Content hand-over Design issues

Contents

Design issues for content hand-over

Creating new URI schemes (or not)

The W3C suggests avoiding creation of new URI schemes[1] but instead dispatching instead on the type of the representation (i.e. the content-type). Creating new content-types is substantially easier than it used to be[2] and is compatible with sending those content-types via http or email.

data:application/vnd.myapp.myobject,here-is-my-object is one way to pass small bits of content-typed data around as URIs, but we should be careful not to overuse inline data.

Ascribing semantics to a subtree of http, like Apple does for [https://developer.apple.com/library/ios/featuredarticles/iPhoneURLScheme_Reference/MapLinks/MapLinks.html#//apple_ref/doc/uid/TP40007899-CH5-SW1" map links] in recent iOS, is another way to avoid new URI schemes.

D-Bus or not?

D-Bus has some subtleties for how it interacts with AppArmor: the caller of a method, or the sender of a signal, needs permission to send to the recipient (in its profile), and the recipient needs permission to receive from the sender (in *its* profile). We are trying to avoid relying on fine-grained message filtering (e.g. by interface) because that won't work on kdbus. This means we might have to work out something clever here if we want pairs of arbitrary apps from different app-bundles to be able to interact directly: we'd need either some way to have "equivalence classes"of apps, or a trusted D-Bus proxy that rate-limits and filters messages.

[1] http://www.w3.org/TR/webarch/#URI-scheme
[2] https://www.iana.org/form/media-types

Another possibility would be for Didcot (or Canterbury) to proxy authorized requests between app-bundles; we could have the requester call a method on Didcot that results in Didcot calling o.fd.Application.Open or o.fd.Application.Activate[3] on the provider, if it considers the provider to be suitable.

This is fine for "launch"and "open URI", but not really up to the job for a more complex interface (search-provider-style) or for an interface with more data (search results coming back). We could potentially have an API through which we fd-pass a socket, or pass through an abstract socket by name, or something, and then do D-Bus over that; but then we lose total ordering of messages, and become sad.

In the non-D-Bus corner, we could use a mesh of 1-1 connections between apps; but then we have a mesh of 1-1 connections between apps, we've still lost total ordering, and we potentially need to reinvent message framing.

smcv's instinct here is to use D-Bus for everything that is a one-off action in response to something the user does; seriously consider using D-Bus for query-style APIs; and probably avoid D-Bus for TPEG, since that presumably already has framing, and we might end up doing the filtering navigation-app-side.

Can we use GDesktopAppInfo rather than reinventing it?

GAppInfo provides nice APIs for the basics of what Didcot does.

Unfortunately, it assumes that applications can be launched with direct D-Bus activation (not true if we are relying on Didcot for launching, unless we have additional infrastructure that helps us out) and that they have .desktop files.

Options:

- patch GDesktopAppInfo for Apertis
- write our own high-level API which will end up rather similar (potentially even identical), perhaps in its own LGPL-licensed library so it can copy bits from GDesktopAppInfo, and optionally patch GLib documentation to say not to use GDesktopAppInfo
- require that every app/agent is DBusActivatable, generate .desktop files from their manifests, and use a D-Bus proxy to transform activation requests into what we need
- generate .service files so that every app/agent *is* DBusActivatable, use a D-Bus proxy or clever AppArmor rules to make that work, and bypass Didcot entirely

Should Didcot and Canterbury be the same thing?

Canterbury is a trusted intermediary for launching apps. So is Didcot, if you think about it. Maybe they should merge?

[3] http://standards.freedesktop.org/desktop-entry-spec/1.1/ar01s08.html

Maintaining established communications

Do we have a use-case for this, that is not already satisfied by "list providers and start querying them again"?

Registration mechanism

It looks like we're going to need:

- I handle foo: (scheme)
- I handle everything in foo://bar (scheme + authority)
- I handle everything starting with foo://bar/baz/ (scheme + authority − path-prefix) (?)
- I open files of type foo/bar
- I open files of type foo/* (?)
- I open all files (?)
- I can share files of type foo/bar, foo/* (?), all files

and for the streaming-ish use cases:

- I implement org.apertis.PointOfInterestProvider (etc.)

This is, not coincidentally, a lot like .desktop files. We could make the encoding in the manifest somewhat similar, and auto-generate a .desktop file if desired. Something like this (using YAML here because it's easier to hand-write, apply the obvious mapping into JSON for production):

```
  entry_points:       com.example.MyApp:          content_types:              -
foo/bar           - foo/*        url_handler:           - "foo:"            -
"foo://bar"       - "foo://bar/baz/"    com.example.MyAgent:      implements:
- org.apertis.PointOfInterestProvider
```

Security considerations

URI scheme/media-type hijacking

It is important to note that URI schemes and media types will, in general, be a "first come, first served"shared resource. The Scheme Hijacking attack described in Unauthorized Cross-App Resource Access on MAC OS X and iOS[4] §3.4 relies on the attacker registering for a URI scheme that an app developer had (mis)used as a general IPC mechanism.

One thing we can do to improve on iOS'behaviour here is to provide IPC mechanisms that automatically convey app and user identity information that cannot be faked, such as D-Bus, and document them as a better way to solve the problems that iOS app developers are trying to solve by making up URI schemes.

For example, there should be a way for a pair of cooperating applications to declare in their app-bundle manifests that each may communicate with the other

[4] https://drive.google.com/file/d/0BxxXk1d3yyuZOFlsdkNMSGswSGs/view?pli=1

via D-Bus. Executables in the same app-bundle should just be able to do D-Bus to each other anyway, no questions asked.

Documentation for the content handover feature should recommend these other IPC mechanisms, and caution against using content handover to transfer credentials.

Bidirectional content handover

One feature that was considered is bidirectional content handover. We recommend treating this as out-of-scope: it requires thought to be put into security between apps, and in particular what we want to allow apps to do. If general bidirectional channels between pairs of apps are required, they should use a protocol such as D-Bus or an AF_UNIX socket, which provides secure authentication (credentials that cannot be faked, including the uid and AppArmor profile). ()

Terminating the launched app

Suppose app A launches app B via content handover. One design question that was considered was whether app A should be able to terminate app B.

We recommend that this capability is not offered: if it was misused, it would be easy for a user to misinterpret it as app B crashing. ()