Preferences and persistence

Contents

Introduction

This documents how system services and apps in Apertis may store preferences and persistent data. It considers the security architecture for storage and access to these data; separation of schemas, default values and user-provided values; and guidelines for how to present preferences in the UI.

The Applications Design, and Global Search Design documents are relevant reading. The Applications Design[1] and the Global Search Design[2] reference the need for storage of persistent data for apps. See Overall architecture for a design covering this.

The Robustness Design[3] document gives more detail on the requirements for robustness of main storage in the face of power loss.

Terminology and concepts

System Settings

A *system setting* is one which does not vary by user, and applies to the entire system. For example, networking settings. This document considers system settings which must be readable by multiple components —settings which are solely for the use of a single system service are out of scope, and may be stored in whichever way that service wishes (typically as a configuration file in /etc). This is particularly important for sensitive settings, for example the shadow user database in /etc/shadow, which must not be readable by anything except the system authentication service (PAM).

User settings

A *user setting* is one which does vary by user, but not by app. User settings apply to the whole of a user's session. For example, the language or theme.

App settings

An *app setting* is one which varies by user and also by an app bundle[4]. App settings apply only to a specific app bundle, and would not make sense outside the context of that app. For example, whether to enable shuffling tracks in the media player; whether to open hyperlinks in a new tab by default in the web browser; or the details for accessing a user's e-mail account.

[1] https://www.apertis.org/concepts/archive/application/applications/

[2] https://www.apertis.org/concepts/archive/application/global-search/

[3] https://www.apertis.org/concepts/archive/application_security/robustness/

[4] https://www.apertis.org/concepts/archive/application/applications/#bundle

Preferences

'*Preferences*'is the general term for system, user and app settings. The terms 'preference'and 'setting'will be used interchangeably throughout this document.

User services

A *user service* is as defined in the Multiuser Design document —a service that runs on behalf of a particular user. Throughout this document, this is additionally assumed to mean a *platform* user service, which is not tied to a particular app-bundle. Services inside of app bundles have the same access to settings as the app's UI.

Persistent data

Persistent data is app state which persists across multiple user sessions. For example, documents which the user has written, or the state of the user's pending downloads.

One distinguishing factor between preferences and persistent data is that vendors may override the default values for preferences (see Vendor overrides), but not for persistent data. For example, a vendor would not want to override information about in-progress downloads; but they might want to override the default background image filename for a user.

The persistent data for an app may be the same as the data it shares between user sessions, or may differ. The difference between persistent data and data for sharing between apps is discussed in the Multiuser Design document.

Persistent data is stored on main storage, whereas shared data is expected to be passed in memory —so while the sets of data are the same, the mechanisms used to handle them are different. Persistent data is always private to an app, and cannot be read by another app or user.

Persistent data might cover all state in an application —such that restoring its persistent data when starting the application is sufficient to make it appear as if it had been suspended, rather than exited. Or persistent data might cover some subset of this. The decision is up to the application authors.

Main storage

A flash disk, hard disk, or other persistent data storage medium which can be used by the system. This term has been chosen rather than the more common *persistent storage* to avoid confusion with persistent data.

GSettings

GSettings[5] is an interface provided by GLib for accessing settings. As an interface, it can be backed by different storage backends —the most common is dconf, but a key file backend is available for storage in simple key files.

GSettings uses a concept of 'schemas', which define available settings, their data types, and their default values. Each setting is strictly typed and must have a default value. A schema has an ID, and is 'instantiated'at one or more schema paths. Typically, a schema will be instantiated at a single path, but may be instantiated at multiple paths to support storing the same settings for multiple objects. For example, a schema for an e-mail account could require a server name, username and protocol, and be instantiated at multiple paths[6], one path for each configured e-mail account.

AppArmor

AppArmor[7] is an access control framework used by Apertis to enforce fine-grained permissions across the entire system, restricting which files each process can open.

Requirements

Access permissions

Access controls must be enforceable on preferences. Read and write permissions must be available. It is assumed that if a component has read permission for a preference, it may also be notified of any changes to that preference's value. It is assumed that if a component has write permission for a preference, it may also reset that preference.

A suggested security policy for preferences implements a downwards flow for **reads**:

- **Apps** may read their own app settings, user settings for the current user, and all system settings.
- **User services** may read the user's application settings, user settings for the current user, and all system settings.
- **System services** may read their own app settings, and all system settings.

Writes are generally only allowed at the same level:

- **Apps** may write their own app settings.

[5] https://developer.gnome.org/gio/stable/GSettings.html#GSettings.description
[6] https://developer.gnome.org/gio/stable/GSettings.html#gsettings-relocatable
[7] http://apparmor.net/

- **User services** may write user settings for the current user.
- **System services** may write system settings for all users, user settings for any user, and app settings for any app for any user.

Note that apps must not be able to read or write each others'settings. Similarly for user services and system services.

Persistent data is always private to a (user, app) pair, though it can be accessed by user services and system services.

Writability

As well as the value of a preference, components must be able to find out whether the preference is writable. A preference may be read-only if the component doesn't have write permission for it (Access permissions) or if it is locked down by the vendor vendor lockdown).

This does not apply to persistent data, which is always read–write by the (user, app) pair which owns it.

Rollback

As discussed in the Applications Design document[8], applications may be rolled back to a previously installed version, but it is the application's responsibility to handle any preference changes that occurred in a future version.

System and app bundle upgrades

As per the Applications Design[9] and the System Update and Rollback design[10], applications must also support upgrading preferences and persistent data from previous application versions to the current version.

Factory reset

The system must provide some means for the user to reset the state of all apps to a factory default for a particular user, or for all users. This is necessary for supporting removing user accounts, refreshing the car for transfer to a new owner, or clearing the state of a temporary guest account (see the Multiuser Design document). Similarly, it must support clearing the state of a single (user, app) pair.

The factory reset must support resetting preferences, persistent data, or both.

[8] https://www.apertis.org/concepts/archive/application/applications/#roll-back

[9] https://www.apertis.org/concepts/archive/application/applications/

[10] https://www.apertis.org/concepts/platform/system-updates-and-rollback/

Abstraction level

The preferences and persistent data APIs may want to abstract the underlying storage backend, for example to support uniform access to preferences stored in multiple locations. If so, details of the underlying storage backend must not be present in the abstraction (a ’leaky abstraction’) —for example, SQL fragments must not be used in the interface, as they tie the implementation to an SQL-based backend and a specific schema.

Conversely, any more than one layer of abstraction is an unnecessary complication.

Minimising I/O bandwidth

As with all components which use main storage, the preferences and persistent data stores should minimise the I/O load they impose on main storage. This is a particular concern at system startup, where typically a lot of data must be loaded from main storage, and hence I/O read efficiency is important.

Atomic updates

The system must make atomic writes to main storage, so that preferences or persistent data are not corrupted or lost if power is lost part-way through saving changes.

An atomic write is one where the stored state is either the old state, or the new state, but never an intermediate between the two, and never missing entirely. In other words, if power is lost while updating a preference, upon rebooting either the old value of the preference must be loadable, or the new value must be loadable.

See the Robustness Design document, §3.1.1 for more details on general robustness requirements.

Transactional updates

The system must allow updates to preferences to be wrapped in transactions, such that either all of the preferences within a transaction are updated, or none of them are. Transactions must be revertable before being applied permanently.

Performance tradeoffs

Preferences are typically written infrequently and read frequently; access patterns for persistent data depend on the app. The implementation should play to those access patterns, for example by using locking which favours readers over writers.

Data size tradeoffs

It is not expected that preference values will be large —a few tens of kilobytes at most. Conversely, persistent data may range in size from a few bytes to many megabytes. The implementation should use a storage format suitable to the expected data size.

Concurrency control

As system preferences may affect security policy, reading them should be race free, particularly from time-of-check-to-time-of-use[11] race conditions. For example, if a preference is changed by process C while process R is reading it, process R must either see the new value of the preference, or see the old value of the preference *and* subsequently be notified that it has changed.

Similarly for persistent data.

Vendor overrides

It may be desirable to support *vendor overrides*, where a vendor shipping Apertis can change the default values of the (app, user or system) preferences before shipping to the end user. For example, they may change the default background image shown to the user.

If these are supported, resetting a preference to its default value (for example, if doing a Factory reset) must restore it to the vendor-supplied default, rather than the Apertis default. There is no need to be able to access the Apertis default at any time.

This does not apply to persistent data.

Vendor lockdown

It may also be desirable to support *vendor lockdowns*, where a vendor shipping Apertis can lock a preference so that end users or non-privileged applications may not change it. For example, they may wish to lock the URI which is checked for system updates.

This does not apply to persistent data.

User interface

There must be some user interface (UI) for setting preferences. This may be provided by a system preferences application, as a separate window in each application, or as individual widgets embedded throughout an application's interface; or a combination of these options.

This does not apply to persistent data.

[11] http://en.wikipedia.org/wiki/Time_of_check_to_time_of_use

Control over user interface

It must be possible for the vendor to have complete control over the way preferences are presented if all applications'preferences are presented in a system preferences application.

This does not apply to persistent data.

Rearrangeable preferences

It must be possible for a vendor to rearrange the preferences from applications if they are presented in a system preferences application, so that (for example) all 'privacy'preferences are presented in a page together.

Searchable preferences

It must be possible for a system preferences application provided by the vendor to allow the user to search all preferences from all applications.

Storage of user secrets and passwords

There must be a secure way to store user secrets and passwords, which preserves confidentiality of these data. This may be separate from the main preferences or persistent data stores.

Preferences hard key

There must be support for a preferences hard key (a physical button in the vehicle) which when pressed causes the currently active application's settings to be displayed. If no applications are active, it could display the system preferences. Some vehicles may not have such a hard key, in which case the functionality should be ignored.

Existing preferences systems

This chapter describes the conceptual model, user experience and design elements used in various non-Apertis operating systems'support for preferences and persistent data, because it might be useful input for decision-making. Where available, it also provides some details of the implementations of features that seem particularly interesting or relevant.

GNOME Linux desktop

Preferences

On a modern GNOME desktop, from which Apertis uses a lot of components, settings are stored in multiple places.

- **System settings**: Stored in /etc by each system service, typically in a text file with a service-specific format. A lot of them have a system-wide default value, and may be overridden per user (for example, each user can set their own timezone and locale, with a system-wide default).
- **User settings**: Defined by shared GSettings schemas (such as org.gnome.system.locale), or schemas specific to individual user services (such as org.freedesktop.Tracker). The values are stored in dconf (see below).
- **App settings**: Defined by app-specific GSettings schemas. The values are stored in dconf (see below).

dconf[12] supports multiple layered databases, each stored separately. For each settings key, a value set for it in one layer overrides any values set in the layers below. The bottom (read-only) layer is always the set of default values which are provided by the schema file. This layered approach allows the system administrator to change settings system-wide in a system database, but also allows users to override those settings in their per-user database. It allows a user to reset all their settings by deleting their per-user database —at which point, the values from the next layer down (typically either a system database or the defaults from schema files) will be used for all settings keys.

Lockdown[13] is supported in dconf in the opposite direction: keys may be locked down at a particular level, and may not be set at levels above that one (but may be set at levels below it, as defaults).

Architecturally, dconf allows direct read-only access to all databases —each app reads settings values directly from the database. Writes to the databases are arbitrated through a per-user dconf daemon which then forces each app to refresh its read-only view of the settings. This allows for fast concurrent reads of settings, at the cost of making writes expensive.

dconf does *not* support access controls, and does not support storing different schemas in different databases at the same layer. Hence a user either has write access to the whole of a system database, or write access to none of it. As the dconf daemon runs per user, any app accessing the daemon may write to any settings key, either its own app settings, another app's settings, or the user's settings.

Persistent data

Persistent data is stored in application-defined formats, in application-defined locations, although many follow the XDG Base Directory Specification[14], which puts cache data in XDG_CACHE_HOME (typically ~/.cache) and non-cache

[12] https://developer.gnome.org/dconf/unstable/dconf-overview.html
[13] https://developer.gnome.org/dconf/unstable/dconf-overview.html#id-1.2.7
[14] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

data in XDG_DATA_HOME (typically ~/.local/share). Below these two directories, applications create their own directories or files as they see fit. There is no security separation between applications, but the normal UNIX permissions restrict access to only the current user.

There are no APIs available in GNOME for automatically persisting an entire application's state —if an application wishes to do this, it must implement its own serialisation and deserialisation functions and save to a file, as above.

Secrets and passwords

On a GNOME or KDE desktop, all user secrets, passwords and credentials are stored using the Secret Service[15] API. In GNOME, this API is implemented by GNOME Keyring; in KDE, by KWallet.

The API allows storage of byte array 'secrets'(such as passwords), along with non-secret attributes used to look them up, in an encrypted storage file which must be unlocked by the user before it can be accessed by applications. Unlocking it may be automatic if the user does not set a password on the file (or if the password is identical to the user's login password). Secrets are stored in 'collections', which may group them for different purposes, and which are encrypted separately.

An application must open a session with the secret service in order to access secrets. The session may be used to encrypt secrets while they are in transit between the service and application, and allows for encryption algorithm negotiation for this purpose.

For certain actions, the secret service may need to interact directly with the user in order to establish a trusted path to the user, and avoid (for example) requiring the user to enter their password into a potentially untrusted application for that application to forward it to the service.

Android

Preferences

Apps can use the SharedPreferences class[16] to read and write preferences from named preferences files, with apps typically using a single preferences file with a default name. These files are stored per-app, and are private to that app by default, but may be shared with other apps, either read-only or read–write.

Preferences are strongly typed, and default values are provided by the app at runtime. There is no concept of layering or of schemas —all definition of the preferences files is handled at runtime.

Preferences are saved to disk immediately.

[15] https://specifications.freedesktop.org/secret-service/latest/index.html
[16] http://developer.android.com/guide/topics/data/data-storage.html#pref

Android uses a custom XML format[17] to allow apps to define preference UIs (known as 'activities'in Android terminology). This format can define simple lists of preferences, through to complex UIs with grouped preferences, subscreens, lists of subscreens, and custom preference widgets. Implementing features such as making one preference conditional on another is possible, but requires complex XML.

A PreferenceFragment[18] can be used to automatically build a screen in an application to display preferences, loading them from the XML file. It will load the current values of the preferences from the SharedPreferences store, and will write new values back to the store as the preferences are modified in the UI.

In order for the system to display the preferences for a particular application, it must execute one or more of the PreferencesFragment classes from that application.

Persistent data

Android offers several options for persistent data[19]:

- **Internal storage**: Files in a per-(user, app) directory, which may optionally be made world-readable or writable to allow access to other apps or users (though this is strongly discouraged).
- **External storage**: Files in a world-readable storage area which is accessible to the user, such as an SD card. Accessible to all other apps and users which hold the READ_EXTERNAL_STORAGE or WRITE_EXTERNAL_STORAGE permissions.
- **SQLite database**: Arbitrary app-defined tables in a per-(user, app) SQLite database. This cannot be shared with other apps or users.
- **Network connection**: Using the normal networking APIs, Android suggests that data can be stored on servers controlled by the app developers. It provides no special API for this.

For saving an application's state, Android offers a persistence API on the Activity class[20]. This automatically saves the state of all UI elements (such as the text in an entry widget, and the position of a list), but cannot automatically save application-specific internal state (member variables). For this, the application must override two toolkit methods (onSaveInstanceState() and onRestoreInstanceState()) and implement its own serialisation and deserialisation of state to a set of key–value pairs which are then stored by Android.

[17] http://developer.android.com/guide/topics/ui/settings.html#DefiningPrefs

[18] http://developer.android.com/guide/topics/ui/settings.html#Fragment

[19] http://developer.android.com/guide/topics/data/data-storage.html

[20] http://developer.android.com/training/basics/activity-lifecycle/recreating.html

Secrets and passwords

Android recommends storing secrets and passwords in two ways. For authentication credentials for online services, it provides an AccountManager API[21] which abstracts authentication for known online services (which are supported by pluggable backends, potentially provided by application bundles) and stores the credentials in an OS-wide store. The service handles authenticating and re-authenticating when the login session ends.

For secrets which are not for online accounts, or otherwise do not fit the AccountManager pattern, Android recommends[22] using the normal preferences API (Preferences), as while preferences are not encrypted in storage, they are only accessible to the application which owns them, so cannot be stolen by other applications. However, if the sandboxing system is compromised (potentially by an attacker with physical access to the device), the stored secrets will be accessible in plaintext.

iOS

Preferences

iOS stores preferences as key–value pairs[23], which are separated into domains by user, application and machine. The same preference may be set in multiple domains[24], and they are searched in a defined priority order to determine which value to use. This means that an application may, for example, choose to share a given preference between all users of that application on a given machine.

Application IDs use the standard reverse domain name syntax to ensure uniqueness.

Preference values may be any type supported by Core Foundation property lists[25], including strings, integers and arrays. Default values must be coded into the application.

Preference keys may be generated at runtime by the application, and do not have to be defined in a schema in advance. However, it is typical to use pre-defined property lists.

Preferences are synchronised with the on-disk store manually, so the application chooses when they are written to disk.

[21] http://developer.android.com/reference/android/accounts/AccountManager.html

[22] http://stackoverflow.com/questions/785973/what-is-the-most-appropriate-way-to-store-user-settings-in-android-application/786588#786588

[23] https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/CFPreferences.html#//apple_ref/doc/uid/10000129-SW1

[24] https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/Concepts/PreferenceDomains.html

[25] https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPropertyLists/CFPropertyLists.html#//apple_ref/doc/uid/10000130i

On certain Apple operating systems, preferences may be 'managed'by the administrator[26], setting an override value which overrides any value set by the user for a given preference key.

Application preferences can either be presented as part of the application, using normal UI widgets, and accessing the NSUserDefaults class[27] for the preference values. Or they can be presented as part of the system-wide settings application[28], which builds the UI for each application's preferences dynamically from that application's property list file for preferences. An application may provide multiple property list files to build a hierarchy of preferences pages. The system-wide settings application accesses NSUserDefaults on behalf of the application to update the stored preferences.

Persistent data

iOS offers several options for persistent data:

- **Filesystem**: Arbitrary files may be written to the filesystem in various app-specific locations[29].
- **Core Data API**: This is an object-graph management API[30], which allows versioned control of instances of objects created from a schema. Instead of being used by an application to persist data, this API is designed to form the core of the application's data model. It supports editing and discarding edits, undo, redo, versioning of the object schema, and large data sets.
- **Property List API**: A property list is a hierarchical, structured piece of data, consisting of primitive data types, arrays and dictionaries which may be nested arbitrarily[31]. Property lists can therefore be used to store arbitrary application data. There is an API to serialise them to the file system.
- **SQLite**: The standard SQLite API may be used, backed by a file, to store relational data in a database.

[26] https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFPreferences/Concepts/BestPractices.html#//apple_ref/doc/uid/TP30001219-118191

[27] https://developer.apple.com/library/ios/documentation/Cocoa/Reference/Foundation/Classes/NSUserDefaults_Class/index.html#//apple_ref/occ/cl/NSUserDefaults

[28] https://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/UserDefaults/Preferences/Preferences.html#//apple_ref/doc/uid/10000059i-CH6-SW6

[29] https://developer.apple.com/library/ios/documentation/FileManagement/Conceptual/FileSystemProgrammingGuide/AccessingFilesandDirectories/AccessingFilesandDirectories.html#//apple_ref/doc/uid/TP40010672-CH3-SW11

[30] https://developer.apple.com/library/prerelease/ios/documentation/DataManagement/Devpedia-CoreData/coreDataOverview.html#//apple_ref/doc/uid/TP40010398-CH28

[31] https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/PropertyLists/AboutPropertyLists/AboutPropertyLists.html

For persisting an entire application's state, iOS provides a solution[32] similar to [Android][Persistent data]. The developer must annotate each UI view class which needs to be saved and restored, and the UI toolkit will automatically persist the state of the widgets in that view when the application is suspended. As with Android, the developer must implement two methods for serialising and deserialising application-specific state from member variables: encodeRestorableStateWithCoder and decodeRestorableStateWithCoder.

Secrets and passwords

iOS uses the same keychain API[33] as OS X. This provides a system service for storing secrets, passwords and certificates. They are encrypted in storage, using an encryption key which is derived from the iOS application's ID and the user' s password.

The keychain is encrypted in backups, and stored without its encryption key, so an attacker cannot extract secrets from backups.

An iOS application can access the secrets it has stored in the keychain, but cannot access secrets from other applications. There is no way to (for example) share login details for a given website between all applications which access that website —they must all query the user for the details and store them separately. This differs from OS X, where all applications can access any stored secrets, subject to the user approving the access (trusting the application).

GENIVI

Preferences and persistent data

GENIVI does not differentiate between preferences and persistent data, and provides one low-level API for saving and loading persistent data. It does not support automatically persisting an entire application's state.

The GENIVI Persistence Management system[34] handles all data read and written during the lifetime of an IVI system. It aims to provide a standard API for all GENIVI platforms to use, which reliably stores data in the face of power disturbances, and the limited write-cycle lifetime of some non-volatile storage devices (flash memory).

It is split into four components:

- Client library: API for writing key–value or arbitrary data to a file, which

[32] https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/PreservingandRestoringState.html

[33] https://developer.apple.com/library/ios/documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html#//appl_ref/doc/uid/TP30000897-CH203-TP1

[34] http://docs.projects.genivi.org/persistence-client-library/1.0/Persistence_ArchitectureManual.pdf

may be used by only the current application, or shared between all applications.

- Administration service: system for installing default values and configuration for the data storage for each application; backing up and restoring stored data; and implementing factory reset of data.
- Common object: used by the other components to access key–value databases through a caching layer.
- Health monitor: system under development to implement data recovery in the case of corruption or loss, using existing backups.

The GENIVI Persistence Management system only supports storage of data as byte arrays —applications must serialise and deserialise their data formats themselves. Similarly, it does not implement versioning of stored data.

The data storage code is implemented as a set of plugins for the client library, implementing different methods for storing data. There are various types of plugins implementing layers of functionality such as hardware information querying, encryption, early loading of data, and the default storage backend.

Key–value data is limited to 16KB per key. Keys are stored per-application, namespaced by an application-chosen arbitrary identifier. As persistent data is stored in a separate file per application, Unix users and groups may be used to enforce access control on the persisted data.

GENIVI has investigated providing an SQLite API for relational data storage, and has provided recommendations for it[35], but has not shipped a version with SQLite support (as of version 0.3.0 of this document).

To persist an application's state, the developer must manually implement serialisation and deserialisation of all UI and internal state of the application using the Persistence client library.

Secrets and passwords

Similarly, GENIVI has no specialised API for storing secrets and passwords —applications must use the persistence management system. The system does allow for encrypted storage of persistent data using a plugin —but that encrypts all stored data, including preferences and application state.

Approach

Preferences and persistent data have largely separate requirements: preferences are small amounts of data; need to be accessed by multiple components; will typically be read much more frequently than they are written; and need to

[35] http://docs.projects.genivi.org/persistence-client-library/1.0/Persistence_ClientLibrary_UserGuide.pdf

support features like Vendor overrides andvendor lockdown. Persistent data may vary from small to large amounts of data; will be read *and* written frequently; in app-specific formats; and do not need to be accessed by other components.

The expected amount of data to be stored, and the relative frequency of reads and writes of that data, is an important factor in the choice of storage format to use. Preferences should be stored in a format which is optimised for reads; persistent data should be stored in a format which is optimised for frequent reads and writes, since apps should update it frequently as they may be killed at any time.

For these reasons, we suggest preferences and persistent data are handled entirely separately. The following sections (6 and 7) will cover them separately, giving our recommended approach and justifications which refer back to the requirements (section 3).

User secrets and passwords (Storage of user secrets and passwords) have different requirements again:

- Confidentiality in storage (encryption).
- Sharing secrets and passwords for a given resource (such as website) between all applications using that website (i.e. secrets and passwords are not necessarily specific to an application, while preferences typically are).
- No fixed schema: the credentials required to access a given service (such as website) may change over time as that service changes.

As the system explicitly does not support full-disk encryption (for performance reasons), user secrets and passwords should be stored via the freedesktop.org Secrets D-Bus API[36], rather than the preferences or persistence APIs. The Secrets D-Bus API explicitly handles encryption of the secret store, whereas a general design for a preferences system should have no need for encryption, and hence adding it to the API would be an unnecessary complication for 90% of the use cases. Accordingly, confidential data will not be considered in the approach below.

For further discussion and designs on the topic of secrets and passwords, see the Security design document[37].

Preferences approach

Overall architecture

Access to app, user and system settings should be through the GSettings API, through the dconf backend for system / user schemas and the key-file backend

[36] http://standards.freedesktop.org/secret-service/

[37] https://www.apertis.org/concepts/archive/application_security/security/

for app schemas, the latter being chosen due to the integrated support for sandboxed settings in Flatpak. (Refer to GNOME Linux desktop for an overview of the way GSettings and dconf fit together.) As system settings are defined as those settings which are accessed by multiple components, settings which are solely for the use of a single system service may be stored in other ways, and are beyond the scope of this document.

Each component should have its own GSettings schema:

- **App schemas**: In the form net.example.MyApplication.SchemaName. Each app may have zero or more schemas, but all must be prefixed by the app ID (in this case, net.example.MyApplication; see the Applications Design document for details on the application ID scheme) to provide a level of namespacing.

- **User schemas**: These may have any form, and will typically re-use existing cross-desktop schemas, such as org.gnome.system.locale, as these are supported by many existing software components used by Apertis.

- **System schemas**: These may have any form, similarly.

Schema files for apps should be packaged with their app. For user services, they could be packaged with the most relevant service, or in a general purpose gsettings-desktop-schemas package (adapted from Debian) and an accompanying apertis-schemas package for Apertis-specific schemas.

All reads and writes of all settings should go through the normal GSettings interface —leaving access controls and policy to be implemented in the backend. App code therefore does not need to treat reads and writes differently, or treat app, user and system settings differently.

The use of GSettings also means that a single schema may be instantiated at multiple schema paths. Typically, a schema will only be instantiated at the path matching its ID; but a *relocatable* schema may be instantiated at other paths. This can be used to store settings for multiple accounts, for example.

It is expected that each app will handle any upgrades to its preference schemas, for example from one major version of the app to the next (System and app bundle upgrades). Apertis will not provide any special APIs for this. As this is highly dependent on the structure of the preference keys an app is storing, Apertis can provide no recommendations here. Note, however, that GSettings is designed with upgradability in mind: new preference keys take their value from the schema-provided defaults until the user sets them; the values for old preferences which are no longer in the schema are ignored. It is recommended that the type or semantics of a given GSettings key is not changed between versions of an app bundle —if it needs to be changed, stop using the old key, migrate its stored value to a new key, and use the new key in newer versions of the app bundle.

Requirements

Through the use of the GSettings API, the following requirements are automatically fulfilled:

- Writability —using g_settings_is_writable()
- System and app bundle upgrades —old keys are either kept, or superseded by new keys with migrated values if their type or semantics change
- Factory reset —for individual keys, using g_settings_reset(); support for resetting entire schemas needs to be supported by the designs below
- Abstraction level —GSettings serves as the abstraction layer, with the individual backends below adding no further abstractions
- Transactional updates —GSettings provides g_settings_delay(), g_settings_apply() and g_settings_revert() to implement in-memory transactions which are serialised in the backend on calling apply
- Concurrency control —g_settings_get() automatically returns the default value if no user-set value exists; there is no atomic API for setting settings
- User interface —g_settings_bind() can be used to bind a GSettings key to a particular UI widget, allowing interface UIs to be built easily (noting the argument in User interface that preferences UIs should not be automatically generated)

Other requirements are fulfilled separately:

- Control over user interface —by generating preferences windows from GSettings schemas in the system preferences application (Searchable preferences)
- Rearrangeable preferences —by hard-coding more behaviour in the system preferences application (User interface)
- Searchable preferences —searching over summaries and descriptions in GSettings schemas (Security policy)
- Storage of user secrets and passwords —using the freedesktop.org Secrets D-Bus API as in the Security design (section 5)
- preferences hard key —implemented according to the Hard Keys design preferences hard key1)

Proxied dconf backend

In its current state (May 2015, detailed in GNOME Linux desktop), dconf does not support the necessary fine-grained access controls for multiple components accessing the user and system schemas. However, a design is being implemented upstream to proxy access to dconf through a separate service which imposes access controls based on AppArmor (mostly implemented as of January 2016).

On the assumption that this work can be completed and integrated into Apertis on an appropriate timescale (see Summary of recommendations), this leads to a design where the dconf daemon runs as a system service, storing all settings in one database file per default layer:

- **User database**: ~/.config/dconf/user
- **System database**: /etc/dconf/db/local

This would be implemented as the dconf profile:

```
user-db:user
system-db:local
```

All accesses to dconf would go through GSettings, and then through the proxy service which applies AppArmor rules to restrict access to specific settings, implementing the chosen security policy (Access permissions). The rules may, for example, match against settings path and the AppArmor label of the calling process.

The proxy service would therefore implement a system preferences service.

Vendor lockdown is supported already by dconf[38] through the use of lockdown files, which specify particular keys or settings sub-trees which may not be modified.

Resetting all system settings would be a matter of deleting the appropriate databases —the keys in that database will revert to the default values provided by the schema files. As this is a simple operation, it does not have to be implemented centrally by a preferences service. Resetting the value of an individual key is supported by the g_settings_reset() API, which is already implemented as part of GSettings.

The existing Apertis system puts

```
include <abstractions/gsettings>
```

in several of the AppArmor profiles, which gives unrestricted access to the user dconf database. This must change with the new system, only allowing the dconf daemon access to the database.

Requirements

This design fulfills the following requirements:

- Access permissions —through use of the proxy service and AppArmor rules
- Factory reset —by deleting the user's database or the user's per-app database
- Minimising io bandwidth —dconf's database design is optimised for this

[38] https://developer.gnome.org/dconf/unstable/dconf-overview.html

- Atomic updates —dconf performs atomic overwrites of the database
- Performance tradeoffs —dconf is heavily optimised for reads rather than writes
- Data size tradeoffs —dconf uses GVDB for storage, so can handle small to large amounts of data
- Vendor overrides —dconf supports vendor overrides inherently
- vendor lockdown —dconf supports vendor lockdown inherently

Development backend

In the interim, we recommend that the standard dconf backend be used to store all system, user and app settings. This will *not* allow for access controls to be applied to the settings (Access permissions), but will allow for development of OS components against the final GSettings interface.

Once the proxied dconf backend is ready, it can be packaged and the system configuration changed —no changes should be necessary in user services to make use of the changed backend.

This development backend would support vendor lockdown as normal.

Requirements

This design fails the following requirements:

- Access permissions —**unsupported** by the current version of dconf

It supports the following requirements:

- Factory reset —**partially supported** by deleting the user's database; resetting a (user, app) pair is not supported as all settings are stored in the same dconf database file
- Minimising io bandwidth —dconf's database design is optimised for this
- Atomic updates —dconf performs atomic overwrites of the database
- Performance tradeoffs —dconf is heavily optimised for reads rather than writes
- Data size tradeoffs —dconf uses GVDB for storage, so can handle small to large amounts of data
- Vendor overrides —dconf supports vendor overrides inherently
- vendor lockdown —dconf supports vendor lockdown inherently

Key-file backend

For Flatpaks, the GSettings key-file backend is used. (This could also be used as an alternative to the development backend before the proxied dconf backend is ready.) This allows the use of Flatpak's native filesystem sandboxing for security, but it creates a unique set of trade-offs that make it less suitable for the other use cases:

- lower read performance due to not being optimised for reads (or in general); this may be less of an issue for applications, which may not have as many settings in a single file
- requiring code changes in user services to switch from the key-file backend to the proxied dconf backend once it's ready; applications will stay using the key-file backend, so no migration is necessary
- requiring settings values to be migrated from the key-file store to dconf at the time of switch over; as before, the backend will stay being used by applications, so no migration is necessary
- not supporting vendor lockdown or vendor overrides; these are primarily intended for system settings, not application settings

Due to the need for code changes to switch away from this backend to a more suitable long-term solution such as the proxied dconf backend, we do not recommend this approach for system components.

In detail, the approach would be to use a separate key file for each schema instance. Flatpak applications handle this automatically, but for user and system schemas, this would require using `g_settings_key_file_backend_new()` and `g_settings_new_with_backend_and_path()` to manually construct the GSettings instance for each schema.

Access control for each schema instance would be enforced using AppArmor rules and Flatpak sandboxing which restrict access to each key file as appropriate. For example, apps would be given read–write access to the key file for their own app settings, but any key files or dconf databases used by user and system services would be unreadable or read-only.

For apps, they will be unable to directly access any non-key-file settings, thus vendor lockdown for those components is inherent in the design (see Application access to system settings). However, vendor lockdown for application settings is not supported.

For the system and user services themselves, vendor lockdown would be supported by vendors patching the AppArmor files to limit write access to specific schema instances. It would not support per-key lockdown at the granularity supported by dconf.

This code for creating the GSettings object could be abstracted away by a helper library, but the API for that library would have to be stable and supported indefinitely, even after changing the backend.

Requirements

This design fails the following requirements:

- Performance tradeoffs —GKeyFile is **equally non-optimised** for reads and writes
- Vendor overrides —**unsupported** by GKeyFile
- vendor lockdown —**unsupported** by GKeyFile

It supports the following requirements:

- Access permissions —supported by AppArmor rules and Flatpak permissions on the per-schema key files
- Factory reset —by deleting the appropriate key files
- Minimising io bandwidth —GKeyFile's I/O bandwidth is proportional to the number of times each key file is loaded and saved
- Atomic updates —GKeyFile performs atomic overwrites of the database
- Data size tradeoffs —GKeyFile's load and save performance is proportional to the amount of data stored in the file, so it is suitable for small amounts of data

Security policy

The key-file backend enforces security policy for apps through Flatpak's filesystem sandboxing, and the proxied dconf backend enforces security policy for user and system schemas through AppArmor rules. (The Development backend does not support implementing security policy at all.)

It is beyond the scope of this document to define how AppArmor rules and Flatpak manifests are configured.

Application access to system settings

Flatpak applications should not be able to see the full host dconf database. Therefore, access to that is not granted by default, rather allowing access to select settings by the use of the XDG Settings portal[39]. In order for this to function, a custom portal backend service[40] must be created that exposes the settings to applications. The use of the portals to access system and user settings does not place any restrictions on the actual storage of the settings, which may still be stored in dconf.

[39] https://flatpak.github.io/xdg-desktop-portal/#gdbus-org.freedesktop.portal.Settings

[40] https://flatpak.github.io/xdg-desktop-portal/#gdbus-org.freedesktop.impl.portal.Settings

User interface

Different options for building preferences user interfaces need to be supported by the system (Control over user interface):

- Individual preferences embedded at different points in the application UI.
- A preferences window implemented within the application.
- A system preferences application which controls displaying the preferences for all installed applications, plus system preferences.

In all cases, we recommend that preferences are defined using GSettings schemas, as discussed in Overall architecture, and that settings are read and written through the GSettings[41] API. This ensures that access control is enforced, and separates the structure of the preferences (including types and default values) from their presentation.

The choice of how preferences are presented ultimately lies with the vendor. In certain cases, an application may choose to display a preference embedded into its UI (for example, as a satellite/hybrid/standard view selector overlaid on a map view), if it makes sense for that preference to be displayed in-context as opposed to in a preferences window. This user experience is something which should be checked as part of app validation.

The majority of preferences should be displayed in a separate preferences window. In order to allow this window to be embedded into a system preferences application if the vendor desires it, the preferences window must be automatically generated. This is because:

- arbitrary code from arbitrary applications must not be run in the context of the system preferences application; and
- the system preferences application cannot be shipped with manually-coded preferences windows for all applications which could ever be installed.

However, automatically generated UIs generally give a bad user experience, due to the limited flexibility a designer has on them, so are suitable only for basic preferences (such as toggle switches; see Discussion of automatically generated versus manually coded preferences UIs). There may be cases where an application has a particular preference which Apertis provides no widgets suitable for editing it. In these infrequent cases, it must be possible for the system preferences application to execute a stand-alone preferences window from the application to set that particular preference.

System preferences application

If an application has preferences, it must give the path to the GSettings schema file which defines them in its application manifest.

[41] https://developer.gnome.org/gio/stable/GSettings.html#GSettings.description

The system preferences application should display a list of applications as its initial screen, including entries for system preferences which it implements itself. The applications listed should be the ones whose manifests specify GSettings schema files, and the application name and icon should also be retrieved from the application manifest and displayed.

If the user selects an application, a preferences window should be displayed which shows all the preferences in the application's GSettings schema file. See Generating a preferences window from a GSettings schema file for details of how this is done. Note that if the schema file defines multiple levels of schema, they should be presented as a hierarchy of pages, with preferences only being shown on leaf pages.

As a system application, the system preferences application would have permission to read and write any application settings via GSettings, so forms part of the trusted computing base (TCB) for preferences.

The vendor may choose the security policy for which users may edit system preferences (such as the language or background) —they could either allow all users to edit these, or only allow administrative users (such as the vehicle owner) to edit them. If so, we recommend showing the entries for these preferences anyway, but making the widgets insensitive and presenting an authentication dialogue for the administrator to authenticate with before allowing the settings to be edited, see the Multi-User Transactional Switching document[42].

Per-application preferences windows

If the vendor wishes to implement a user experience where each application shows its own preferences window, this should be implemented using the system preferences application in a different mode. A settings button or menu entry in the application should launch the system preferences application.

It should support being launched with the name of a GSettings schema to show, and it would render a preferences window from that schema (see Generating a preferences window from a GSettings schema file). If the schema file defines multiple levels of schema, they should be presented as a hierarchy of pages, with preferences only being shown on leaf pages. It is up to the vendor whether the user can navigate 'up'from the top level of the schema to a list of all applications.

As the system preferences application is part of the TCB for preferences, it must not allow an application to launch it with the name of a GSettings schema file which does not belong to that application. For example, that would allow one application to trick the user into editing their preferences for another application.

[42] https://www.apertis.org/concepts/archive/application_security/multiuser-transactional-switching/

Generating a preferences window from a GSettings schema file

A GSettings schema file[43] can be turned into a UI using the following rules:

- A <schema> element is turned into a preference page. If it has an extends attribute, the widgets from the schema it extends are added to the preferences page first.
- The first non-relocatable <schema> element in a <schemalist> will be taken as providing the preferences page for the application. Subsequent <schema> elements will be ignored unless pulled in as preferences subpages using a <child> element.
- A <child> element is turned into an entry to show a preferences sub-page for the corresponding sub-schema. The label for this entry should come from a new (non-standard) label attribute on the <child> element.
- Relocatable <schema> elements (those without a path attribute) are ignored unless pulled in as a preferences sub-page using a <child> element.
- A <key> element is turned into a widget with its label set from the <summary> element and its description set from the <description> element. The type of widget is set by the type attribute, which specifies a GVariant type[44]:
  - b (boolean): Switch or checkbox widget.
  - y, n, q, i, u, x, t (integers): Integer spin button. Its range is set to the smaller of the bounds of the integer type or the values of the <range> element (if present).
  - h (handle): Not supported.
  - d (double): Floating point spin button. Its range is set to the smaller of the bounds of the double type or the values of the <range> element (if present).
  - s (string): Text entry widget. If a <choices> element is present, a drop-down box should be used instead, displaying the options from the <choice> elements.
  - o (object path): Not supported.
  - g (type string): Not supported.
  - ? (basic type): Not supported.
  - v (variant): Not supported.
  - a (array): Not supported in any form.
  - m (maybe): Not supported in any form.

[43] https://gitlab.gnome.org/GNOME/glib/-/blob/main/gio/gschema.dtd
[44] https://developer.gnome.org/glib/stable/glib-GVariantType.html#id-1.6.18.6.9

    - (), r (tuple): Not supported in any form.
    - {} (dictionary): Not supported in any form.
    - * (any): Not supported in any form.

- If a <key> element contains an enum attribute and no type attribute, a drop-down box should be used, displaying the options from the nick attributes of the <value> elements in the corresponding <enum> element.
- If a <key> element contains a flags attribute and no type attribute, a checkbox list should be used, displaying a checkbox for each each of the nick attributes of the <value> elements in the corresponding <flags> element.
- If a key's name attribute matches a mapping to a wizard application (see Support for custom preferences windows) in the application's manifest, that key should be displayed as a menu entry which, when selected, launches the wizard application as a new window.

Support for custom preferences windows

If an application has a particularly esoteric preference or set of preferences which are not supported by the generated preferences UI (see Generating a preferences window from a GSettings schema file), it may provide a 'wizard'application as part of its application bundle which allows setting those preferences (and only those preferences). For example, this could be used to show a 'wizard'for configuring an e-mail account; or a map widget for selecting a location.

A wizard application presents a single window of preferences, and its widgets cannot be integrated into a preferences window generated by the system preferences application —it must be launched using a menu entry from there.

The wizard application must be listed in the application's manifest as part of a dictionary which maps GSettings schemas or keys to commands to run.

For example, a particular manifest could map the key /org/foo/MyApp/complex-setting to the command my-app –show-complex-setting. Or a manifest could map the schema /org/foo/MyApp/EmailAccount to the command my-app –configure-email-account.

Application bundles which contain keys for this in their manifest should be subjected to extra app store validation checks, to establish that the wizard application's UI is consistent with other preferences UIs, and that it does not implement preferences which should be handled by a generated UI.

The wizard application must set the relevant preferences itself before exiting, and runs with the same privileges as the rest of the application bundle (so will only have access to that application's preferences, as per Security policy).

It may be necessary for the window manager to treat windows from wizard applications specially, so that they appear more like a window which is part of the system preferences application than a window from a separate application. This can be solved by adding appropriate metadata to the wizard application windows so the window manager treats them differently.

Searchability of preferences

To allow the system preferences application to search over all applications'preferences (Searchable preferences), it must load all the GSettings schemas from applications whose manifests specify a schema. Searching must be performed over the user-visible parts of the schema (the <summary> and <description> elements), and results should be returned as a link to the relevant application preferences window. System preferences should be included in the search results too.

Reorganising preferences

Implementing arbitrary reorganisation of preferences (Rearrangeable preferences) is difficult, as that requires an OEM to know the semantics of all preferences for all possibly installable applications.

We recommend that if an OEM wants to present a new group of a certain set of preferences, they must choose specific preferences from known applications, and implement a custom window in the system preferences application which displays those preferences. Each preference should only be shown if the relevant application is installed.

An alternative implementation which is more flexible, but which devolves more control to application developers, is to tag each preference in the GSettings schemas with well-defined tags which summarise the preference's semantics. For example, an application's preference for whether to submit usage data to the application data could be tagged as 'privacy'; or a preference determining the colour scheme to use in an application could be tagged as 'appearance'. The OEM could then implement a custom preferences window which queries all installed GSettings schemas for a specific tag and displays the resulting preferences. We do not recommend this option, as even with app store validation of the chosen tags, this would allow application developers too much control over the appearance of a system preferences window.

Preferences list widget

In order to help make all preferences UIs consistent (including those implemented by the vendor, System preferences application; and those implemented by application developers as wizard applications, Per-application preferences windows), Apertis should provide a standard widget which implements the conversion from GSettings schemas to UI as described in Generating a preferences window from a GSettings schema file.

This widget should accept a list of GSettings schema paths to display, and may optionally accept a list of keys within those schemas to display (ignoring the others), or to ignore (displaying the others); and should display all those keys as preferences. It should implement reading and writing the keys'values using the GSettings API, and must assume that the application has permission to do so (see Security policy). It must check for writability of preferences and make them insensitive if they are read-only (seevendor lockdown1). It cannot give the application more permissions than it already has.

If application developers use this widget, the vendor can ensure that preferences UIs are consistent between applications and the system preferences application through the theming of the widget.

Vendor lockdown

If the vendor locks down a key in a GSettings schema for an application (or system preference) vendor lockdown –supported by Proxied dconf backend and Development backend, but not Key-file backend), that is enforced by the underlying settings service (most likely dconf), and cannot be overridden or worked around by applications.

However, it is up to applications to reflect whether a preference is read-only (due to being locked down) in their UIs. This is typically achieved by hiding a preference or making its widget insensitive. Applications can use the g_settings_is_writable[45] method to determine whether a preference is read-only. Any preferences widgets provided by Apertis (Preferences list widget) must implement this already.

If an application developer uses a custom widget to display a preference, and forgets to check whether that preference is read-only, their application might enter an inconsistent state (which is their fault), but the system will not let that preference be written. Convenience APIs like g_settings_bind_writable[46] can reduce the risk of this happening.

Discussion of automatically generated versus manually coded preferences UIs

In an ideal world, our recommendation would be that: while automatically generating preference UIs can rapidly produce rough drafts, in our experience it can never result in a high-quality finished UI with:

- logically grouped options;
- correctly aligned controls;
- a concept of which preferences are most important, which ones are 'advanced', and which ones should be hidden;

[45] https://developer.gnome.org/gio/unstable/GSettings.html#g-settings-is-writable

[46] https://developer.gnome.org/gio/stable/GSettings.html#g-settings-bind-writable

- conditional defaults (for example, when you set up IMAP e-mail, the default port should be 143, except if you have selected old-style SSL in which case it should be 993); and

- the ability to hide or disable preferences that do not apply because of the value of another preference (for example, if you switch off Bluetooth completely, then the widget to change the name that is broadcast over Bluetooth should be hidden or disabled).

If the uniform appearance of preferences UIs is a concern, we believe this should be addressed through: convention; the default appearance of widgets in the UI toolkit; and the use of a set of human interface guidelines such as the GNOME HIG[47]. Specifically, we recommend that preferences are:

- integrated into the main application UI if there are only a small number of them;

- instant-apply[48] unless doing so would be dangerous, in which case they should be explicit-apply for all preferences in the dialogue (for example, changing monitor resolutions is dangerous, and hence is explicit-apply); and

- grouped logically in the UI.

If, after the preferences UIs of several applications have been implemented, some common widget patterns have been identified, we suggest that they could be abstracted out into new widgets in the UI toolkit. The goal of this would be to increase consistency between preferences UIs, without implementing essentially a separate UI toolkit for them, which would be the result of any template- or auto-generation-based approach.

An alternative way of thinking about this is that preferences are subject to a model–view split (the model is GSettings schema files; the view is the preferences UI), and it is typically inadvisable to generate a view from a model when following that pattern.

However, we realise that the goal of having a unified system preferences application with a consistent appearance (which is enforced) conflicts with these recommendations, and hence these recommendations are not part of our overall suggested approach.

Preferences hard key

A preferences hard key must be supported as detailed in the Hard Keys design. In a configuration where a system preferences application is used, it must launch that application, already open on the preferences window for the active application. If no application is active, or if the currently active application has

[47] https://developer.gnome.org/hig/stable/dialogs.html.en

[48] https://developer.gnome.org/hig/stable/dialogs.html.en#instant-and-explicit-apply

no GSettings schemas listed in its manifest file, the main page of the system preferences application should be shown.

In a configuration where applications implement their own preferences windows, the active application must be sent a 'hard key pressed'signal for the preferences hard key, which the application can handle how it wishes (i.e. by showing its preferences window). If there is no active application, the system preferences application (which in this configuration only contains system preferences) should be shown.

The policy for exactly what happens in each situation and configuration is under the control of the hard keys service, which is provided by the vendor. It should have access to the manifest for the active application so it can find information about GSettings schemas.

Existing preferences schemas

As GSettings is used widely within the open source software components used by Apertis, particularly GNOME, there are many standard GSettings schemas for common user settings. We recommend that Apertis re-use these schemas as much as possible, as support for them has already been implemented in various components. If that is not possible, they could be studied to ensure we learn from their design successes or failures.

- org.gnome.system.locale
- org.gnome.system.proxy
- org.gnome.desktop.default-applications
- org.gnome.desktop.media-handling
- org.gnome.desktop.interface
- org.gnome.desktop.lockdown
- org.gnome.desktop.background
- org.gnome.desktop.notifications
- org.gnome.crypto
- org.gnome.desktop.privacy
- org.gnome.system.dns_sd
- org.gnome.desktop.sound
- org.gnome.desktop.datetime
- org.gnome.system.location
- org.gnome.desktop.thumbnailers
- org.gnome.desktop.thumbnail-cache

- org.gnome.desktop.file-sharing

Various Apertis dependencies (for example, Mutter, Tracker, libfolks, IBus, Geoclue, Telepathy) use their own GSettings schemas already —as these are not shared, they are not listed.

Alternative model: If the locale is a system setting, rather than a user setting, systemd's localed[49] should be used. This would require the locale to be changed via the localed D-Bus API, rather than GSettings, which would affect the implementation of the system preferences app.

Persistent data approach

Overall architecture

As discussed in sections 5.3.1 and 7 of the Applications Design, and the Multiuser Design, there is a difference between state which an app needs to persist (for example, if it is being terminated to switch users), and state which an app explicitly needs to share (for example, if a transactional user switch is taking place to execute an action as a different user). The Multiuser Design encourages app authors to think explicitly about these two sets of state, and the differences between them. It is the app which chooses the state to persist, rather than the operating system —storage space is too limited to persist the entire address space of an app, effectively suspending it.

The state each app chooses to persist will differ, and cannot be predicted by Apertis. There could be a lot of state, or very little. It could be representable as a simple key–value dictionary, or might have a complex hierarchical structure.

Well-known state directories

As mentioned in the Applications Design document (sections 5.3.1 and 7), we recommend that Apertis provides a per-(user, app) directory for storage of persisted data, and a public API the app can call to find out that directory. The API should follow Flatpak conventions, differentiating between cache and non-cache state, with cache state going in $HOME/.var/*net.example.MyApp*/cache and non-cache state going in $HOME/.var/*net.example.MyApp*/data. This fulfils the factory reset requirement (Factory reset).

The former is effectively equivalent to a per-(user, app) XDG_CACHE_HOME directory, and the latter to a XDG_DATA_HOME, as defined by the XDG Base Directory Specification[50].

Flatpak permissions, by default, grant an app write access exclusively to its cache and data directories, and not to other apps'state directories, with AppArmor profiles potentially providing further defense-in-depth. This is the extent

[49] http://www.freedesktop.org/wiki/Software/systemd/localed/

[50] http://standards.freedesktop.org/basedir-spec/basedir-spec-latest.html

of the security needed, as state storage is simply an interaction between an app and the filesystem.

As with preferences, app bundles must be in charge of upgrading their own persistent data when the system is upgraded (or the app is upgraded) (System and app bundle upgrades). Recommendations are given in the subsections below.

Recommended serialisation APIs

As each app's state storage requirements are different, we suggest that Apertis provide several recommended serialisation APIs, and allow apps to choose the most appropriate one —or something completely different if that fulfils their requirements better.

Alongside, Apertis should provide guidelines to app developers to allow them to choose an appropriate serialisation API, and avoid common problems in serialisation:

- minimise writes to main storage (Minimising io bandwidth);
- ensure all updates to stored state are atomic (requirement Atomic updates); and
- ensure transactions are used for groups of updates where appropriate (Transactional updates).

> Atomic in the sense that either the old or new states are stored in entirety, rather than some intermediate state, if power is lost partway through an update.

Depending on the requirements it is believed that apps will have, some or all of the following APIs could be recommended for serialising state to main storage. For comparison, Android only provides a generic file storage API, and an SQLite API, with no implemented key–value store APIs[51]. Apps must implement those themselves.

GKeyFile

https://developer.gnome.org/glib/stable/glib-Key-value-file-parser.html

Suitable for small amounts of key–value state with simple types. Suitable for small amounts of data.

All updates to a GKeyFile are atomic, as it uses the atomic-overwrite technique: the new file contents are written to a temporary file, which is then atomically renamed over the top of the old file. Transactional updates can be implemented by saving the key file to apply the transaction, and discarding the in-memory GKeyFile object to revert it.

[51] http://developer.android.com/guide/topics/data/data-storage.html

The amount of I/O with a GKeyFile is small, as the amount of data which should be stored in a GKeyFile is small, and the file is only written out when explicitly requested by the app.

System upgrades have to be handled manually by app bundles —if the persistence data format has to change, the app must migrate data from the old format to the new format the first time it is run after an upgrade. In this case, it is recommended that all GKeyFiles used for persistent data contain a 'Version'key specifying the data format version in use.

GVDB

https://git.gnome.org/browse/gvdb

Memory-mapped hash table with GVariant[52]-style types, suitable for small to large amounts of data which are read much more frequently than they are written. This is what dconf uses for storage.

All updates to a GVDB file are atomic, as it uses the same atomic-overwrite technique as GKeyFile. Transactions are supported similarly —by writing out the updated database or discarding it.

The amount of I/O for reads from a GVDB file is small, as it memory-maps the database, so only pages in the data it actually reads (plus some metadata). Writes require the entire file to be updated, but are only done when explicitly requested by the app.

GVDB supports per-file versioning (though this is not currently exposed in the public API). This can be used for handling system upgrades (System and app bundle upgrades) —the database must be explicitly migrated from an old version to a new version when an upgraded app is first started.

SQLite

http://sqlite.org/

https://wiki.gnome.org/Projects/Gom

Full SQL database implementation, supporting simple SQL types and more complex relational types if implemented manually by the app. Suitable for medium to large amounts of data which are read and written frequently. It supports SQL transactions.

SQLite is not a panacea. It is designed for the specific use pattern of SQL databases with indexes and relational tables, with frequent reads and writes, and infrequent deletions of data. Apps will only get the best performance from SQLite by defining their own table structure, indices and relations; imposing a common key–value-style API on top of SQLite would give lower performance.

[52] https://developer.gnome.org/glib/stable/glib-GVariant.html

SQLite has limited support for SQL schema upgrades with its ALTER TABLE[53] statement, which supports renaming tables and adding new columns to tables. Apps must implement their own data migration from old to new versions of their database schema; documenting this is beyond the scope of this design.

Apps should only use SQLite if they have considered issues like their vacuuming policy —how frequently to vacuum the database after deleting data from it. See:

- *https://blogs.gnome.org/jnelson/2015/01/06/sqlite-vacuum-and-auto_vacuum/*
- *https://wiki.mozilla.org/Performance/Avoid_SQLite_In_Your_Next_Firefox_Feature*

If using GObjects to represent entries in an SQLite database, the GOM[54] wrapper around SQLite may be useful to simplify code.

GNOME-DB

http://www.gnome-db.org/

This is **not** recommended. It is an abstraction layer over multiple SQL database implementations, allowing apps to access remote SQL databases. In almost all cases, directly using Sqlite is a more appropriate choice.

When to save persistent data

As specified in the Applications Design (section 5.3.1), state is saved to main storage at times chosen by both the operating system and the app. The operating system knows when the logged in user is about to change, or when the system is about to be shut down; the app knows when it has changed some of its persistent state in memory, and hence needs to write it out to main storage.

An action could be implemented in each app which is triggered by the ActivateAction method of the org.freedesktop.Application D-Bus interface[55] if, for example, that interface is implemented by apps. When triggered, this action would cause the app to store its persistent state.

Recently used and favourite items

Section 6.3 of the Global Search Design specifies that an API for apps to store their favourite and recently used items in will be provided. As this is data shared from an app to the operating system, and is typically append-only rather than strongly read–write, we recommend that it be designed separately from the persistent data API covered in this document, following the recommendations given in the Global Search Design document.

[53] https://www.sqlite.org/lang_altertable.html

[54] https://wiki.gnome.org/Projects/Gom

[55] http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#dbus

Summary of recommendations

As discussed in the above sections, we recommend:

- Splitting preferences, persistent data storage and confidential data storage (Approach).
- Providing one API for preferences: GSettings (Overall architecture).
- Apps provide a GSettings schema file for their preferences, named after the app (Overall architecture).
- Existing GSettings schemas are re-used where possible for user and system settings (Existing preferences schemas).
- Using the normal GSettings approach for handling app upgrades (Overall architecture).
- Developing against the normal dconf backend for GSettings storage for user and systems schemas (section Development backend.
- Switching to the proxied dconf backend once it's ready, to support access control (Proxied dconf backend).
- Use the key-file backend for *only* applications; we do *not* recommend using it as an alternative to the dconf backends for system and user schemas. (Key-file backend).
- Permissions to modify user or system settings are controlled by the app's manifest (Security policy).
- User interfaces for preferences are provided by the vendor, automatically generated from GSettings schemas; or provided by applications (User interface).
- Apertis provides a standard widget to present GSettings schemas as a preferences UI (Preferences list widget).
- Preferences hard key support is added according to the Hard Keys design preferences hard key).
- Providing API to get a persistent data storage location (Well known state directories).
- Persistent data is private to each (user, app) pair (Well known state directories).
- Recommending various different data storage APIs to suit different apps' use cases (Recommended serialisation APIs).
- Apps explicitly define which data will persist, and are responsible for saving it and migrating it from older to newer versions (Overall architecture).

- Apps can be instructed to save their persistent state by the operating system via a D-Bus interface (When to save persistent data).
- User secrets and passwords are stored using the freedesktop.org Secrets D-Bus API, not the Apertis preferences or persistence APIs (Approach).