www.kernel.org/doc/Documentation/cgroups/memory.txt
[20] https://gitlab.gnome.org/GNOME/glib/merge_requests/1005
[21] http://git.gnome.org/browse/gnome-settings-daemon/tree/plugins/housekeeping/gsd-disk-space.c#n693
[22] application-bundle-metadata.md

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.

Non-Apertis applications should be treated as if they have the background permission – they should not be killed unless the system is extremely low on resources, or they will provide an inconsistent user experience when they don't save state like a native Apertis application.

Responsibilities of the Application Manager

Application life-cycle is dictated by the application manager. When the user interacts with an icon or a link, the application manager is responsible for launching the appropriate application.

Collabora recommends that the application manager also be responsible for content-type-based (MIME-type-based) launching. The manager could provide a D-Bus interface through which it can be asked to launch an appropriate application for a specific content type. A list of available handlers and their invocation details would be gathered from the application entry points.

If multiple applications are capable of handling the same content-types, the user may wish to have a way to select which one takes precedence. One possible solution is to have the application manager provide a dialog any time an ambiguous content-type launch is required, allowing the users to choose their preferred handler, with a check-box that can be set to remember the selection. This is how the Android operating system handles this situation.

There are security concerns when allowing content-type-based application launching – it can potentially lead to an untrusted source (like a web page) being capable of launching a store application with a known security bug.

Giving the application manager control over content-type-based launches can allow it to restrict the usage of content-type-based launching. The manager would be able to deny certain applications the ability to launch handlers for specific data types (perhaps the web browser should never be allowed to launch a handler for a certain data type), filter the handlers available to an application to only allow trusted built-in applications to be used, or allow a system upgrade to blacklist a store application's handler while waiting for a fix from a third party.

The application launcher is itself a built-in application, and as such its storage

is governed by system rollbacks. In the event of a system rollback, all of the launcher's settings (icon placement, for example) will be automatically reset to the state they were in just prior to the last system upgrade.

References

This document references the following external sources of information:

- XDG Base Directory Specification: *http://standards.freedesktop.org/basedir-spec-latest.html*
- Apertis System Updates and Rollback design
- Apertis Multiuser design
- Apertis Supported API design
- Apertis Preferences and Persistence 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*)