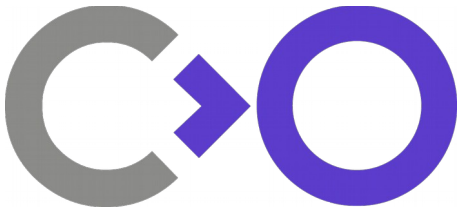

Apertis Applications Design

Author:	Derek Foreman, Simon McVittie
Contributors:	Gustavo Noronha
Version:	0.5.4.1
Status:	Draft
Date:	17 November 2015
Last Reviewer:	Ekaterina Gerasimova

DOCUMENT CHANGE LOG

Version	Date	Changes
0.5.4.1	2015-11-17	• Fix formatting • Delete obsolete document properties • Rename to use Apertis/improve wording
0.5.4	2015-06-02	• Formatting fixes
0.5.3	2015-06-01	• Consolidate jargon terms into a Terminology section • Redraw illustrations as SVG with the new terminology • Correct more references to SAC to refer to Apertis • Consistently refer to AppArmor, Debian, GLib and Linux with those capitalization conventions • Clarify that libraries cannot be a security boundary, so wrappers that impose a security boundary must be a separate process
0.5.2	2015-05-28	• The OS is now named Apertis • Consistently identify apps with reversed DNS domain names • Avoid using “application” as a jargon term since it is unclear: disambiguate into “bundle” and “graphical program” instead • Redefine “platform” to include both system-level and user-level infrastructure (some of which was previously ambiguously included in “built-in applications”), since we are treating them interchangeably in practice • Redefine “built-in application” so it only includes things structurally similar to bundles
0.5.1	2014-12-09	• Update to new template
0.5.0	2014-03-20	• Added section on API to check for resource usage during life cycle of applications (5.5)
0.4.1	2013-05-13	• Respond to internal reviewers
0.4.0	2013-05-03	• Better explain the rollback handling of the var subvolume (3.2) • Clarify that Collabora does not recommend building the application management infrastructure on top of apt/dpkg (12) • Mention that disjoint instances of dpkg on one system isn't well tested (12.1) • Remove outdated information about the depth of the “back” stack (5) • Mention D-Bus activation as a method of process launching for system service developers (5.1) • Allow combinations of DRM locking criteria (2.6)

		• Placeholder icons are never generic (4.1.2, 8) • Application launcher is a built-in application (5.7) • Licenses aren't stored in /var/lib (3.1) • Application synchronization via USB storage (2.5) • More details on external storage encryption (3.3) • Built-in application subvolumes are updated before post-upgrade reboot (4.2.2) • Explain choice of BTRFS for application storage (3.4) • Remove recommendation against SIGSTOP/SIGCONT (5.3) • Applications don't have permission to change AppArmor profiles (6) • Elaborate on hardware used for subvolume mount performance testing (6) • Applications can't contain more than one launchable and one agent (4,8.3) • Give libosso's state saving API as an example (5.3.1) • Add (2.7) on permissions, move some existing text into it • Add the user shared storage to the listings in (7) • Remove the “store wrapper” and replace it with a directory added to the application bundle. • Update installation and update procedures
0.3.0	2013-02-18	• General storage is now shared storage • Rename store.asc to store.sig, explain where it comes from • Upgrade procedure no longer incorrectly implies that multiple instances of an application can be concurrently running • Remove mention of D-Bus activation for process launching – this will not be available to application developers • Outline difficulties in allowing a user to partially approve security permissions • Add a chapter on APT/dpkg as a potential application management back end • Add some information regarding BTRFS subvolume mount times and their impact on system startup time • Add subsection on extending storage capacity with SD cards (3.3) • Clarify “store version” requirement • Mention statistics for a percentage completion indicator (4.1.3) • restore text about the standard method of bringing an application to the foreground (5.2) • Mention storage of license text (4.4,8,8.3)

		• Suggest application developers depend on SDK's persistence API (4.1.5) • Mention possibility of removal of stale application rollback subvolumes on system upgrade (4.1.5, 4.2.4) • Suggest keeping up to two rollback versions (4.1.5) • Alter procedures for built-in applications slightly to simplify boot time procedures (4.2,6) • Add section on reliable download service (10) • Add placeholders for the application manager to display while an application is installing (4.1.2, 8.3) • Discussion of a “frozen” task state (5.4) • Add a large discussion of system extensions (4.3, System Extensions) • Add section on system runtime strorage (3.2) • Use symbolic links to aid non-SAC apps (5.6, 8.3, 9) • Add more details to upgrade procedure (4.1.3) • Add location of application manager's private storage (3.1) • Replace all occurrences of md5sum with sha256 hash • Defer DRM related decisions • Add diagram showing mount points and subvolumes (7) • Add some suggestions for the application release process (2.4)
0.2.2	2012-09-28	• Add a new storage type (3)
0.2.1	2012-09-17	• Incorporate Gustavo Noronha's text: ◦ Rename “from-store” applications to “store”, rename pre-installed to built-in, and add the concept of pre-installed store applications ◦ explain the API break problem
0.2.0	2012-09-13	• Split installation into acquisition and installation (4.1.1) • Mention DRM based on vehicle ID • Minor elaboration on store signing (8.2) • Clarify storage requirements (3) • Clarify reasons for a special purpose package manager (1) • Explain how “system back-ends” are deployable through the application infrastructure (4) • Rollbacks of pre-installed applications are only possible via full system rollback (4.2.4) • Revise multi-tasking paradigm (5.2) • Improve explanation of store versions (2.2) • Mention rollback problem with existing settings software (4.1.5) • Better explain inability to rollback a metadata only application update (4.1.5)

		• Merge “frozen” state with exit – if all applications must save state on exit, frozen is the same as exit • Make infrastructure heavily dependent on BTRFS • Add new chapter on system startup (6) • Mention agents for background operation • Add section on license agreements (4.4) • Prefer “Application manager” over “launcher” to match Bosch documentation • Add a references chapter (13)
0.1.2	2012-07-17	• Address reviewers comments
0.1.1	2012-07-12	• Internal release
0.1	2012-05-01	• Initial version

Table of Contents

Index of Illustrations

Index of Code Listings

1 INTRODUCTION

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 section 1.1 “Terminology”, below.

Apertis is a multiuser system; see the Multiuser design document for more on the specifics of the multiuser experience and the division of responsibilities between middleware and HMI elements. At the time of this writing it is assumed that different user profiles will have access to personalized subsets of installed applications, but applications will be installed in a single application storage area (in other words: all users of an application will use the same version of that application).

While the rest of Apertis will use a package management system based on a mature preexisting solution (See the System Updates and Rollback design), Collabora suggest that a special purpose, lightweight package manager is considered exclusively for application bundles[1]. The main reasons behind this suggestion are:

- Apertis has separate application-bundle and system storage. System storage is read only at run time, and the system package manager is designed to deal with this - new packages installed by the system package manager aren't visible until the next time the system is restarted. As applications should be visible immediately after installation, this makes using the system update framework undesirable.
- System rollbacks are easy to implement with BTRFS subvolumes and existing tools – a full system snapshot is made, updates are then installed. Applications must be individually tracked for rollback purposes.
- 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.

1 Chapter 12,Implementing the Package Manager Using Existing Tools attempts to explain how the system described in this document could be based on the APT package management tools.

- 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.

While some existing systems (notably Meego and openSUSE) use BTRFS snapshots to implement an atomic update mechanism, they're designed for a single “stream” of system updates and won't help rollback applications individually.

System updates are ordered and can be safely rolled back one at a time, but application bundles each have their own stream of updates. For example, if bundle A is updated, then bundle B is installed, a rollback of bundle A to its previous state should not uninstall bundle B. Using one of these existing package managers would not allow rolling back bundles independently from each other.

1.1 TERMINOLOGY

1.1.1 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.

1.1.2 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.

1.1.3 STORE ACCOUNT

Section 2.6 “Digital Rights Management” 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.

2 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*.

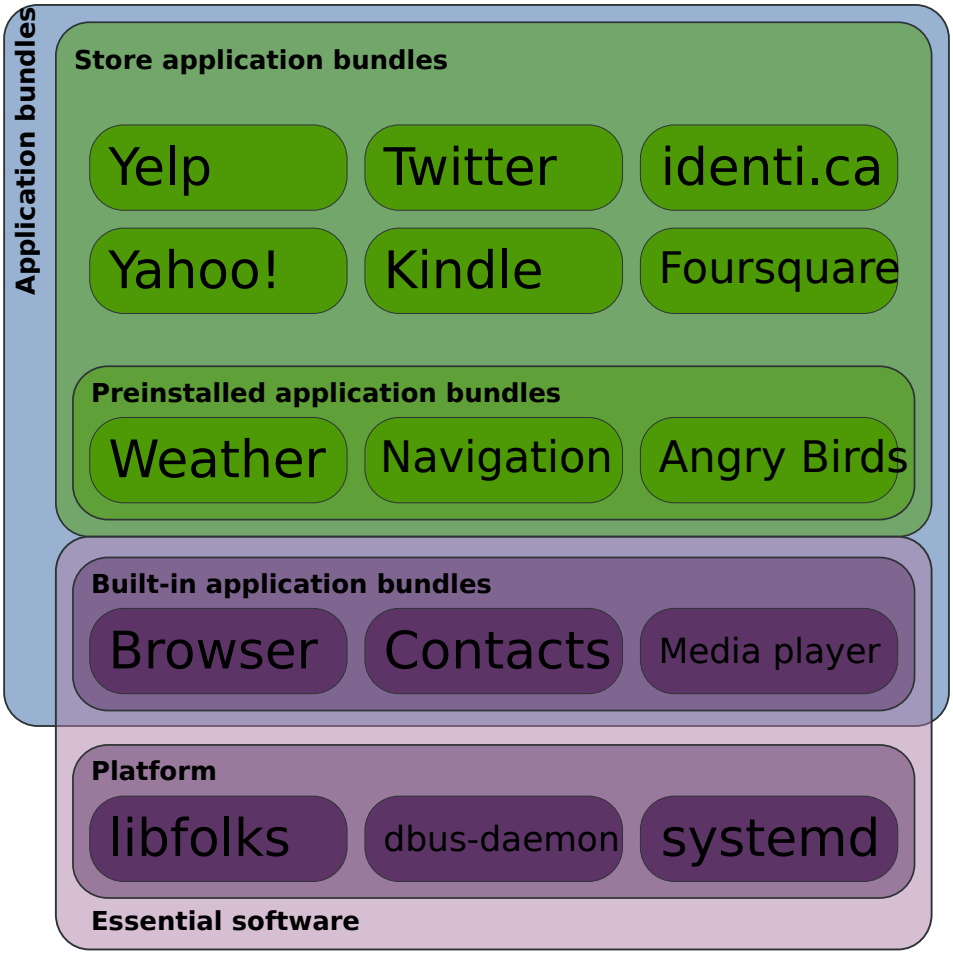

Illustration A: Examples of software components that make up an instance of Apertis

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.

2.1 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[2]; 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

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

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.

2.1.1 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.

2.2 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 8.2, Store Directory) that contains some generated meta-data 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 meta-data 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[3]. 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.

2.3 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, as discussed in 8.3, The Manifest
- 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
- 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:

- The *bundle name* is a case-sensitive string matching the syntactic rules for a D-Bus interface name, i.e. two or more components separated by dots,

3 This approach closely resembles the versioning scheme used in dpkg and rpm packages, which combine an “upstream version” with a “packaging revision”.

with each component being a traditional C identifier (one or more ASCII letters, digits, or underscores, starting with a non-digit).[4]

- 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 name. 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 name; 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[5], 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 name, or any D-Bus well-known name for which the bundle name 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. Its name is the bundle name: for instance, all programs in the `org.apertis.Frampton` bundle would run under the `org.apertis.Frampton` profile.
- 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

4 This scheme makes every bundle name 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.

5 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)

`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 name as 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 is a syntactically valid D-Bus well-known name, then the peer belongs to that bundle
- otherwise, the peer is part of the platform (in particular, any profile containing "`/`" is necessarily part of the platform)

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.

2.4 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 name 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 manifest (8.3, The Manifest).
- Generate the application's store directory (8.2, Store Directory).
- Make the application available at the store.

2.5 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.

2.6 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 section 1.1, “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.

2.7 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 set of permissions[6] 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.

Application developers will declare the permissions their application depends on in the application's manifest (8.3,The Manifest), 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

6 http://developer.android.com/reference/android/Manifest.permission.html

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 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 agents(see 5.1) 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).

3 DATA STORAGE

Applications will have access to several types of writable storage[7]. Care should be taken to select the appropriate area as different areas are handled differently if rollbacks (See 4.1.5, 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.
  Suggested path: for example
  `/Applications/com.example.MyApp/Storage/user1/`
- 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.
  Suggested path: `/Applications/com.example.MyApp/Everyone/`
- 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.
  Suggested path: for example
  `/home/user1/.cache/com.example.MyApp/`
- 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 the Multiuser design.
  Suggested path: `/home/_Shared/`

In an effort to adhere to the XDG Base Directory Specification[8], environment variables will be set prior to application launch as follows (assuming a user named `user1` for the sake of illustration):

- `$XDG_DATA_DIRS` will, in addition to the usual locations (`/usr/share/`, `/usr/local/share/`), point to a directory in the system extensions subvolume (`/var/lib/apertis_extensions`).
- `$XDG_CACHE_HOME` will point to the user's base cache directory (`/home/user1/.cache/`)
- `$XDG_CONFIG_HOME` will point to `/home/user1/.config`, as it does on a conventional Linux system, in order to help non-Apertis applications function correctly. Apertis applications should not use this location to store their settings, and should prefer Apertis APIs for such tasks, such as

7 For low level details of application storage see 7, Application Storage Layout
8 http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

those that implement the Preferences and Persistence design document[9].

3.1 APPLICATION MANAGER STORAGE

The application manager itself will require storage. It should have a directory under `/var/lib` which it can use for its databases and any temporary storage requirements, such as downloading applications to be installed or upgraded.

The application manager may use this area to store blacklists of application versions with critical bugs, icons, or other data required for its operation. It should not store any easily re-created cache data here – it should use `/var/cache` for that data instead.

Since icons and manifest data can be recreated from a scan of the installed application subvolumes, it might be good to consider these to be cache data. Care will have to be taken when making cached copies of icons, as the application bundles don't guarantee that icons have unique names. Icon names will either need to be made unique via directory structures or the cache copies can be given arbitrary names and be looked up in an index.

During application rollbacks (as described in the various sections of 4, Application Management), the application manager will have to update its own database, but a system rollback will automatically reset the application manager's database to the state it was in prior to the most recent system upgrade.

3.2 SYSTEM RUNTIME STORAGE

The system directories in Apertis are read-only at run-time, so some writable space needs to be made available. Directories like `/var` and `/home` must be writable for the system to function. There will be multiple subvolumes in the general storage area to meet these storage needs.

The `/var` directory will be mounted from a BTRFS subvolume. This will provide writable storage for the Linux standard `/var/cache` and `/var/log` directories. This subvolume won't have snapshots taken as part of the system update process.

During a system rollback, `/var/cache` will be cleared to ensure no service is provided with cache files of a newer version than the software. Cache can also be cleared to free up space when the filesystem is too full.

Data in `/var/log` (which could exist if logging is enabled for diagnostic purposes) would not be appropriate to roll back. In the case of a system rollback, this information could be vital in determining what problem caused the rollback to occur.

Over top of this, another subvolume will be mounted for `/var/lib` providing a state directory that will be rolled back with a system rollback. The contents of this directory are created and managed by system components, and should be rolled

9 See the Preferences and Persistence design document available from https://wiki.apertis.org/ConceptDesigns. This document was based on version 0.2.0 of that document.

back with a system rollback.

A third subvolume for Apertis extensions (such as themes or additional system frameworks) will be mounted as `/var/lib/apertis_extensions`. This is kept distinct from `/var/lib` to allow the application manager to make checkpoints of it while performing maintenance operations.

Refer to (7, Application Storage Layout) for more information on what subvolumes exist and where they will be mounted.

3.3 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[10] 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 encrypted SD card would contain application subvolumes in the same way internal storage would (and described in section 7, Application Storage Layout).

10 https://code.google.com/p/cryptsetup/

The device could be recognized by its label as reported by the `blkid` command, and added to the startup application scan in section 6, 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.

3.4 SELECTION OF BTRFS FOR APPLICATION STORAGE

The requirements of the rollback system – that data be stored with its primary application and rolled back if the application is rolled back – make BTRFS subvolumes an attractive solution for application storage.

When an application is upgraded, the user's existing data can be copied as a subvolume snapshot. Subvolume snapshots are copy-on-write, so the backup of the data won't waste a significant amount of space at creation time. This savings will reduce over time if the user overwrites old application data.

This design counts on a few of the special aspects of BTRFS to be present and reliable – that subvolume snapshots can be created from other subvolumes, and that subvolume renames are atomic. These features are present and functional.

4 APPLICATION MANAGEMENT

Applications will be distributed by the application store[11] as compressed “application bundles” (see: 8, Anatomy of an Application bundle) 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.

A manifest file (See: section 8.3, The Manifest) 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.

Built-in applications will be developed in the same manner as store applications, but their bundles will be wrapped in .deb packages and handled by the system update framework. (See the System Update and Rollback design) Built-in applications and their handling are discussed in section 4.2.

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 section 4.3.

4.1 STORE APPLICATIONS

4.1.1 ACQUISITION

Applications will be made available through the application store as compressed files (tar archives with xz compression). These compressed files will be a combination of a store directory containing generated metadata (see: 8.2, Store Directory) and the developer's application (see: 8.1, Bundle Layout).

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 10, 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.

4.1.2 INSTALLATION

In this section and following section, application specific metadata related text

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

has been consolidated and moved to its own section: 9, Bundle Specific Metadata.

If an application is being installed directly from the store, the manifest will be downloaded and the user allowed to select a subset of application 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 manifest and icon are used from the store directory in the application bundle.

Once a compressed file from the application store is acquired, either from a USB device or directly from the application store, the file will be used by the application management framework to perform the following steps:

1. A new directory will be created in temporary storage.
2. Only the store directory contents will be unpacked to that directory.
3. If this is an installation from a USB stick, at this point an icon can be displayed in the launcher and the user can be asked to approve permissions. If this is a direct install, this will have been set up prior to downloading.
4. The signature in the `store.sig`[12] file will be validated with the application store's public key to ensure the included `store.json` file was generated by the store.
5. The rest of the application bundle will be decompressed in a new BTRFS subvolume named `installer-temp`. If this subvolume already exists it will be deleted.
6. The SHA256 hashes of all unpacked files will be checked against the list available in the `store.json` file. This confirms that the files are an application that has been approved by the store for distribution.
7. The application specific metadata will be configured.
8. The installer-temp subvolume will be renamed based on its store ID from the manifest and mounted in the application directory (Listing 2: Layout of the application directories).

If any of the steps above fail, the subvolume will be removed and the user should be notified by the application handling software with an appropriate error message indicating the cause.

Once the application is successfully installed the temporary directory created in step 1 will be deleted, as will the application bundle file, if the installation took place over a network. Bundles provided on USB devices will be left untouched.

If a user tries to install an application that is already installed by another user of the system, no download will occur and the application will be set up for the user. This setup will include the creation of a directory for that user in the storage directory in the application's BTRFS subvolume, and will not occur unless the user

12 `store.sig` is a gpg external signature generated by the application store with the store's private key.

agrees to any required permissions in the application's manifest. If a newer version than the one installed is requested, the application will be updated for all users.

The outlined method does not support concurrent installation of multiple applications. If a user wishes to install an application before the previous installation completes, the application management framework should create a queue of requests and process them serially.

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 a file in the store directory, and use tar's “checkpoint”[13] facility to send an update to the user interface occasionally.

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.

4.1.3 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.

Downloads will take place through the reliable download service (10, Reliable Download Service) and be passed to the application management software on completion. During the download process, the download service should provide some statistical information (current and average transfer speed, bytes received) so the HMI can display a progress indicator to the user.

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

13 http://www.gnu.org/software/tar/manual/html_section/checkpoints.html

a user by e-mail when updates are available, or show a list of updated application when the user logs in to the store.

Sometimes a system upgrade will break applications by changing API/ABI - A problem that is the subject of the Supported API design. Using information from the manifest, the application manager can verify whether an upgrade will render an installed application unusable before the upgrade is carried out, and allow the user to decide to wait for updated versions of the applications they care about.

Collabora envisions the update manager will provide a UI showing which applications would stop working, which already have updated versions and so on, similar to how Firefox handles its add-ons during upgrades, with the important distinction that the information would be provided before the upgrade is performed.

Once a newer version application is acquired, it will be handled in the following way:

1. If the application is running, it will be closed. Launching the application will be disallowed for the duration of the update.
2. The application specific metadata will be removed.
3. The application's subvolume will be unmounted.
4. A new snapshot named `installer-temp` will be created from the contents of the application's subvolume. If the `installer-temp` subvolume already exists, it will be deleted.
5. The application bundle directory will be removed from `installer-temp`.
6. A new directory will be created in temporary storage.
7. Only the store directory will be unpacked to that directory.
8. The signature in the `store.sig`[14] file will be validated with the application store's public key to ensure the included `store.json` file was generated by the store.
9. The rest of the application bundle will be decompressed into the `installer-temp` subvolume.
10. The SHA256 hashes of all unpacked files will be checked against the list available in the `store.json` file. This confirms that the files are an application that has been approved by the store for distribution.
11. The `installer-temp` subvolume will be renamed and mounted in the application storage directory.
12. Only one prior version is to be stored for rollbacks, so If an older version was being saved it will be deleted.
13. The application specific metadata will be configured.
14. Launching the application will be allowed and the agent, if present,

14 `store.sig` is a GPG external signature generated by the application store with the store's private key.

will be started.

As application updates are system-wide, if a user performs an update of an application, all users with that application installed will also be upgraded to the latest version.

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.

It is possible that the application store will push a version that changes only the application meta-data (the store version changes, but the application version does not). This will result in a simplified upgrade process.

1. All running versions of the application will be closed and launching disallowed.
2. The application's subvolume will be unmounted.
3. The application specific metadata will be removed.
4. The application's subvolume will be remounted.
5. The application specific metadata will be configured.
6. Launching the application will be allowed.

This is to prevent using up a rollback “slot” with a meta-data only update.

4.1.4 REMOVAL

Apertis is a multi-user system and different users may use a different subset of the installed applications - both to prevent wasted space in the application manager view and to prevent running background services a user doesn't need. In order to allow a user to remove an application they don't want without removing it from all accounts, applications will be reference counted. Once the last user with access to an application removes it from their profile, the application – in the case of a store application - can be removed from the system. If an old version of the application is available for a rollback, it will also be removed at this time.

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.

4.1.5 ROLL-BACK

Apertis should 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.

The actual rollback process is simple.

1. If the application is running, stop it. Prevent launching for the duration of the rollback procedure.
2. Remove the application specific metadata.
3. Unmount the application's subvolume.

4. Delete the new version's subvolume.
5. Mount the old version's subvolume.
6. Configure the application specific metadata.
7. Allow launching the application.

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

Old application subvolumes stored for rollbacks may eventually be made un-startable by system upgrades. It may be worthwhile to have Apertis check for this during a system upgrade and remove any application rollback subvolumes that can't be started by either the primary or fallback system subvolume. (Details of the system rollback process are in the System Updates and Rollback design)

If only a single rollback version is kept around when an application is updated, there is no guarantee that any version is capable of running after a full system rollback. For this reason it may be worthwhile to keep up to two snapshots for rollback purposes – the most recently replaced, and the newest version still capable of running if a system rollback occurs.

4.2 BUILT-IN APPLICATIONS

4.2.1 INSTALLATION

Built-in applications will be installed by the Debian package management tools (dpkg and apt) at system image creation time.

The read-only storage area will have a directory structure under `/usr/Applications/` that matches the format of the store application directory. (See Listing 2: Layout of the application directories)

Since the built-in applications are stored in a read-only subvolume, the user data for built-in applications must be stored in writeable storage somewhere else. User data will be placed in subvolumes in the store application directory in the user storage area. The application manager will create these subvolumes before launching built-in applications if they don't exist yet.

4.2.2 UPGRADES

Upgrades for built-in applications will be part of the regular system update process as defined in the System Updates and Rollbacks design. The practical implication of this is that the user won't actively accept updates for built-in applications, they'll be silently upgraded along with the rest of the system.

When built-in applications are upgraded as part of a system update, the application manager needs to create the appropriate subvolumes for new versions based on snapshots of the previous version's storage. This will be done at the end of the update process before the system is rebooted.

4.2.3 REMOVAL

Built-in applications are considered to be part of the base system, and have no removal procedure.

4.2.4 ROLLBACK

While there will be no explicit support for rolling back built-in applications, they are all implicitly rolled back as part of a system rollback.

Special care should be paid to built-in application subvolumes if the system reclaims unstartable subvolumes as described in 4.1.5,Roll-back, as any built-in application that hasn't been launched since the last upgrade may not have a current subvolume. The data in the existing subvolume may be for a version of the application that can no longer be started, but the data may still be required in the future.

If these subvolumes are to be reclaimed, this should be done with the additional constraint that the most recent subvolume is never deleted.

4.3 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 bundle manifest will have a list of known extension “types” (Section 8.3, The Manifest) and extension components inside an application bundle will be handled differently 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.

These extensions must be stored in the user storage area for several technical reasons:

- Adding anything to the read only system area requires a reboot before the new components become active on Apertis.
- Performing a platform upgrade (As described in the System Updates and Rollback design) replaces everything in the system area, and would remove any extensions installed there.
- Making a change to the system area replaces one of the two rollback slots, so installing additional software there could reduce robustness

against certain low likelihood failures.

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.

4.3.1 INSTALLATION

There will be no difference between an application bundle or a “system extension bundle” - and it my 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 the creation of symlinks in the system extension directory (`/var/lib/apertis_extensions`).

Since symbolic links are being made to files or directories that now exist in an application directory, there is no concern that parts of the system will see incomplete files. However, 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 manifest whether the new 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.

4.3.2 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 (4.1.3, Upgrades).

4.3.3 REMOVAL

Once again, the process only deviates from (4.1.4, 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.

4.3.4 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.

4.4 LICENSE AGREEMENTS[15]

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.

15 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.

5 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.

5.1 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.
- 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 manifest file; 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 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 manifest will have an “agent” permission to allow users to know they're installing an application that uses one.

Collabora feels that the first three types of launch should be under the control of the application manager. (See: section 5.7, 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.

5.2 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 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 set the “background” permission in their manifest. 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 – (sending a _NET_ACTIVE_WINDOW message to the root window, which will be handled by the

window manager), though it may be worth considering handling this via D-Bus as well.

5.3 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 5.3.1, 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.

5.3.1 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[16], 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 doesn't persist across reboots. On startup, an application would attempt to re-read that memory, and if no pre-existing state was present, would start over.

5.4 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

16 http://maemo.org/api_refs/5.0/5.0-final/libosso/group__Statesave.html

resume regular operation.

5.5 RESOURCE USAGE

To make better use of the available memory, it's recommended that applications listen to the cgroup notification *memory.usage_in_bytes*[17] 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 signal that they can listen for.

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[18], 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 manifest where such files will be stored, so the system can delete them when needed.

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.

5.6 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.

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.

Additionally, user data for non-Apertis applications might not fit within the Apertis rollback system well. The rollback system changes the location of stored settings, and applications that don't understand it will store their settings in standard locations (like `~/.config`) instead. This also makes cleaning up on application removal more difficult.

It is anticipated that the majority of non-Apertis applications can be fooled into reading their data from an application subvolume with the use of symbolic links.

17 http://www.kernel.org/doc/Documentation/cgroups/memory.txt
18 http://git.gnome.org/browse/gnome-settings-daemon/tree/plugins/housekeeping/gsd-disk-space.c#n693

The manifest will specify where symbolic links need to be created to make this possible (refer to section 8.3, The Manifest)

If the symbolic link configuration in the manifest changes between versions for a non-Apertis appliation, then a system rollback might result in stale symbolic links being left behind. The application framework should occasionally scan through manifest files to garbage collect these stale symbolic links.

Deploying non-Apertis applications from the app-store will be possible if the applications bundle their own versions of libraries not in the Supported API document. It may, however, be mandatory to recompile these applications for Apertis in order for the run time linking to work properly.

5.7 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 handles and their invocation details would be gathered from manifest data of installed applications.

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.

6 BOOT TIME PROCEDURES

The application infrastructure relies heavily on BTRFS subvolumes – to allow the rollback system to function, each store application is housed in its own subvolume in the user storage area along with the user data it operates on. Built-in applications are part of the system image, but they also need a subvolume for each in order to house the user's data. For more details of the subvolumes present in the system, see 7, Application Storage Layout.

Applications won't be available after the system boots until all the appropriate subvolumes have been mounted. To make sure all the appropriate subvolumes are ready for use, the following prodedure needs to be followed at boot time:

1. The `var` subvolume will be mounted – if the `var` subvolume does not exist it will be created and populated with default state from a directory in the read only system area.
2. The `/var/lib/` subvolume will be mounted – if the it does not exist, it will also be populated with default state.
3. The `apertis_extensions` subvolume will be mounted under `/var/lib/` - this will also be created with default state if not present. If a snapshot of the `apertis_extensions` subvolume exists, it indicates an application management function was interrupted; the snapshot will replace the interrupted volume.
4. If the `installer-temp` subvolume is present – indicating that an application installation or upgrade was interrupted – it will be deleted. No automatic installation re-attempts will be made as part of the system boot. A user may re-try an application installation, and the application update system will perform its usual automatic updates again later.
5. For built-in applications, if a subvolume with a higher version than the built-in application exists, it will be deleted, as this indicates a system rollback has occurred.
6. Appropriate application subvolumes will be mounted.
7. Application AppArmor profiles will be activated – this is done by the application manager as the applications are unable to install or remove AppArmor profiles.
8. If applications use agents, they will be started after the subvolumes are mounted.

This process depends on mounting a BTRFS subvolume being a fast operation, as the number of subvolumes to be mounted at boot increases with the number of installed applications. Some simple tests have been performed to measure the impact of mounting a BTRFS subvolume.

In testing, on a Sabrelite development board (1GHz, Quad core) with a class 10 SD card, 300 BTRFS subvolumes were mounted in just under 1 second, giving around 3.3ms for a single BTRFS subvolume mount. This implies that there will be an additional 3.3ms of startup time per installed application – including built-in

applications - due to BTRFS overhead.

On an Apertis device that's been used for a long time, several system updates may have occurred, and a large number of application subvolumes may have accumulated. Some applications will have two or even three subvolumes to choose from at boot time. The correct subvolume to mount in step 4 is always the highest version still capable of launching on the running version of the operating system.

If the system has been rolled back, this may not be the highest version available. Application rollback results in the deletion of the unwanted new version, so there is no chance of accidentally rolling forward to a version the user does not want.

7 APPLICATION STORAGE LAYOUT

As shown in Listing 1: Application Subvolumes, the general storage subvolume will contain subvolumes for all applications currently installed on the system – whether they're store or built-in.

The subvolumes for built-in applications will only contain user storage, as the application itself is already stored in the read-only system area.

The names of the application volumes are derived from their type – app-str for store apps, app-blt for built-in apps – and their bundle name and version number.

```
<general storage subvolume>
    |- apertis_extensions
    |- Applications
    |    |- app-str-com.example.MyApp-1.2.2-1
    |    |- app-str-com.example.MyApp-1.0.8-2
    |    |- app-blt-org.apertis.Frampton-2.5.1-1
    |    |- installer-temp  (only present during application installation)
    |- User storage
    … <other subvolumes unrelated to application storage>
```

Listing 1: Application Subvolumes

A subset of these subvolumes will be mounted under the Applications directory as shown in Listing 2: Layout of the application directories.

```
var
    |- lib
    |    |- apertis_extensions <system extension subvolume>
usr
    …
    |- Applications         (built-in applications)
    |    |- org.apertis.Frampton
    |    |    |- Bundle
    |    |    |    |- <Application bundle contents>
    |    |    |- AppArmor
    |    |    |    |- <AppArmor profiles>
    …
home
    |- _Shared              (shared storage among all users)
    |- UserName             (user's private storage)
    …
Applications                (store applications, mounted from subvolumes)
    |- com.example.MyApp <mounted from app-str-com…MyApp-1.2.2-1 subvol>
    |    |- Application
    |    |    |- Bundle
    |    |    |    |- <Application bundle contents>
    |    |    |- AppArmor
    |    |    |    |- <store generated AppArmor profiles>
    |    |- User Storage
```

```
|    |    |- Username
|    |    |    |- <Username's settings and data for this application>
|    |- Everyone Storage
|    |    |- <User invariant application storage>
|- com.example.MyOtherApp <mounted from app-blt-…MyOtherApp-2.5.1-1>
|    |- Storage
|    |    |- Username
|    |    |    |- <Username's settings and data for this application>
```

Listing 2: Layout of the application directories

Illustration B: Mount Points shows the mapping of BTRFS subvolumes on the storage device to mounted filesystems. The (read-only) root filesystem is provided by one of the two system subvolumes, and other subvolumes are mounted on top of it.

The root subvolume of the general storage subvolume will be mounted on `/General`. Mounting the root subvolume in this way allows the system to view a list of all subvolumes on the device and inspect their contents before mounting the subvolumes individually in other places.

Only one subvolume per application is mounted – under normal circumstances, this will be the one with the highest version. After a system rollback it might be an older version if the most recent is unlaunchable.

The `/var` subvolume is mounted in place of the usual `/var` directory, for persistent system state such as cache and log files. `/var/lib` is mounted from the `varlib` subvolume for state that should be tracked by system rollbacks. An `apertis_extensions` subvolume will be mounted under `/var/lib` for system extensions.

The user storage space will be mounted on `/home`. Since application data will be stored with the applications, most stored data will exist in the application subvolumes, but having `/home` directories will be important in making non-Apertis applications function, as well as providing user specific storage for many standard libraries.

Shared storage among all users will be in `/home/_Shared` (and it should be ensured that a user can't create an account with the name "`_Shared`" to confuse the system). `/home/_Shared` will not be a regular user directory, and will not contain any special per-user files.

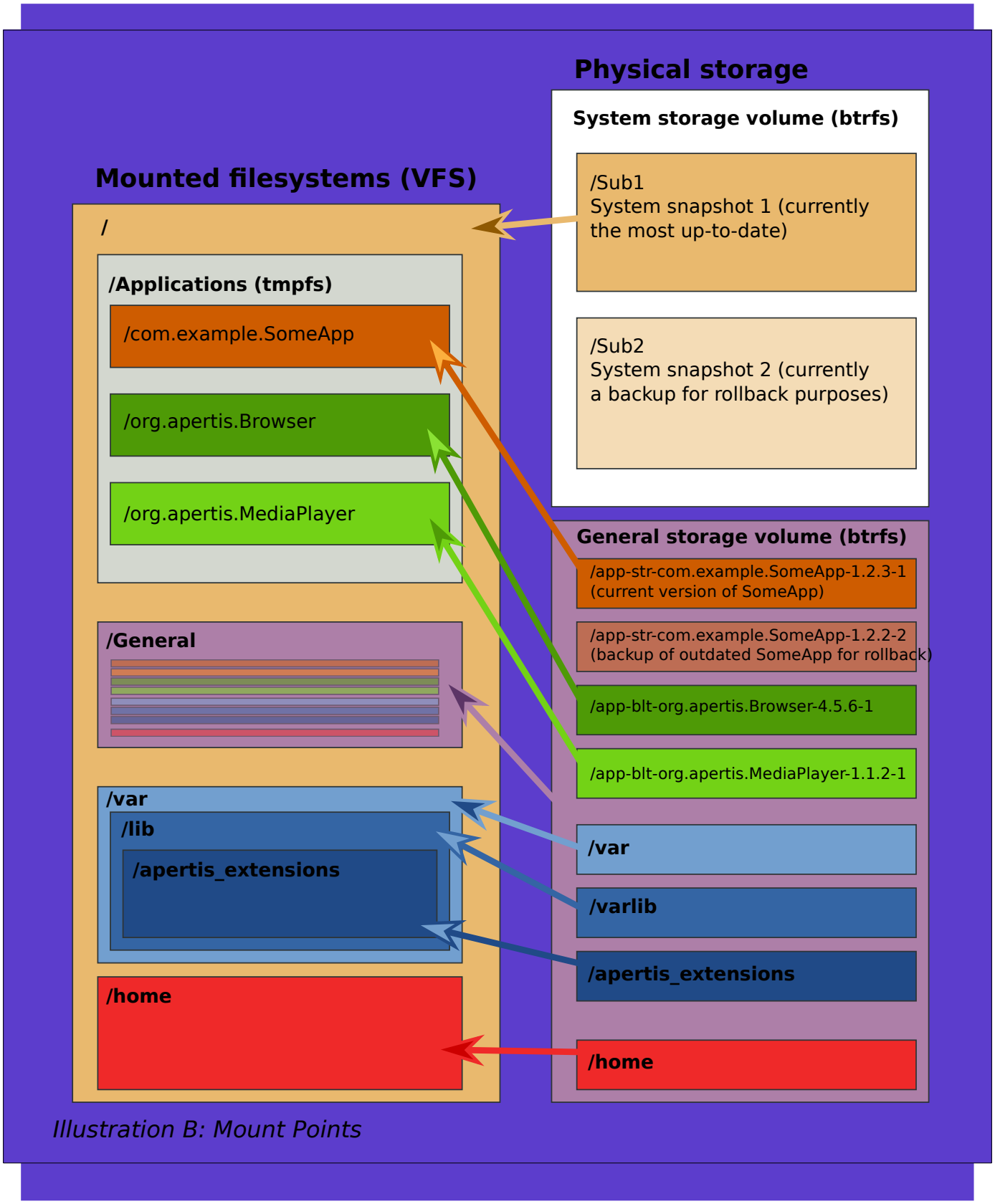

Illustration B: Mount Points

8 ANATOMY OF AN APPLICATION BUNDLE

Store applications are distributed as bundles that contain an application and a directory full of store meta-data. Built-in applications will be further packaged into .deb files to be pushed through the system upgrade infrastructure.

8.1 BUNDLE LAYOUT

It is common for applications in a UNIX-like system to have their files installed in the system following the system hierarchical organization: binaries go in a directory, usually **/usr/bin**, data files such as icons and translations go on a second directory (**/usr/share**), and so on. Apertis applications will not follow that pattern, and will instead be self-contained, having all of their files installed to a separate directory hierarchy.

All applications will have a JSON[19] file called **manifest.json** in their root directory, containing important information about the application, such as its name, icon, and the application's security and Internet connection requirements.

See chapter 8.3,The Manifest, for more details on what the manifest will contain and how it will be used.

Libraries may also be included in the application's directory. Apertis will ship with a number of APIs that can be used by applications, with varying levels of backwards-compatibility guarantees. In case the application also needs to use any *External APIs*[20] it needs to bundle the libraries which provide them.

Since Apertis will ship in different market regions, applications must be capable of supporting multiple languages. The internationalization (i18n) process is more clearly explained in the Internationalization design. A simplistic explanation for the purposes of this design is that all human-readable text in an application is provided in multiple languages – each language in a separate “.mo” file. Application bundles will need to provide .mo files for every language they intend to support – if no .mo file is found for Apertis' configured language, then text will be displayed untranslated in whatever form the software developer used when writing the application.

For some applications (such as those built with open source technologies) it may be required to make license information available to the user. For example, when distributing an application built from source code licensed under the GPL (GNU General Public License), it is required that a copy of the GPL text be provided, usually in the form of a file named “COPYING”. For GPL licensed software, information on how to obtain the source code must be made available to the user.

The “license” directory in the bundle is intended for this kind of information, and will only be present if license information is required for the specific application. Files from the license directory should be made available to the user in some way, but don't necessarily have to be presented to the user. The manifest (see: 8.3, The Manifest) specifies a way to require a user to agree to a license.

19 http://en.wikipedia.org/wiki/JSON
20 See discussion on External APIs in the Supported APIs design

The following is a sample of what the directory structure for an application may look like:

```
[com.example.MyApp-1.2.2-1.tar.xz]
    |- Application Store                (See 8.2:Store Directory)
    |    |- store.json
    |    |- store.sig
    |    |- com.example.MyApp-1.2.2-1.profile
    |    |- placeholder_icon.png
    |    |- manifest.json
    |- manifest.json                    (See 8.3:The Manifest)
    |- bin
    |   |- mybinary
    |   |- myotherbinary
    |- lib
    |   |- libexternal.so
    |- data
    |   |- icon.png
    |   |- data.xml
    |- extension
    |   |- theme
    |   |    |- ThemeName
    |   |    |    |- <theme data>
    |- i18n
    |   |- en_US
    |   |    |- LC_MESSAGES
    |   |    |    |- MyApplication.mo
    |- license
    |   |- COPYING
    |   |- readme.txt
```

Listing 3: Example application bundle directory structure

8.2 STORE DIRECTORY

The store directory contains store generated meta-data: a secure hash file (`store.json`), a signature (`store.sig`) generated by gpg using the application store's private key, and an AppArmor file containing profiles for each executable in the application.

The `store.json` file will be a JSON format file containing a list of all archive contents (except `store.json` and `store.sig`) with their SHA256 hashes, and any DRM restrictions for this bundle.

The signature file will contain a secure signature generated for the `store.json` file with the application store's private key. The signature will be generated using the application store's private 2048 bit RSA key. By validating the signature with public keys included in the system image (or provided through system updates[21]), Apertis may make the following assertions:

21 See the Security design for details

- `store.json` was generated by the application store.
- Any file whose name and SHA256 hash pair matches the one in `store.json` is a file that was approved by the application store.
- If all files match the SHA256 hashes, the archive contains an approved application that may be installed on Apertis.

In addition to store generated files, the store directory will also contain copies of the application manifest and icon. These are present so external media updates can have the same user experience as on-line ones – such as an icon present in the launcher, and permission approval prior to decompressing the entire application.

The directory will always be the first in the archive in order to allow it to be quickly extracted before beginning an installation from external media.

8.3 THE MANIFEST

The manifest is a JSON format file that aggregates application meta-data into a single convenient file for developers – Listing 4 provides an example. Some of the information will be used to determine if the application is installable or not, other sections will be parsed by the application store to generate system configuration files.

```
{
    "title":[
          { "locale":"en_US", "title":"My Application" }
    ],
    "api-version" : "1",
    "author":"Your company",
    "bundle-id":"org.apertis.MyApp",
    "application-version":"1.2.2-1",
    "launch": [
                "title":[
                      { "locale":"default", "title":"MyApplication" }
                      { "locale":"en_US", "title":"MyApplication" }
                ],
                "icon":"data/icon.png",
                "exec":"bin/mybinary"
          ],
    "handler":[
                "title":[
                      { "locale":"default", "title":"MyApplication viewer"
}
                ],
                "mime-types":"text/csv;text/plain",
                "exec":"bin/myotherbinary %f"
          },
    ],
    "links": [
```

```
            { "target":"MyApplication_config_dir",
              "linkname":".config/MyApplication" }
    ],
    "extensions": [
            { "theme":"ThemeName",
              "connection-manager":"CMName" }
    ],
    "placeholder":[
            "icon":"data/placeholder_icon.png"
    ],
    "agent":[
                "title":[
                      { "locale":"default", "title":"MyApplication
agent" }
                ],
                "exec":"bin/agent"
                ],
    ],
    "license":[
            "readme.txt"
    ],
    "permissions":[
            "network", "location", "agent"
    ]

}
```

Listing 4: Example manifest.json file for a hypothetical application bundle called `org.apertis.MyApp--1.2.2-1.tar.xz`

The “`bundle-id`”, “`application-version`”, and “`author`” tags in conjunction with the store version supplied by the application store will be used to construct the name of the application bundle and determine the target directory in the storage area. If a version of the application is already installed, the version number in the manifest will be used to determine if an upgrade is being performed, and prevent installation of older versions of already installed software.

It may be desirable to have the application's name displayed in the local language, so any title strings can be specified in multiple languages. Only the “`default`” entry is mandatory, and will be the fallback if the system if configured for a language the application hasn't been translated to.

The manifest will also specify the minimum system version (API version) required to run the application. This information is used to prevent installation of an application on an old version of Apertis that doesn't provide a compatible run-time environment.

To let the application manager make appropriate decisions, all application bundles must use the same format for their version strings. Collabora recommends making “major.minor.patchlevel” the required format, with major as the most significant number, followed by minor, followed by patchlevel. There will be a

“store version” appended to the version string (in the version string “`3.2.4-1`”, the `1` is the store version). The store version allows the store to push an update even if the application version hasn't changed, and it will be the lowest significant figure. For example, version “`2.2.0-1`” is newer than version “`2.1.99-4`”.

How an application developer chooses to set the version numbers is ultimately their decision; Apertis only defines how they are to be compared. The store version will re-start at `1` any time the application version is increased, and will be incremented if a new intermediate release is required.

An application bundle can contain multiple programs to be launched by content-type (MIME type) association. These will be defined in the “`handlers`” section. Each entry will specify an association between a binary and its accepted mime-types. In addition to a file name to launch, information may also be provided to create proper command lines for the executable.

Since the installation process is not instant, placeholder icons must be provided and specified in the manifest. This icon will be copied into the store directory by the application store during publication. It will be displayed by the application manager instead of the application until the installation is completed. The application launcher will also be able to display a progress indicator or – if multiple applications are being installed – a position in the install queue.

The `links` section is for assisting Apertis in running non-Apertis applications. Symbolic links can be used to make programs that expect to store files in locations like `/home/user1/.config` actually use locations in the application subvolume. The `linkname` field is relative to the installing user's home directory, and the target is relative to the application subvolume.

A section called “`extensions`” will also result in the creation of symbolic links. This is how Apertis will support system extensions such as themes. The type of extension will be specified, along with the directory inside the application bundle that contains the extension data. The Application manager will use this information to create appropriate symbolic links in the `apertis_extensions` directory to make these extensions active.

Agents will be specified in the manifest, with a localized list of names for the agent, along with the location of the executable file to launch. Since agents can be long running and have an impact on device performance, any application with an agent should also set the “`agent`” permission (permissions are described later in this section) so the user can choose not to install the application.

Optionally, a single “`launch`” item may be specified to provide an icon for presentation in the application manager. If no icon is presented it won't be obvious to the user that they have the application installed, so the application store screening process should carefully consider whether an application should be allowed to install services and type handlers with no icon for the manager.

Applications may only have a single launch item and a single agent – no manifest should ever contain more than one of each. Bundles that provide only system extensions would not specify either.

All executable filenames will be relative to the top level of the application bundle and can not contain “`..`” or start with a “`/`”. This will be tested by the store's bundle generating tools so the application management framework doesn't have to perform the test.

The optional “`license`” section defines any license files that must be agreed to by the user before using the application – files in the license directory of the bundle but not mentioned in this section will still be copied the device, and the HMI components should provide some way to view that information later.

Finally, a “`permissions`” array which is used by the store to create an application specific AppArmor profile for inclusion in the bundle. As stated in section 2.7:Permissions, the user will be able to partially accept permissions – the permission array in the manifest is a complete list of the permissions the application would use if allowed.

9 BUNDLE SPECIFIC METADATA

To simplify the descriptions of installation and update procedures, this chapter provides the details of application specific metadata (AppArmor profiles, `.desktop` files, etc) and how they're handled during application management.

This data will be described in the application manifest and can be set up and torn down by the application manager using the manifest data and files in the application bundle.

Any time the application management system removes, upgrades, or rolls back an application, the metadata for the existing version is removed. At the end of the procedure, or at the end of a new application installation, new metadata is configured before the application can be launched.

For the AppArmor profile specifically, it's important to make the old version inactive before removing an application, since the old version will no longer be present after the operation completes. Likewise, a new profile should always be made active immediately after installation or update to ensure that AppArmor is protecting the system.

The system will have a special BTRFS subvolume called the `apertis_extensions` subvolume which will be used to store certain data in a central location. It will be rolled back as part of a system rollback.

System files (files that would normally exist in the read only system directories, such as Telepathy connection managers) will be symbolically linked into the `apertis_extensions` volume. When an application rollback occurs, the appropriate links can be removed and replaced with older versions from the appropriate application subvolume.

To protect the application management system against power loss, the `apertis_extensions` subvolume will be saved with a BTRFS snapshot prior to operations taking place, and the snapshot removed once the operations are complete. At boot time, if the backup snapshot exists, the system will assume the process was interrupted, and safely revert the change.

9.1 METADATA CONFIGURATION

System extensions as specified in the manifest will have symbolic links made in the `apertis_extensions` subvolume.

The bundle's AppArmor profile will be made active.

The application manager will add the bundle's manifest data to its database.

If the bundle has symbolic links specified in its manifest, they will be created.

If present, an application agent can be started.

9.2 METADATA REMOVAL

If an agent is part of the application bundle, it must be stopped.

If the bundle specifies symbolic links in its manifest, they will be removed.

The application's AppArmor profile will be made inactive.

The application's symbolic links in the `apertis_extensions directory` will be removed.

The application manager will remove the application's manifest data from its database.

10 RELIABLE DOWNLOAD SERVICE

Since applications and system updates may be large, and Apertis may have inconsistent connectivity or be shut down frequently (such as when the vehicle is parked or being refueled), a reliable download service will be written to provide a stable method of acquiring application bundles and system updates.

To be useful, the download service must have the following properties:

- Obeys connectivity policy – if multiple network connections are available or have usage charges, they are used in accordance with the customers preferences.
- Resumes partial downloads automatically when the system is restarted or when network connectivity becomes available.
- Provides a callback (over D-Bus) when a transfer completes.
- Can queue multiple downloads and complete them sequentially.
- Provides completion and transfer speed statistics to HMI components.

Such a service is useful for the application management and system update frameworks, but could also be very useful to application developers. It may be worth considering allowing applications to use the reliable download service.

If the download completion callback is handled by the application manager, then an application need not be running while using the service, and can be automatically started and brought to the foreground to handle the completion event.

In order to keep the service simple, it will download files sequentially, and attempt to complete requests in the order they were received. Future improvements to the service could include the use of multiple concurrent downloads and prioritized queuing of submissions.

11 NEW SOFTWARE COMPONENTS

Not all of the functionality in this design currently exists in available software. While this is true of some parts of other designs, it may be particularly obscure in this one, so a brief summary of new development is provided here:

- The convenience APIs for state saving
- A D-Bus interface for querying handler applications
- The application management infrastructure
- A reliable download service for acquiring application bundles from the store
- The boot time components for mounting and creating appropriate application subvolumes
- gsettings/dconf will need to be extended to handle rollbacks
- It is difficult to predict how many pieces of existing software won't properly co-exist with the rollback infrastructure

12 IMPLEMENTING THE PACKAGE MANAGER USING EXISTING TOOLS

Interest has been expressed in implementing the Apertis package manager using an existing package manager as a back-end. Collabora has considered this approach and believes it to be more difficult than creating a simplified new package manager exclusively for applications. The details are included for completeness.

Since the Debian APT package manager is already installed on the system for managing system updates, it would be the obvious starting point to pursue such an approach..

This chapter attempts to give an insight into what is provided by the Debian package management tools, and how they could be used to build a package management framework supporting a subset of the concepts described in this document.

The functionality provided by the existing package management tools is:

- dpkg – the back end package installation/removal tool.
- APT – the front end package downloader and dependency resolver

APT supplies a collection of command line tools that can search for packages based on their description, display information about specific packages, or download a package and all the packages it depends on for passing to dpkg for installation. Downloading and upgrading of installed packages is also a feature of the APT tools.

The dpkg tools take package files and extract their contents to the appropriate directories, provide lists of installed packages, and perform a variety of system maintenance functions.

While some of the APT/dpkg functionality is not useful for application management (the dependency resolution features, for example), APT can provide an update system, and dpkg can provide a package installation system, provided some of the concepts of this document are simplified.

There are many ways APT and dpkg could be combined to perform application management on Apertis. The following sections describe one possible mapping.

12.1 DPKG FOR APPLICATION MANAGEMENT

While Dpkg is theoretically capable of having two disjoint instances co-exist on a single operating system, this functionality is not well tested. It is possible that trying to do this would expose previously unnoticed bugs.

Dpkg doesn't have any special knowledge of BTRFS subvolumes - it simply decompresses archives into directories. In order to have the Apertis application rollback system work, the subvolumes for application storage would have to be created prior to this decompression.

Since dpkg allows scripts supplied as part of an archive's packaging to be run

before and after an installation, these scripts can be used to generate appropriate subvolumes before installation and delete them post removal.

However, while this may seem like a reasonable use of the packaging capabilities, it will make making any changes to the way applications are handled difficult, as the application storage logic would be embedded in every package individually, and it is impossible to change packages that have already been downloaded by a user.

In order to future-proof the implementation, the application manager would be required to perform any subvolume related steps. The safest way to proceed is to have every application package assume it will be installing into the previously described installer-temp subvolume. After the installation is complete, the application manager will appropriately rename the subvolume.

To perform a removal, the application's subvolume should first be renamed to installer-temp, so dpkg can perform its delete operations – which must exactly match the filenames it created on installation. Dpkg will delete every file individually, which may take significantly longer than just removing a subvolume.

Due to the amount of work that must be done external to dpkg, it is providing little more functionality than “tar” - and makes the application removal procedure slower than a simple subvolume removal. However, if APT is to be used as a front end, dpkg will need to be used as a back-end.

The revised installation procedure would be:

1. A new application subvolume named `installer-temp` will be created. If this subvolume already exists it will be deleted.
2. The application dpkg file will be installed with `dpkg -i`. This will result in its installation into the `installer-temp` subvolume.
3. The application specific metadata will be configured.
4. The `installer-temp` subvolume will be renamed based on its store ID from the manifest and mounted in the application directory (Listing 2: Layout of the application directories).

While this may look like a shorter list of steps than previously enumerated, note that it does not in any way verify that the package is legitimate, and does not provide a temporary icon for installation. Package validation in a standard APT+dpkg setup relies on APT to verify that packages are legitimate.

If those features are still desired, they must be implemented in another way. For example, the package file could be distributed in a tar file with a gpg signature and an icon. But then the application manager would have to extract those pieces before passing the package on to dpkg for installation.

12.2 APT FOR UPDATES

Even if APT's package resolving features aren't useful for Apertis, its ability to download and update packages could be used for keeping applications up to date. Instead of the application store tracking installed applications, all store

applications would be placed in an apt repository.

The applications repository would need to be separate from the system repository, since the system directories are read only and require a special upgrade procedure. If the repositories were combined, an update operation might attempt to change both applications and read only system components.

Since each application package needs to be installed into the same subvolume (installer-temp), the APT tools must be used to download but not install packages, and the resulting packages must be handled individually using the following procedure:

1. If the application is running, it will be closed. Launching the application will be disallowed for the duration of the update.
2. The application specific metadata will be removed.
3. The application's subvolume will be unmounted.
4. A new snapshot named `installer-temp` will be created from the contents of the application's subvolume. If the subvolume exists, it will be deleted.
5. “`dpkg -i`” will be called with the package file for this application
6. The `installer-temp` subvolume will be renamed and mounted in the application storage directory.
7. Only one prior version is to be stored for rollbacks, so If an older version was being saved it will be deleted.
8. The application specific metadata will be configured.
9. Launching the application will be allowed and the agent, if present, will be started.

APT can be configured to verify archives when they are downloaded, so only packages signed by the application store will be installed. It should be noted, though, that some mechanism should be in place to restrict what packages a device can download on the server distributing packages. If a user ever managed to get a root shell on a device, they would be able to download any application from the store and distribute them.

13 REFERENCES

This document references the following external sources of information:

- Wikipedia article on JSON: http://en.wikipedia.org/wiki/JSON
- 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)