Interface discovery

Contents

Various features on Apertis require a way to discover the applications and/or agents that implement a particular set of functionality. We refer to the "API contract" for this set of functionality as an ***interface***.

Use cases

- A global search user interface[1] requires a list of agents that can act as "Auxiliary Sources" (see §6.2 in the Global Search design document). For example, a Spotify client might register itself as a search provider so that searching for a term in a global search will find artists or songs matching that term.
- An application that will display a Sharing[2] menu similar to the one in Android requires a list of applications with which files or data can be shared.
- A navigation app, potentially from an app-store, obtains points of interest[3] from a number of providers, again potentially from an app-store. In a "pull" model, the navigation app would consume the interface "points-of-interest provider" by sending queries to the implementors and getting results back, and the points-of-interest providers would implement that interface. Conversely, in a "push" model, the navigation app could implement the interface "points-of-interest sink", and the points-of-interest providers could consume that interface by sending points of interest to each sink.
- If more than one navigation app is installed (for example because an Apertis system includes the OEM's own simple navigation solution, but it is possible to install premium navigation software from the app-store), a settings user interface to select the preferred navigation app might need to list all the possible navigation apps.

[1] /images/apertis-global-search-design-0.3.2.pdf
[2] https://sjoerd.pages.apertis.org/apertis-website/concepts/sharing/
[3] https://sjoerd.pages.apertis.org/apertis-website/concepts/points_of_interest/

- Interface discovery could potentially be used with the interface "is the preferred navigation app" to start the navigation app on-demand. If it is, it must be possible to mark one as preferred.
- A navigation app could have a preferences dialog in which points of interest[4] providers can be selected or deselected. It should not display points of interest from deselected providers, and should not waste system resources on receiving points of interest from those providers. However, if another application also consumes points of interest, disabling a points of interest provider in the navigation app should not prevent it from being used by the other application.
- The platform could have a preferences dialog in which points of interest[5] providers can be selected or deselected. If a POI provider is deselected here, POI consumers such as the navigation app should behave as though the deselected provider had not been installed at all.

In other systems

GNOME Shell's search provider API[6] relies on applications registering their support for the search provider "interface" by installing files in `/usr/share/gnome-shell/search-providers`. This is not ideally suited to a platform like Apertis with a strong division between the "platform" and "app bundle" layers, and does not generalize trivially (each interface would have to define its own location in which to place metadata files).

The freedesktop.org Desktop Entry specification[7] shared by GNOME, KDE and other open source desktop environments uses `.desktop` metadata files to store metadata about applications. It defines an `Interfaces` key[8] whose value is a list of syntactically valid D-Bus interface names[9]. Each interface name may represent either a D-Bus interface, or any other "API contract"; there is no requirement that D-Bus is actually used.

Security considerations

Restricting who can advertise a given interface

If arbitrary ISVs[10] can publish app-bundles that advertise arbitrary interfaces, there is a risk that consumers of those interfaces would have an inappropriate level of trust in those app-bundles by assuming that only their own app-bundles

[4] https://sjoerd.pages.apertis.org/apertis-website/concepts/points_of_interest/
[5] https://sjoerd.pages.apertis.org/apertis-website/concepts/points_of_interest/
[6] https://git.gnome.org/browse/gnome-shell/tree/js/ui/remoteSearch.js
[7] http://standards.freedesktop.org/desktop-entry-spec/latest/
[8] http://standards.freedesktop.org/desktop-entry-spec/desktop-entry-spec-latest.html#interfaces
[9] http://dbus.freedesktop.org/doc/dbus-specification.html#message-protocol-names-interface
[10] https://en.wikipedia.org/wiki/Independent_software_vendor

can advertise "their" interfaces, for example "leaking" private information to them.

Communication between consumers and implementors

If a particular interface involves direct communication between a consumer and an implementor, then discovery is not sufficient: it is also necessary to ensure that the security model allows the consumer and the implementor to communicate. Conversely, if a particular interface forces all communication between a consumer and an implementor through a trusted intermediary such as Didcot, then it is necessary to ensure that the security model allows both the consumer and the implementor to communicate with the trusted intermediary, and that the trusted intermediary is able to determine that forwarding data between consumer and implementor will not violate the security model.

The desired security model for this interface is that some subset of interfaces are considered to be ***public interfaces***. Trusted platform components may list the implementors of any interface, public or not, and may initiate communication with those implementors. Store applications may list the implementors of public interfaces, and may initiate communication with the implementors of public interfaces, but cannot do the same for non-public interfaces.

Visibility of applications to other applications

Our security model does not consider it to be acceptable for app-bundles to be able to enumerate other app-bundles' entry points (with the exception that public interfaces may be enumerated). This implies that the implementation of get_implementations() (and the objects that it returns) must be done via IPC (most likely D-Bus) to a trusted service such as Didcot or Canterbury, which can read the .desktop files in XDG_DATA_DIRS/applications and apply appropriate filtering for the caller's limited view of the system.

Recommendation

For each application or agent (*entry point*) in an application bundle, we recommend that a freedesktop.org `.desktop` file is provided in a standard location such as `/var/lib/apertis_extensions/applications` by installing the application bundle. Possible implementations of this:

- The store publication process could verify that the contents of the provided `.desktop` file are appropriate for the application's manifest.
- The store publication process could generate a `.desktop` file from the application's manifest, with no control from the application author, other than to the extent that they can control the manifest and still have it approved by the app-store curator[11].

[11] https://sjoerd.pages.apertis.org/apertis-website/concepts/app_store_approval/

- The application manager could generate a `.desktop` file from the application's manifest during installation.

The resulting `.desktop` file should contain the standardized `Interfaces` key as described above.

This information should be made available to API users via a C API resembling GLib's GAppInfo[12] and GDesktopAppInfo[13] APIs, in particular g_desktop_app_info_get_implementations()[14]. However, we recommend an asynchronous version of that API in order to support the implementation being via D-Bus. Specifically, it should look something like this, with Namespace replaced by some suitable API namespace such as Didcot:

```
void namespace_app_registry_get_implementations_async (NamespaceAppRegistry *self, const gchar *interface_name, GCancellable *cancellable, GAsyncReadyCallback *callback, gpointer user_data);
/* Returns: (element-type GAppInfo) (transfer full): */ GList *namespace_app_registry_get_implementations_finish (NamespaceAppRegistry *self, GAsyncResult *result, GError **error);
```

where the result is a list of objects that implement the GAppInfo GInterface. If there is an order of preference, the most-preferred should come first. If there is no particular preference order, the implementation should use a predictable order, such as ordering by most-recently-used, most-recently-installed or alphabetically.

Either this could be implemented in terms of a D-Bus API, or it could have a D-Bus API based on it for access by non-C applications, for example:

```
/* returns a list of pairs (desktop file ID, text of .desktop file) */
org.apertis.Namespace1.GetImplementations(s interface_name) → a(ss)
```

For interfaces (API contracts) that already have a system-wide registration mechanism, such as Telepathy connection managers, D-Bus session services and systemd user services, we recommend adopting the existing mechanism instead, using appropriate subdirectories of `/var/lib/apertis_extensions` where necessary.

Selecting a preferred implementation

Some of the possible use-cases for interfaces benefit from the concept of a preferred implementation: for example, a navigation button should launch the preferred (default) navigation application, and if points-of-interest providers have a "push" model, they should not start non-preferred navigation applications in order to push points of interest into those implementations.

[12] https://developer.gnome.org/gio/stable/GAppInfo.html
[13] https://developer.gnome.org/gio/stable/gio-Desktop-file-based-GAppInfo.html
[14] https://developer.gnome.org/gio/stable/gio-Desktop-file-based-GAppInfo.html#g-desktop-app-info-get-implementations

For other use-cases, having a preferred implementation is unnecessary: for example, for a Sharing menu, global search, or points-of-interest providers with a "pull" model, the natural design is to query all known implementations in parallel, possibly excluding some that have been disabled.

We recommend addressing the question of a default/preferred implementation on a case-by-case basis (for example by introducing a platform setting for each interface that needs a preferred choice), and only developing a more general solution if experience demonstrates that it is needed in practice.

For example, a preferred navigation application could be selected with an API like

```
void namespace_app_registry_get_default_navigation_implementation_async (NamespaceAppRegistry *self, GCancellable *cancellable, GAsyncReadyCallback *callback, gpointer user_data); GAppInfo *namespace_app_registry_get_default_navigation_implementation_finish (NamespaceAppRegistry *self, GAsyncResult *result, GError **error);
```

if required.

The storage of preferred implementations should be considered to be an implementation detail of the platform component that implements this platform API. For example, it could have a GSetting for each well-known interface, whose value is the string app-ID (D-Bus well-known name) of the preferred entry-point, or an ordered list of preferred entry-points with the most-preferred first.

Enabling/disabling providers

If a provider is disabled system-wide, the platform component that implements interface discovery (for example Didcot) must behave as though it was not installed at all when answering queries from other components. The storage of enabled/disabled implementations can be considered to be an implementation detail of that component: for example it could store a string-list of disabled app IDs in GSettings. Uninstalling a provider should probably remove it from that list, so that reinstalling the provider automatically enables it.

If a provider is disabled for a particular consumer, we recommend that the consumer stores its own string-list of disabled app IDs, and filters the results of queries on the client-side, encapsulated in a library. This probably only makes sense for interfaces where the consumer will use all non-disabled implementations.

Restricting who can advertise a given interface

We recommend that interfaces advertised by a provider should be restricted by app-store curators[15], as follows:

[15] https://sjoerd.pages.apertis.org/apertis-website/concepts/app_store_approval/

- each ISV[16] that will publish apps on the app-store registers one or more reversed-DNS prefixes with the app-store curator as part of their app-developer account (for example, Collabora Ltd.[17] might register com.collabora and/or uk.co.collabora)
- the app-store curator verifies the ISV's ownership of the relevant domain names before accepting uploads from that ISV
- app-bundles published by the ISV may implement interface names in the namespace of those reversed-DNS prefixes without necessarily triggering extensive checking by the app-store curator (for example, Collabora Ltd. could publish an app-bundle implementing com.collabora.MyInterface)
- a whitelist of known-"safe" interface names in shared namespaces such as org.apertis, org.freedesktop and org.gnome could also be implemented without necessarily triggering extra checks by the app-store curator (for example, an org.apertis.SharingProvider interface which adds the app to the Sharing menu might be considered to be "safe" for anyone to implement)
- all other interface names would be "red flags" leading to rejection or additional checking by the app-store curator

This implies that cooperating ISVs cannot invent their own interfaces without app-store curators' involvement.

It is important to note that if the platform initially has this policy, it cannot be relaxed to "anyone may implement any interface" later. If it was, ISVs writing previously-correct code would potentially become susceptible to cross-app resource access attacks (for example, if the ISV owning example.net had assumed that every implementation of net.example.MyInterface was necessarily trusted code).

Communication between consumers and implementors

App-bundles that implement of public interfaces should receive AppArmor profiles allowing them to receive D-Bus method calls from anywhere.

App-bundles that do not implement public interfaces should receive AppArmor profiles that allow D-Bus method calls from platform services, and from other processes from the same app-bundle, but deny other D-Bus method calls.

Visibility of applications to other applications

App-bundles' AppArmor profiles should not give them read access to `/var/lib/apertis_extensions/applications` or to other app-bundles' manifests.

The implementation of the interface discovery should be done via D-Bus. The service providing this D-Bus API (for example Didcot) should be a platform component. It is considered to be a trusted component for the purposes of

[16] https://en.wikipedia.org/wiki/Independent_software_vendor
[17] https://www.collabora.com/

security between app-bundles: it must reveal public interface implementations to other app-bundles, but must only reveal non-public interface implementations to trusted platform components.