# Concept document

## GENIVI Application Framework

7th July 2016


Document Author: Simon McVittie

Document Reviewer: Sjoerd Simons
Document Reviewer: Philip Withnall
Document Reviewer: Guy Lunardi

Document Owner: Collabora - Guy Lunardi
E-mail: guy.lunardi@collabora.com
Tel: : +1-801-200-3848



Collabora Document Number: GEN0001

Collabora Document Version: v0.3 (draft)


| Author | Date | Version | Change | Approved |
|---|---|---|---|---|
| SM | 29-06-2016 | 0.1 | Initial version | SM |
| SM | 04-07-2016 | 0.2 | Revised internal version | SS |
| GL | 23-06-2016 | 0.3 | First draft shared with GENIVI | GL |

# Application Framework scope and requirements

This document outlines requirements relevant to the GENIVI Application Framework effort. However some of these requirements may well be considered out of scope for requirements to the GENIVI Application Framework due to overlap with other GENIVI initiatives. They are included here as they are perceived to be within the context of an application framework.

This document does not aim to specify a particular implementation for any requirement.

The terms *privilege*, *privilege boundary*, *confidentiality*, *integrity* and *availability* have their usual information-security meanings (for definitions, please refer to Apertis Security design).



## Table of Contents

# What's in an app

There are two commonly-used definitions of an "app": either a user-facing launchable program (an *entry point*) such as would appear in launcher menus, or a user-installable package or bundle such as would appear in an app store.

A user-installable bundle would most commonly have exactly one entry point. However, it might not have any entry points at all, for example if it is a theme or some other extension for the operating system. Conversely, it might have more than one entry point: for example, a user-installable bundle for an audio player might contain separate menu items for music and audiobooks, launching different executables or the same executable in different modes to provide an appropriate UX for each use-case.

In this document, when we need to distinguish between the two meanings, we will say that a user-installable *bundle* contains zero or more *entry points*. Entry points are similar in scope to Android activities.

Some vendors might decide to restrict the apps available in their app stores to have at most one entry point, but that is a policy decision by those vendors and should not be reflected in the more general app framework.

Entry points might be written as native code (for example compiled from C or C++), or they might run under an interpreter or JIT in a runtime environment that provides GUI functionality analogous to native code (for example if the app is written in Java, Python, or JavaScript for the node.js, gjs or seed runtime environments), or they might run in a HTML5 runtime environment. We treat all of these as fundamentally similar: they result in the execution of app-author-chosen code.

(Note that whether an app is written in native code has no bearing on whether it is what GENIVI calls a *native application*, which is an app that is built into the platform, or a *managed application*, which is one of the user-installable apps discussed here: either may be written in either native code or an interpreted/JITted environment.)

- The app framework must be capable of running native-code (C or C++) executables.
- The app framework must be capable of running programs that require an interpreter/JIT-based runtime environment such as Java or Python. It may require that the runtime environment provides suitable library functionality to work with the framework (for example, if the framework uses D-Bus for IPC, then it does not need to support runtime environments that do not have a D-Bus implementation or binding).
- The app framework must be capable of running programs that run in a HTML5 runtime environment: in other words, it must be possible to package a web application into a form suitable to be an app bundle.

The entry points to an app might include GUIs and/or background services (agents, daemons).

- It must be possible for an app to contain zero or more GUI entry points. Each of these

might appear in menus (see App launching) and/or be available for launching by other means (see Document launching, URI launching, Data sharing).

- It must be possible for an app to contain zero or more background services with no GUI, which can be launched for purposes such as Data sharing. For example, a search provider for a global search feature similar to GNOME Shell search or Unity Lenses, such as the one described in Apertis Global Search design, might be implemented in this way.
- It must be possible for the GUIs and background services to be implemented by the same executable(s) run with different options, or by separate executables.

  Some vendors might decide to restrict the apps available in their app stores to have at most one executable, or to have at most one GUI and one non-GUI executable, but that is a policy decision by those vendors and should not be reflected in the more general app framework.

Each bundle should have *bundle metadata* to represent the app in situations like an app store, a system settings GUI or a prompt requesting app permissions.

- As a minimum, this metadata should include a globally unique identifier, an icon, and an international (English) name and description.
- Additionally, app bundles should be able to contain translations (*localization*) which replace the international name and description, and any other fields that are marked as translatable (*internationalization*), when displayed on devices configured for a specific language and/or country.
- The metadata fields in an entry point should be in line with what is typically present in other interoperable package metadata specifications such as freedesktop.org AppStream and the parts of Android manifests that do not relate to a specific <activity>.
- The base set of metadata fields should be standardized, in the sense that they are described in a vendor-neutral document shared by all GENIVI vendors and potentially also by non-GENIVI projects, with meanings that do not vary between vendors. For example, AppStream XML would be a suitable implementation.
- We anticipate that vendors will wish to introduce non-standardized metadata, either as a prototype for future standardization or to support vendor-specific additional requirements. It must be possible to include new metadata fields in an entry point, without coordination with a central authority.
- For example, this could be achieved by namespacing new metadata fields using a DNS name (as is done in D-Bus), namespacing them with a URI (as is done in XML), or using the X-Vendor-NewMetadataField convention (as is done in email headers, HTTP headers and freedesktop.org .desktop files).

Apps are expected to be numerous.

- The app framework must be designed such that it does not need to place an arbitrary limit on the number of apps installed on the system, as long as their total size on storage (flash) fits within the available space.
- The app framework must be designed such that it does not need to place an arbitrary limit on the number of apps running at the same time, as long as their total size in RAM fits within the available space.

## Data management

The app framework must provide a location where app programs can write their private data.

**Open question**: is this in-scope for the app framework, or is there some other platform component that does it?

The framework should provide a location that is treated as private data in which to store *cached data*, defined as data that can be recovered in a straightforward way by downloading it from the Internet or computing it from non-cached data.

- The framework may delete files from the cached data area at any time to free up storage space, and apps should be written to expect this.
- For app author convenience, the framework may also provide conventional locations for other sub-categories of private data such as *configuration* (data that has a useful default, but can be reconfigured by the user, and whose deletion would be considered to be data loss) and *state* (data with no useful default, whose deletion would likewise be considered to be data loss).

The app framework must provide a mechanism by which an app program's private data can all be deleted by another system component, for example as part of removal or a factory reset.

The app framework should provide a mechanism by which all app programs' private data can be deleted in a single operation during a factory reset, so that the factory reset procedure does not need to enumerate app programs and iterate through them.

Deleting per-user data and per-device data during a factory reset is also anticipated to be necessary, but is outside the scope of this framework.

# Sandboxing and security

App processes should run in a sandbox which partially isolates them from the rest of the system.

We anticipate that each app bundle will act as a security domain, similar to the concept of an *origin* on the Web: in other words, there is a security boundary between each pair of app-bundles, but for simplicity there is no privilege boundary within an app bundle (for example between two programs in the same app bundle).

Each app is assumed to store *private data* which is specific to that app. On a multi-user system, this private data is also specific to a user: in other words, there is one private data location per (app, user) pair.

- Any data with this access model is considered to be private data, whether it is in files directly written by the app, files written by platform libraries used by the app, or other data stored on behalf of the app by platform services (for example accessed via inter-process communication).
- *Private data availability*: when a specific user runs a program that is part of a specific app, that program can read and write the data owned by that (app, user) pair.
- *Private data confidentiality and integrity*: an app must not be able to read, add, change or delete data owned by a different app and the same user without the other app specifically sharing it. The program must also not be able to read, add, change or delete data owned by the same app but a different user.

  Note that the App confidentiality requirement below imposes a stronger requirement than this: the first app must not even be able to know that the second app's private data *exists*.

Some categories of data might be specific to a single app but common to all users. We call these *per-app data*.

- The app framework may have support for per-app data. If it does, the availability, confidentiality and integrity requirements are analogous to those for private data. The per-app data is considered to be jointly owned by all users, therefore there is no expectation of confidentiality or integrity for the per-app data of programs from the same app bundle running as different users.

Some categories of data are not necessarily specific to a single app; instead, they might be shared between all apps. We call these *per-user data*. For example, the user's address book (contacts) and the user's calendar (appointments) might be among these categories.

- Any data with this access model is considered to be per-user data, whether it is in files directly written by multiple apps, files written by platform libraries used by multiple apps, or other data stored on behalf of multiple apps by platform services (for example accessed via inter-process communication).

- We anticipate that in practice, per-user data would most commonly be kept outside apps' sandboxes and accessed via inter-process communication to a shared service. For example, Android contacts provider services, GNOME evolution-data-server and KDE Akonadi all use this model for address books.
- *User data availability* (read): the apps that require access to this per-user data must be able to read it. For example, a messaging application might require access to the address book so that it can read the thumbnail photos representing contacts and display them in its user interface.
- *User data availability* (write): the apps that require write access to this per-user data must be able to add, change and delete it. For example, a messaging application might require write access to the address book so that it can add contacts' instant messaging addresses to it.
- *User data confidentiality* with least-privilege: an app must not be able to read per-user data without user consent, other than what that app needs to carry out its normal function. For example, a compromised messaging app would still be able to read the address book until the compromise was somehow detected, but would not be able to read (for example) the user's appointments calendar.
- *User data integrity* with least-privilege: an app must not be able to modify per-user data without user consent, other than what that app needs to carry out its normal function. For example, a compromised messaging app would still be able to modify the address book until the compromise was somehow detected, but would not be able to modify the user's appointments calendar.

Some categories of data are not necessarily specific to a single app or to a single user; instead, they might be shared between all apps and between all users, like Android's /sdcard. We call these *per-device data*.

- The app framework may have support for per-device data. If it does, the availability, confidentiality and integrity requirements are analogous to those for per-user data, except that there is no expectation of confidentiality or integrity for per-device data.

The user might install a *malicious app* that has been written or modified by an attacker, or the user might install an app with a security flaw that leads to an attacker being able to gain control over that app (referred to below as a *compromised app*). Either way, the attacker is assumed to be able to execute arbitrary code in the context of that specific app.

- The requirements stated above for private and user data confidentiality and integrity mitigate this attack by restricting what the malicious or compromised app can do.
- *App integrity*: a malicious or compromised app app must not be able to modify the executables and static data of other apps.
- *App confidentiality*: in general, a malicious or compromised app must not be able to list the other apps that are running on the system or the other apps that are installed, either by their bundle names, by their entry points, or by inferring their

presence from private or per-app data that they have written. Both are potentially sensitive information that could be used to "fingerprint" a particular user or class of users (for example customers or employees of a particular organization).

- Note that if an app has written per-user data or per-device data, then it has potentially given up its own app confidentiality, in the sense that a malicious or compromised app could potentially identify it from the per-user or per-device data that it has written out. We recommend minimizing the number of apps able to write per-user and per-device data for this reason, and preferring to use content selection, document launching and data sharing to satisfy the use-cases for which other platforms would use a per-device filesystem.
- Similarly, in general an app must not be able to communicate with other apps without user consent. Controlled exceptions to this general rule might exist for use cases such as data sharing.
- *System integrity*: a malicious or compromised app app must not be able to violate the integrity of the system as a whole (for example by modifying the executables or static data of the system, or by altering the system's idea of what is a trusted app source).

*Resource limits*: A malicious, compromised or buggy app might use more than its fair share of system resources, including CPU cycles, RAM, storage (flash) or network bandwidth.

- Each app must have its own limit for these various metrics, for example by using cgroup resource controllers.
- If this limit is exceeded, the vendor may choose how to respond to this. Options include killing or freezing the app, rate-limiting requests, denying requests, and/or reporting the app to the app-store as potentially malicious.

# App permissions

A very simple app, for example a calculator or a simple to-do list, might not need to do anything other than the operations allowed to all apps: display a GUI when launched, run code in a sandbox, store its own private data up to some reasonable limit, and so on.

To carry out its designed purpose, a more complex app might need permission to carry out actions that can compromise confidentiality (user privacy), integrity, or availability (the absence of denial-of-service). For example, a more elaborate to-do list app might be able to synchronize the to-do list to a cloud service, requiring it to have Internet access which would make it technically able to copy whatever data it can read to a location not under the user's control; it might ask to read the user's geographical location, to provide location-based reminders; and it might support attaching photos to its to-do items, requiring it to read files that are not its private data.

Some permissions have technical constraints that makes it impractical to request user permission before they are used. For example, one possible permission flag is "has unrestricted Internet access", which might be used for a voice-over-IP client app. To support this control, the life-cycle manager would need to launch the app program with unrestricted Internet access either allowed or forbidden: it cannot be adjusted later.

- App bundles must be able to specify permissions without which they will not work, given in bundle metadata.
- The user might be asked whether to grant those permissions on installing that app bundle or on launching any entry point from that bundle, or the framework might automatically grant certain permissions based on approval from an app-store curator without user interaction.

Some permissions can usefully be granted or denied at runtime. For example, address book access on Android works like this: the permissions framework can be configured to prompt the user on each attempted access.

- Operations that cross a privilege boundary between processes should include a step where a platform security framework is queried, to check whether the user's permission for the privileged action has been given. This should have at least three possible policy outcomes: allow, deny, or ask the user.

Some operations that cross privilege boundaries naturally include an opportunity for the user to reject the operation. To minimize driver distraction, the system should provide that opportunity instead of having a separate permission prompt.

- If an operation will naturally result in the user being prompted for a decision of some sort, there should not be an additional prompt for whether to allow the action. Instead, the user can indicate lack of consent by declining to make the requested decision. For example, content selection could use this approach: the user implicitly indicates consent to open or attach a file by selecting it, or indicates lack of consent by cancelling the file-selection dialog.

- The framework might require that particular privilege-boundary-crossing operations are declared in advance even though they imply an opportunity for the user to reject the operation, for example if those operations are considered to be particularly sensitive or vulnerable to social engineering attacks. If it does, then it may make attempts to invoke those operations fail unconditionally, as if the user had canceled them but without prompting the user at all.
- Operations that cost money might be considered to be particularly sensitive — for example, a parent installing apps on behalf of a child is likely to want to prevent them — so the framework implementor might wish to ensure that operations like "send SMS" and "make in-app purchases" must be declared in advance.
- Access to online accounts (such as social media) might be considered particularly susceptible to social engineering (since a user might not recognize when a request to fill in their social media account/password is or isn't legitimate), so the framework implementor might wish to ensure that operations involving these accounts must be declared in advance.

# App launching

A bundle may contain zero or more entry points. These are typically started from a *launcher*, which might take the form of a home screen, main menu or application list.

- A launcher must be able to list all of the visible, available entry points in any installed bundle, together with enough metadata to display them in its menus. As a minimum, this would typically include a multilingual/localized name and an icon. Other metadata fields, such as categories, could be useful or unnecessary depending on the launcher's UX.
- The metadata fields in an entry point should be in line with what is typically present in other interoperable menu-entry specifications, such as freedesktop.org .desktop files or the <activity> element in Android manifests.
- The base set of metadata fields should be standardized, in the sense that they are described in a vendor-neutral document shared by all GENIVI vendors and potentially also by non-GENIVI projects, with meanings that do not vary between vendors. For example, .desktop files would be a suitable implementation.
- We anticipate that vendors will wish to introduce non-standardized metadata, either as a prototype for future standardization or to support vendor-specific additional requirements. It must be possible to include new metadata fields in an entry point, without coordination with a central authority.

  For example, this could be achieved by namespacing new metadata fields using a DNS name (as is done in D-Bus), namespacing them with a URI (as is done in XML), or using the X-Vendor-NewMetadataField convention (as is done in email headers, HTTP headers and freedesktop.org .desktop files).
- Because of the requirement that ordinary app bundles are not allowed to enumerate other app bundles or entry points, if a launcher is implemented as a user-installable app bundle (as is sometimes done on Android), it must have a special permissions flag allowing it to carry out that restricted action.

Some entry points might be flagged to not be visible in menus. For example, an app that is a viewer for some file type such as PDF might register itself as a handler for files of that type, but might not have anything useful to do if it appears in menus otherwise.

- Entry point metadata must indicate whether the entry point is to be visible in menus.
- The mechanism used by the launcher to list entry points may either include or exclude invisible entry points. If it does include those entry points, it must also provide the launcher with an indication that they are to be made invisible.

When the user selects an entry point, the expectation is that the program that implements that entry point should be launched.

- If the program that implements the entry point is not already running, the system

must run it. (See also life-cycle management.)

- The program might implement more than one entry point. It must be told which entry point was launched, for example via command-line arguments or an inter-process communication call.

We do not anticipate that ordinary (non-launcher) app bundles would have a reason to launch specific entry points in this way: we expect that if app bundles need to communicate, they will do so via document launching, URI launching or data sharing. This does not preclude one executable in a bundle from running another executable in the same bundle directly.

- **Open question**: Do ordinary app bundles need to be allowed to launch other bundles' entry points by name? If so, why?
- Android does allow this, but Android does not appear to provide app confidentiality.
- One possible use-case for a program launching a program outside its bundle would be to bring up the system settings. For example, Android apps that make use of location services often have a shortcut button to bring up the Location panel in the built-in Settings app, because the user-installable app would not be able to enable location itself, but its author wishes to make it easy for the user to do so.

  However, a vendor-specific Settings app is part of the platform rather than being a user-installable app bundle, so the constraints applying to it and the APIs that can be used with it do not have to be the same as for app bundles.

  This would also be easy to implement without launching the Settings app by name: the built-in Settings app could register for URI launching as the launcher of a URI scheme, similar to the way the iOS Settings app used to register the prefs URI scheme, and the user-installable app could launch a URI of that scheme.

# Document launching

Some app entry points will provide handlers for particular file types.

- An entry point must be able to identify the file types that it can receive. For example, a document viewer might register itself to receive Microsoft Word documents, Open Document Text files, and PDFs.
- We recommend that these are identified via IETF media types (also known as content types or MIME types), because the IETF media types are an extensible standard, are ubiquitous in existing operating system environments such as Windows, OS X, Android and freedesktop-based environments such as GNOME, and are part of key Internet technologies such as HTTP and email.
- The app framework must be able to identify the format of a file on secondary storage (flash), for example via its extension or "magic number". Unidentified files must be considered to have a documented generic format, for example application/octet-stream in the IETF media type system.
- **Open question**: it has been suggested that app-bundles should be able to define their own new file types. Is this a requirement?
- This requirement seems unwise from the point of view of system integrity: if an app-bundle can define its own file types with their own extensions and/or "magic numbers", then it can introduce a conflict with other app-bundles or even alter the interpretation of existing files.

  If this is implemented at all, we recommend that it should be tightly controlled by app-store curators.
- *Choice of document handler*: When a file is activated (for example by tapping its icon) from a non-app context such as the home screen, the app framework must locate the entry points that are able to handle that file. It must either choose one of those entry points for use, or prompt the user to choose one.
- When a file is activated from the context of an app (the *initiating app*), for example if the user activates an attachment in an email app, the app framework must behave similarly. It may opt to follow a different policy for choosing the correct entry point in this case; for example, it might prompt the user for confirmation even if there is only one possible handler.
- System vendors must be able to force a particular app to handle particular file types. For example, a vendor might wish to make their video player handle all videos.
- If no handler is available for the selected file type, the app framework should arrange for a suitable fallback to be displayed. For example, it might show an error message, or it might launch its app store user interface with a search query for the handlers for that file type.

- *No feedback to initiator*: It should do this itself or by interacting with other system components instead of feeding back an error code to the initiating app (if any), because otherwise the initiating app would be able to use this as an "oracle" to gather information about the set of installed app bundles.
- *User confirmation*: If exactly one handler is available for the selected file type, the app framework may launch it directly, or ask the user for confirmation. If the user cancels a request for confirmation, the app framework should neither launch the handler nor feed back an error code to the initiating app.
- If more than one one handler is available for the selected file type, the app framework may launch a preferred handler directly, or ask the user to make the choice. If the user cancels a request for app choice, the app framework should neither launch a handler nor feed back an error code to the initiating app.
- The app framework must arrange for the file's content to be made available in a location where the chosen app can read it (see sandboxing and security).
- If the program that implements the entry point is not already running, the system must run it. (See also life-cycle management.)
- The program must be told that it was launched to open a file, and given the filename of the file to open, for example via command-line arguments or an inter-process communication call. The filename that it is given might differ from the original file that was activated, for example if the file had to be copied or linked across a privilege boundary to be made available in the program's sandbox. The program must be able to distinguish between this action and ordinary app launching.
- Programs should be careful not to treat documents received in this way as executable code, or assume that the source of the document is trustworthy. For example, macro languages in "office" document formats should be disabled, and if arbitrary code execution in a program can be triggered by a malformed document, this should be considered to be a security vulnerability.
- We do not anticipate a need for the initiating app to be able to influence the choice of launched app.
- If the initiating app could influence the choice of launched app, a malicious app could potentially use this to break or undermine app confidentiality. For example, suppose org.example.Secret opens .secret files. If the app com.example.Spy wanted to determine whether org.example.Secret was installed, it could register an entry point com.example.Spy.SecretHandler which also opens .secret files, create a .secret document, and launch that document specifying org.example.Secret and com.example.Spy.SecretHandler (in that order) as the preferred handlers. If com.example.Spy.SecretHandler was launched, then com.example.Spy could be sure that org.example.Secret was not installed. Conversely, if com.example.Spy.SecretHandler was not launched, then com.example.Spy could infer that org.example.Secret was likely to be installed.

Apertis Content Hand-over Use Cases contains some similar requirements-capture that was carried out for the Apertis platform.

## URI launching

Some app entry points will provide handlers for particular URI schemes such as https, mailto or skype.

- file URIs must not be included in this mechanism. Instead, they should be decoded into filenames and processed via document launching.
- An entry point must be able to identify the URI schemes that it can receive. For example, a multi-protocol voice-over-IP client might support receiving sip and xmpp URIs.
- When a URI is activated, the app framework must locate the entry points that are able to handle that URI and choose one for launching, much like file type handling. The same points about choice of handler, user confirmation, and lack of feedback to the initiating app apply equally here.
- As with URI schemes, system vendors must be able to force a particular app to handle particular URIs. For example, a vendor might wish to make their built-in web browser handle all http and https URIs.
- If the program that implements the entry point is not already running, the system must run it. (See also life-cycle management.)
- The program must be told that it was launched to open a URI, and given the URI to open, for example via command-line arguments or an inter-process communication call. The program must be able to distinguish between this action, document launching and ordinary app launching.
- As with document launching, we do not anticipate a need for the initiating app to be able to influence the choice of launched app, but system components might need to do so.
- Programs should be careful not to interpret URIs in a way that a malicious or compromised initiating app could use to violate integrity, confidentiality or availability. For example, telephone calls and text messages (SMS) could cost money, distract the driver, or divulge sensitive information to a third party. As a result, an app that acts as a tel: URI handler may respond to URI launching by offering the user a choice of actions to carry out (for example "call" and "send SMS" buttons, perhaps with a text input widget pre-filled with SMS text taken from the URI), but must not actually initiate the call or send the SMS until the user requests it.

  Similarly, if a URI scheme is designed in such a way that dereferencing a URI can cause content to be modified or deleted (an unsafe request in HTTP terminology), then the program interpreting the URI should ask the user before proceeding.

Apertis Content Hand-over Use Cases contains some related requirements-capture that was carried out for the Apertis platform.

## Content selection

App programs might wish to interact with data stored in locations that are not naturally accessible to the app. For example, an attachment to an email would be private data for the email app as run by the user whose email account is accessing it.

However, we would like to avoid such data passing through a per-device data storage area that is shared between all apps (similar to Android's /sdcard), because in practice data passed between programs will typically include sensitive data such as photos and documents.

The solution that is used in Apple's iOS and planned for the Flatpak system is to have an API call that creates a file-opening or file-saving dialog. While visually presented as if it was part of the requesting app, this dialog actually exists outside the app's security context (it is privileged), and it is able to browse all of the user's files. iOS calls this the Document Picker, while Flatpak calls it the Document Portal.

- The app framework should provide a way to ask the user to browse for a file to open for reading, similar in principle to the conventional "Open" dialog on desktop operating systems.
- If the user does so, the framework must make this file available to the app program for reading.
- If the user cancels this prompt, the framework must indicate this to the requesting app, and must not grant it any additional access to any files.
- The app framework should provide a way to ask the user to browse to a location in which to write a file, and simultaneously choose a name for that file.
- As above, depending on the user's choice, the framework must either provide a way for the app to write to that location and name, or indicate cancellation and not provide any additional access.
- If the user selects an existing file outside the app's sandbox, it must be overwritten atomically if the underlying filesystem supports that.
- The app framework may provide specialized versions of this functionality for specific file types, in particular images/photos.

# Data sharing

The system might require the ability to enumerate the implementations of a particular service or set of functionality. In this document we will refer to that set of functionality as an *interface*. One use-case for this is that a global search facility within the platform needs to discover a list of background services (entry points) within app bundles that can provide search results in response to user queries entered into some global search UI; for example, a Spotify client could use the search term to match artists or songs.

- Suitably privileged components of the system must be able to enumerate the implementations of an interface.
- Suitably privileged components of the system must be able to communicate with the implementations of an interface.
- If the system initiates communication with an implementation of an interface that is not already running, the app framework must arrange for the implementation (an entry point) to be started.

An app might also require the ability to enumerate the implementations of a particular interface. One example use-case here is that if an app will display a Sharing menu similar to the UX seen in Android, it needs to be able to list the apps with which files or data can be shared, in order to populate that menu. Due to the app confidentiality requirement, this should only be allowed if the interface in question is one whose implementors are aware that it will result in other apps being able to enumerate their apps. In this document we will refer to this as a *public interface*.

- An app with appropriate app permissions must be able to enumerate the implementors of a public interface.
- Depending on the system and the interface in question, a special permission flag per public interface might be required to list the implementors, or that information might be available to every application.
- An app with appropriate app permissions must be able to communicate with all of the implementors of a public interface, for example via an inter-process communication channel such as D-Bus.
- If an app initiates communication with an implementation of an interface that is not already running, the app framework must arrange for the implementation (an entry point) to be started.

The Apertis Interface Discovery design and Apertis Data Sharing design describe use-cases, requirements and proposed implementations for this topic in the Apertis system.

# Sharing menu

One specific use-case for data sharing is a menu for sharing content with other users, for example via social media, email or real-time communications, similar to the Android Sharing menu.

Two possible UXs for this facility are presented in the Apertis Sharing design. Each UX motivates rather different requirements for how this facility interacts with apps, and in particular its impact on app confidentiality.

**Open question**: Is this in the scope of the application framework? If it is, which UX do we intend to support?

# Life-cycle management

Under various circumstances (including those described in app launching, document launching, URI launching and data sharing), the system must be able to start a program provided by an app bundle.

This topic overlaps with the functionality of the GENIVI Node Startup Controller, and more generally the GENIVI Lifecycle cluster. It should potentially be considered to be an orthogonal topic outside the scope of the App Framework design. Some requirements in this area are outlined here in the hope that they can be used to clarify the division of responsibilities.

The possible states of a program in an app are as follows:

- Not installed
- Inactive (installed but not running)
- Running
- Paused

The valid state transitions move linearly through that list in single steps, as follows:

- Not installed → inactive: install app bundle
- Inactive → running: start (launch), see this section
- Running → paused: pause, see this section
- Paused → running: unpause, see this section
- Running → inactive: stop (kill, terminate), see this section
- Inactive → not installed: remove app bundle

Transitions do not skip a step: for example, a paused app process cannot be stopped without first unpausing it, and an app bundle cannot be removed until all of its processes have been stopped.

**Open question**: some GENIVI documents have the concept of "activating" a program, which appears to be distinct from launching it. Does this correspond to selection, similar to single-clicking an icon in a desktop environment where double-clicking would cause

launching; or does it represent a transition away from an intermediate state where a newly installed app is unavailable until an activation, enabling or licensing step has been performed, similar to the concept of activating a Windows installation; or is it something else?

As a prerequisite for sandboxing and security, app processes must be identifiable.

- The app framework must be able to start processes, either directly or by asking a separate service manager such as the Node Startup Controller to start them.
- *Process tagging*: each process executing code from an app bundle must be marked with the unique identifier of that bundle (for example by placing it in a suitably named cgroup or by running it under a suitable LSM context).

  Those processes and their child processes, whether running the same or a different executable from the app bundle or running an executable provided by the system, must not be able to enter a state where they are no longer identifiable as belonging to their bundle.
- Depending on the vendor's UX design and the app author's UX design, the entry point might start in a default state, or it might start by restoring the last-used context. The app framework should be able to send a hint that indicates which of these modes is preferred (see the section on Last-used context).

The application launch has various interactions with the graphical user interface. See Apertis Compositor Security design for more detailed requirements-capture for the interaction between the GUI shell and apps. The Apertis design assumes that the compositor and the GUI shell are combined, as was done in Apertis' Mildenhall reference UI and in GNOME's GNOME Shell. In a system where the GUI shell and compositor are separate, those requirements should be read as being requirements for the combined system consisting of the GUI shell and the compositor.

- Processes may request that windows (surfaces, layers) are displayed. The GUI shell must be able to identify the app bundle to which a window belongs, so that it can instruct the compositor (layer manager) to display it (or not display it) according to its UX policy.
- The GUI shell must be able to identify which windows belong to the same user-facing app, so that they can be associated visually, and so that it can prevent apps from setting up misleading situations like a dialog from one app drawn over another app's window.
- The GUI shell might have an application-switcher similar to the one in Android. It must be possible to mark each app's collection of windows with a name and icon as is done in Android. This is important for the integrity of the UX — otherwise, it would be impossible for the user to tell which app is producing a given window, for example to see which app is responsible for an advertising popup (*output integrity*), or which app is requesting entry of a password (*input integrity*).

- If application launching is in progress but no window has been displayed yet, the framework must avoid focus stealing: in other words, it must ensure that input intended to go to the previous foreground window in a particular screen area is not inadvertently directed to a window presented by the newly launched application.
- One possible implementation is to disable input, send the previous app to the background, or display a placeholder while waiting for a launched app to become available, so that the app cannot appear while the user is halfway through another interaction with the previous app.
- Another possible implementation is to track whether user continues to interact with the previous app, and if they do, keep the previous app in the foreground and place the newly launched app's window in the background when it appears.

To improve perceived responsiveness, the GUI shell might display an indication that a particular entry point or app is starting.

- Startup notification (successful case): the GUI shell must be notified by the life-cycle manager when a particular entry point is starting. It must also be notified when the entry point becomes available, either explicitly (another notification from the life-cycle manager) or implicitly (a window is displayed by the appropriate app-bundle with the entry point's identifier as metadata) so that it can withdraw the indication.
- To meet the app confidentiality requirement, these notifications must not be visible to other apps.
- Startup notification (unsuccessful case): the GUI shell should be notified by the life-cycle manager when an attempt to start a particular entry point fails, so that it can withdraw the indication and display a warning instead.
- To meet the app confidentiality requirement, these notifications must not be visible to other apps.

If an app program crashes or otherwise exits unexpectedly, the system might restart it.

- This must be rate-limited, to avoid infinite restart loops that could consume disproportionately many CPU cycles. For example, apps might be configured such that more than *n* restarts within *t* seconds will cause further attempts to restart the app to be abandoned. For responsiveness, we recommend that the restart counter and time are reset when the user specifically launches an entry point.

An app program might have costly graphical processing which its author wants it to stop doing while not visible.

**Open question**: Are these requirements regarding visibility applicable to the application framework, or to life-cycle management, or are they in the scope of the compositor or the combined system consisting of the compositor and GUI shell?

- The app framework should send a notification to the app program at each transition from one or more windows visible to no windows visible, telling it that it has been

moved to the background (become invisible).

- The app may still paint its window(s) while in the background. Their new contents must be used in any context where the app's windows would briefly become visible, for example as thumbnails in an app-chooser.
- The app framework should send a notification to the app program before each transition from no windows visible to one or more windows fully visible, telling it that it has been moved to the foreground (become visible).
- Until the app can redraw itself, its last known window contents must be painted.

The app framework will sometimes stop apps from running, most obviously due to user request or during device shutdown. It may also stop apps if they are running in the background and there is insufficient RAM for a user-requested operation such as starting a new app, similar to the behavior of background apps in Android.

- The app framework should have a mechanism to send a request to the app process, asking it to terminate itself gracefully. (For example, systemd uses SIGTERM for the equivalent request to its managed processes.)
- A well-behaved app process should respond to this request by saving its state and terminating. The app framework must detect its termination and consider this to be a successful stop.
- The app process should update its last-used context as part of its response to this request, so that it can resume from the last-used context when started again.
- If the app process does not terminate within a reasonable time (anticipated to be limited to a few seconds), the app framework must forcibly terminate it (kill it). It must not be possible for the app process to block this forcible termination. (For example, systemd uses SIGKILL for the equivalent request to its managed processes.)
- If a stopped app is brought to the foreground, the app framework must arrange for it to be started with the last-used context.
- If the app framework needs to remove (uninstall) an app bundle that has one or more running or paused programs, it must stop those programs before commencing removal. If those programs are paused, it must unpause each one before stopping it.

If the system has a relatively large amount of RAM but a relatively slow CPU, it might be desirable to pause app processes that been sent to the background, preventing them from executing code. For example, the implementation might use SIGSTOP.

- The app framework should have a mechanism to send a request to the app process, asking it to prepare for being paused.
- The app process may respond to this request by finishing or canceling a pending operation. It should not start new operations unless they are expected to be fast.
- The app process should update its last-used context as part of its response to this

request, so that if power is lost, it can resume from the last-used context when started again.

- If the app process responds to this request, it may be paused at any time after it has sent the response.
- If the app process does not respond to this request promptly (implementation-defined, but expected to be of the order of magnitude of a few seconds), it will be paused anyway.
- If the app framework notifies an app that it will be paused, but then decides that it will not actually pause the app (for example because it is brought to the foreground), it must notify the app as though it had been unpaused.
- The app must be careful to process these notifications in-order, so that if an unpause request arrives while it is still processing a pause request, the pause request is canceled.

Paused apps can be unpaused, at which point they will continue to execute.

- If the app is brought to the foreground, the app framework must unpause it first.
- If a request is to be processed by the app process, for example for data sharing, document launching or URI launching, it must be unpaused first.
- If the app framework needs to stop an app program that is paused, it must unpause that app, then stop it.
- Whenever the app is unpaused, it must resume execution from the point at which it was paused, analogous to a laptop that has been placed in a "suspend to RAM" state. Shortly after it resumes execution, the app framework must either notify it that it has been unpaused, so that it can resume normal operation, or notify it that it is to be stopped, so that it can terminate itself gracefully.
- The app must be careful to process these notifications in-order, so that if an unpause request arrives while it is still processing a pause request (perhaps one for which the app framework timed out and paused it before it had responded), the pause request is canceled.
- Some design documents refer to the unpause operation as "restarting". We recommend avoiding that term, since it can mislead developers into believing that it refers to terminating the app, waiting for it to terminate, and starting it again, similar to systemctl restart.

Under some circumstances, other system components might forbid an app from being launched. For example, if an app is found to have a serious security vulnerability or contain malicious code, the system might mark it as forbidden.

- Other system components must be able to mark an installed app as *forbidden*. Newly forbidden apps must be stopped immediately (if running or paused), and all attempts to run them must be rejected.

    - A bundle might be marked as forbidden because it contains a serious security vulnerability.
    - A bundle might be marked as forbidden because it has been found to contain malicious code.
    - A bundle might be marked as forbidden due to conditional access.
    - **Open question**: is there a requirement that we can mark bundles or entry points as forbidden under specific operating conditions, for example at speeds over 20mph or at night?
- Whether a bundle is forbidden might be tracked per-user.
    - A parent might use a parental control interface to mark a bundle as forbidden for their child's user account, or to limit use time so that the bundle automatically becomes forbidden after 10 minutes of use per day.
- In contexts where bundles or entry points are listed (for example by a launcher), the forbidden apps must be included in the list, with metadata indicating that they are currently unavailable. This enables vendors to make a UX decision whether to display forbidden apps (for example with a desaturated icon or a "forbidden" emblem indicating that they cannot be launched), or whether to hide them from the GUI altogether.
- The system must be able to remove the *forbidden* state. After this has been done, the app may be run normally.
    - For example, if the app was forbidden due to a security vulnerability, the forbidden flag can be removed after upgrading it to a non-vulnerable version.
- There could be multiple reasons why an installed app is forbidden. It must be considered to be forbidden if at least one of those reasons is still valid.
    - For example, if the app was forbidden due to a security vulnerability and also forbidden because its conditional-access license has expired, and an update has resolved the security vulnerability, the app must still be considered to be forbidden until a new license is obtained.
- To avoid denial of service, unprivileged apps must not be able to mark apps as forbidden.

# Last-used context

The system must allow each app to store a *last-used context* that encodes its user-visible state during its most recent use.

The last-used context must be treated as private data.

- If an app does not have any particular state, a reasonable fallback implementation is that its last-used context is the same as normal app launching. The extent to which state is saved is a quality-of-implementation issue for the individual apps: if a particular app does not save its state correctly, this is not considered a flaw in the app framework, as long as the app was given an opportunity to save its state.
- **Open question**: do we want to require that the app is given the opportunity to save a snapshot of its window contents, so that they can be used by the GUI shell to represent the stopped app?

  If we do, then they must be stored in a prescribed location/format to be understood by the GUI shell, whereas the rest of the last-used context does not have any particular requirement about the structure or even location of the last-used context.

  Alternatively, this use-case could potentially be satisfied by having the GUI shell or compositor take a snapshot without the app's involvement.
- As noted in Life-cycle management, the app program should be given the opportunity to save its last-used context before it is paused or stopped.
- The app program may save its last-used context whenever its author wishes to do so. For example, a music player might save its last-used context after it starts playing each new track.
- Long-running app programs should not save last-used context at arbitrary times (for example every 10 minutes), only when a significant event has occurred.
- The app framework must be able to notify app programs that now is a good time to save last-used context.
- The app program may save its last-used context in response, but is not required to do so.
- The app program should respond to this notification. If it does not, the app framework should wait for a reasonable time (anticipated to be a few seconds) and then proceed as though it had.
- This is preferable to having long-running app programs save their state at an arbitrary time, because it gives the app framework the opportunity to influence the choice of arbitrary time. For example, the framework could notify the first app program, wait for a response, notify the second app program and so on.
- When the app is launched without any particular parameters, it must have the

opportunity to load its last-used context.

- The app framework should give the app an indication of whether it is expected to load its last-used context or not.

  **Open question**: do we expect this to be a boolean option (app should load LUC / app should not load LUC), or a tri-state (app should load LUC / app should not load LUC / app may decide)?

- Whether/when the app actually loads its last-used context is a UX decision for the platform vendor and the app vendor.
- When the app is launched for a specific purpose such as document launching or URI launching, that specific purpose takes precedence over the last-used context.
- If the app is capable of having more than one simultaneous context (for example a web browser with multiple tabs or multiple windows), the purpose for which it was launched should take precedence (for example, a tabbed web browser should load the URI from URI launching as a new foreground tab). It may additionally load its last-used context (for example, a tabbed web browser might load all the tabs from its last-used context as low-priority background tabs).
- The app framework should give the app an indication of whether, if possible, it is expected to load its last-used context in the background or not.
- Whether/when the app actually loads LUC in this case is a UX decision for the platform vendor and the app vendor. The decision made here is not necessarily the same as the decision made during launching with no particular parameters.

The app framework must also be able to store its own *last-used context*, consisting of the visible (foreground) app programs, and optionally some or all of the app programs that were running and/or paused in the background.

- On events such as a system reboot, the app framework may load its last-used context if desired. Whether to do this is a UX decision by the platform vendor. If it does:
- The foreground app programs should be run, each with its own last-used context.
- The background app programs may either be run with its last-used context, run with its last-used context and paused soon after, or left in the stopped state to be run with its last-used context later.
- The app framework may use the background app programs' last known window contents as a placeholder for their app window.

  **Open question**: is this something we want? If we do, we need either a requirement that the per-app LUC includes a snapshot of the window contents in a known location/format, or a requirement that the GUI shell or compositor can take the required snapshot.

# Download management

Management of app-initiated downloads has been suggested as a topic that is potentially in the scope of the app framework. We feel that this should probably be considered to be an orthogonal topic, to be designed separately.

The platform should provide a HTTP download manager for use by apps. The download manager may also be used by platform components, but that is outside the scope of a standard interface.

- It must be possible to have multiple downloads in parallel.
- The system may have a limit on the maximum number of downloads that will proceed in parallel. If it does, additional downloads must be held in a queue, with one additional download resuming every time an active download finishes successfully or unsuccessfully. This limit may be user-configurable.
- The system may start an arbitrary number of downloads in parallel, up to a specified bandwidth-usage limit. If it does, additional downloads must be held in a queue as above, with an additional download resuming when a heuristic indicates that there is enough bandwidth quota available. This limit may be user-configurable.
- Pending downloads must be saved periodically, and should be saved before system shutdown, so that they can be resumed automatically on next startup if the server supports it.
- Implementors should be aware that many servers do not support resuming HTTP downloads, either because they do not support the Range HTTP header properly or because an up-to-date session cookie is required.
- The list of pending downloads and their progress and pause/resume states must be treated as private data:
- Programs associated with an app bundle must be able to list, pause, resume and cancel the pending downloads that were started by that app bundle running as the same user.
- The progress of each pending download must be updated regularly. If a program from the initiating app is running, it must be able to receive progress reports on that download without polling.
- Programs associated with an app bundle must not be able to list, pause, resume or cancel the pending downloads that were started by a different app bundle.
- Programs running as a user must not be able to list, pause, resume or cancel the pending downloads that were started by a different user.
- The downloaded files themselves must be treated as private data:
- When an app requests that a file is downloaded, it must either be downloaded into the private data area for that (user, app) pair, or into a temporary location that is not

accessible by any app. When the download is completed, if it is in a temporary location, it must be moved into the private data area for that (user, app) pair.

- It must not be possible for the app to trick the download manager into overwriting data outside its private data area, for example by creating a symbolic link and having the download manager traverse that symbolic link.
- Programs associated with an app bundle must not be able to list, pause, resume or cancel the pending downloads that were started by a (non-app) platform component.
- When a download that was initiated by an app finishes (successfully or unsuccessfully), the system must arrange for one of that app's entry points to be started (if not already running), unpaused (if paused), and notified about the status of the download.

It has been suggested that the download manager should record a history of completed downloads per user, per app and/or per session.

- **Open question**: What are the use cases for this feature?
- If this is done, the user must be able to clear the history somehow. Without knowing the use cases for this history, we cannot say whether this should be functionality that is exposed to apps, or whether it should be considered to be a privileged action.

# Installation management

Management of app bundle installation has been suggested as a topic that is potentially in the scope of the app framework. We feel that this should be considered to be an orthogonal topic, in the scope of the GENIVI Software Management design. Some requirements in this area are outlined here in the hope that they can be used to clarify the division of responsibilities.

App bundles are expected to be user-installable, and may be updated on a schedule not matching the underlying platform.

- *Installation*: New app bundles can be installed, for example from an app store.
- It must be possible to install apps from removable storage media such as a USB thumb drive.
- *Upgrade*: Installed app bundles can be replaced by a newer version.
- The system should check for upgrades periodically.
- All programs from the app bundle must be stopped (see Life-cycle management) before proceeding with the upgrade. They must be blocked from running until the upgrade is complete.
- If an app was installed from removable storage media, it must remain possible to upgrade it by other means (for example using an Internet connection).
- *Rollback*: When an app bundle is upgraded, the version that was available prior to the upgrade must be saved, together with the state of its private data and per-app data at the time of the upgrade. The user must be able to roll back to the saved version at any time.
- Rollbacks are anticipated to be an unusual event, so the saved version may be compressed as a space/time trade-off, and its cached data may be deleted to minimize the storage cost.
- All programs from the app bundle must be stopped (see Life-cycle management) before proceeding with the rollback.
- Private and per-app data corresponding to the new version are not necessarily compatible with the saved version, so these must be rolled back too. Any changes made since the upgrade are lost.
- *Removal*: The user must be able to remove an installed app bundle.
- All programs from the app bundle must be stopped (see Life-cycle management) before proceeding with the removal.
- The app bundle's private data and per-app data must be removed. This matches what is done on Android, and is necessary to prevent a "masque attack" in which a user is induced to install a malicious bundle of the same machine-readable name from a

different origin (for example via social engineering), after which the malicious bundle would be able to gain access to the private and per-app data of the original bundle.

- Per-user data and per-device data must be unaffected.
- **Open question**: it has been suggested that there should be a requirement that apps must not download in parallel, with at most one app at a time actively downloading, and the rest queued.

  Is this a requirement? This seems like something that should be a quality-of-implementation decision for implementations: an implementation that expects to run on comparatively fast hardware might wish to maximize user convenience by carrying out downloads and installations in parallel, while an implementation that optimizes for implementor convenience or comparatively slow hardware might prefer to impose a limit of one download or installation at a time.

On a multi-user system, each user might wish to have a different set of apps installed. However, physically downloading and copying each app bundle for each user might be considered to be unacceptably inefficient.

- When a user installs an app bundle that is not yet physically installed, the system must carry out the actual installation.
- When a different user is active, the system should behave as if that app bundle was not physically installed: it must not be run, its entry points must not be available for launching or data sharing, and so on.
- As an exception to that general rule, privileged app management GUIs should be able to enumerate the app bundles that are physically installed, for example so that they can illustrate how storage space has been used.
- This could usefully be implemented by treating it as forbidden for the other users.
- When a user installs an app bundle that has already been physically installed by another user, the system must stop hiding the app bundle from that user. For example, it must now be made available for launching by that user, assuming there is no other reason why it would be forbidden.
- If a user has installed an app from a particular origin, then another user is not required to be able to install an app of the same name from a different origin.
- If a user has installed an app at a particular version, then another user is not required to be able to install a different version of that app.
- If a user upgrades or rolls back an app, the app may be upgraded or rolled back for all other users.
- **Open question**: do we want to mandate that the physical installation of apps must be per-device, or leave that open?

A vendor might wish to include app bundles in the original factory state of the system, while

subsequently allowing them to be upgraded and uninstalled by the user, in the same way that Google apps are typically handled on Android devices.

- *Preinstalled apps*: it must be possible to preinstall app bundles on the system, while leaving them available for installation management (upgrade, rollback, removal) in the usual way.

## Conditional access

App-store curators and app vendors might wish to provide publish apps on a time-limited basis.

This is a complex topic and we recommend that it is considered separately. The Apertis Conditional Access design has some proposed requirements for this topic.

# Appendix: mapping to GENIVI Platform Compliance Specification 10.0

- SW-APPFW-AM-001 *Manifest file for Application*: this is the bundle metadata, the app permissions, and the entry point metadata (including the details demanded by document launching and URI launching). **Open question**: Do we need an explicit statement of what else would go in here, like required API levels?

  This appears to be taking an implementation detail (the manifest file) of the motivating requirements (framework must be able to [...]) and declaring it to be a requirement in its own right. We have attempted to re-state it in terms of requirements.

- SW-APPFW-AM-002 *Support for LUC*: Last-used context

- SW-APPFW-AM-003 *Failure handling in case of application doesn't respond on state change*: Life-cycle management

- SW-APPFW-AM-004 *Launch application from another application*: this is document launching, URI launching and perhaps app launching.

- SW-APPFW-AM-005 *Factory reset*: Data management

- SW-APPFW-AM-006 *Prohibit to start an application*: see Life-cycle management and specifically Forbidden apps.

- SW-APPFW-AM-007 *Activation of application*, SW-APPFW-AM-008 *Deactivation of application*: What is activation?

- SW-APPFW-AM-009 *Support for activation of application* (sic): from its descriptive text, this seems to actually be app launching.

- SW-APPFW-AM-010 *Support for switching the application* (sic): from its descriptive text, this seems to actually mean stopping the application. Life-cycle management

- SW-APPFW-AM-011 *Support for pausing an application*: Life-cycle management

- SW-APPFW-AM-012 *Support for resuming application*: Life-cycle management

- SW-APPFW-AM-013 *Support for stopping application*: from its descriptive text, this is specifically stopping a *paused* application. Life-cycle management

- SW-APPFW-AM-014 *Application framework shall provide a mechanism to tell an application to change its state*: the states specified are START (not running), BACKGROUND (running and in background), SHOW (running and in foreground), RESTART (from its descriptive state not actually a state, and not the systemd-style restart action either, but in fact the "resume" transition from PAUSE to either SHOW or BACKGROUND), OFF (what is the

difference between this and START in terms of states?), and PAUSE (understood to be essentially SIGSTOP'ed). See Life-cycle management.

These state names demonstrate some confusion between states and state transitions. We have specifically documented states, not transitions, and provided details of the allowed transitions.

- SW-APPFW-AM-015 *Application states*: the states specified are either (INSTALLED, ACTIVATED, LAUNCHED, PAUSED) or (START, BACKGROUND, SHOW, RESTART, OFF, PAUSE) depending which column we believe. See Life-cycle management.

  It is unclear what these states mean, particularly ACTIVATED. We have described a different set of states in these requirements.

- SW-APPFW-AM-016 *Installed application info*: this is the part of app launching that deals with listing what we can launch.

- SW-APPFW-AM-017 *Access restriction for apps*: this is our sandboxing and security. It's a big topic in its own right.

- SW-APPFW-AM-018 *Support for different applications running in different runtimes*: the application framework should support JVM- or HTML5-based runtimes. Stated in What's in an app.

- SW-APPFW-AM-019 *Support for any number of applications*: stated in What's in an app, under the assumption that this is referring to lack of *arbitrary* limits. If the intention is to cope with exceeding RAM by telling excess apps to shut down gracefully, that's harder but could be done. If the intention is to cope with exceeding flash space by "swapping out" apps to cloud storage or something, that's impractical for a device that might not have constant connectivity and should not be required.

# Appendix: mapping to Suma's proposed requirements

- App-FW-001 *Protect the system against altering of any data by a malicious app*: App integrity, System integrity, Per-user data, etc.

- App-FW-002 *Protect the system against collecting and sharing of any data by a malicious app*: App confidentiality, Private data, Per-user data etc.

- App-FW-003 *Protect the system against usage of system resources* etc.: Resource limits

- App-FW-004 *An application shall not [...] interfere with [...] the other application*: App integrity, App confidentiality, Private data

- App-FW-005 *read, alter or delete non-application data*: System integrity, Per-user data.

  As written, this requirement states that this must be forbidden entirely. We have assumed that the intention was to forbid it with exceptions where necessary for the app to do its job.

- App-FW-006 *Users data are protected against access by another user*: Private data, Per-user data

- App-FW-007 *deny access to APIs to which an App has not requested permission*: Sandboxing and security

  This requirement wrongly conflates APIs with privilege boundaries. There is never any reason to deny access to APIs that do not cross a privilege boundary, because such APIs cannot do anything that the app could not do itself.

- App-FW-008 *per-app rollback*: Rollback

- App-FW-009 *Shall support applications with UI or UI less*: What's in an app

- App-FW-010 *Restore LUC*: Last-used context

- App-FW-011 *information about mime type*: Document launching

  Consideration has been given to possible ways to select file types, other than media types. We have included the recommendation that using anything other than IETF media types would be unwise.

- App-FW-012 *Resource handling*: Life-cycle management

- App-FW-013 *Inform apps about states*: Life-cycle management

- App-FW-014 *shutdown*: Life-cycle management

- App-FW-015 *Frozen state*: Life-cycle management (we're calling it "pause" in this document)
- App-FW-016 *blacklist apps*:

  We think this may be conflating two distinct behaviors. The first is to cope with apps that go into a crash loop, which must be rate-limited. The second is to have a way to stop apps executing altogether, which this document refers to as Forbidden apps.
- App-FW-017 *apps with a validity period*: Conditional access
- App-FW-018 *app requesting permissions every launch*: App permissions.

  Note that we only really recommend this for permissions where there's nothing better we can do, like "unrestricted Internet access".
- App-FW-019 *apps can communicate with other apps*: Data sharing
- App-FW-020 *Content hand-over*: Document launching, URI launching.
- App-FW-021 *content type can be opened only by...*: Document launching
- App-FW-022 *It shall be possible for an app to register a new content type*: Adding media types
- App-FW-023 *Sharing a content to be transferred out of the system*: (Android-style Sharing API): Sharing menu
- App-FW-024 *POI provider but no access to location data*: implicit in sandboxing and security and app permissions.

  This requirement appears to be conjecturing that registering an app as a points-of-interest provider would cause it to have additional permissions somehow, but whether an app is registered as a points-of-interest provider should be entirely orthogonal to whether it has the permissions that would allow it to access location data.
- App-FW-025 to App-FW-032 *Download manager*: Download management
- App-FW-032 to App-FW-036 *Internationalization*: not mentioned here.

  As Gunnar says, this is a SDK API issue, not a platform services issue. It is entirely feasible to implement internationalization through a shared library provided by the platform (part of glibc in practice) and some data files in the app (gettext .mo files) without ever crossing a security boundary, and we recommend doing exactly that.
- App-FW-037 *installation of application bundles*: Installation
- App-FW-038 *Native application*: we are unsure how this is relevant to a

GENIVI design, since the interaction between vendor-supplied native apps and the vendor-supplied platform is presumably up to the vendor.

Terminology note: GENIVI's *native applications* are the same thing as Apertis' *built-in applications*. It is nothing to do with whether the app is written in native code compiled from C/C++. GENIVI applications that are not native applications are said to be *managed applications*, which are the same as Apertis' *store applications*.

- App-FW-039 *Pre installed app vs. store downloadable apps*: Preinstalled apps
- App-FW-040a *Install app from a storage device*: Installation
- App-FW-040b *sync up with app store*: We have interpreted this to mean that after installation from removable media, it must still be possible to upgrade via the Internet.
- App-FW-041 *facilitate handling of permissions*: app permissions
- App-FW-042 *provide data storage structure to an app*: private data and optionally per-app data, per-device data, per-user data.
- App-FW-043 *an app can't contain more than one program [...] or more than one agent/service*: What's in an app

  There has been some resistance to this requirement, and we have written the requirements in this document to say that vendors may impose this limit, but the framework should not.
- App-FW-044 *system extensions*: What's in an app
- App-FW-045 *downloaded and installed only once* (i.e. apps appear to be per-user but are really system-wide): Installation management
- App-FW-046 *queueing mechanism for app download* (i.e. apps do not install in parallel): Software download limiting
- App-FW-047 *App upgrades shall be checked periodically*: Upgrade.