Protocols and formats

- Working with DBus

Working with DBus

> This article is a work in progress. It'll likely be expanded over time, but for now it's incomplete.

What is DBus?

DBus is a standardized 'message bus' protocol that is mainly used on Linux. It serves to let different applications on the same system talk to each other through a standardized format, with a standardized way of specifying the available API.

Additionally, and this is probably the most-used feature, it allows for different applications to 'claim' specific pre-defined ("well-known") namespaces, if they intend to provide the corresponding service. For example, there are many different services that can show desktop notifications to the user, and the user may be using any one of them depending on their desktop environment, but whichever one it is, it will always claim the standard `org.freedesktop.Notifications` name.

That way, applications that want to *show* notifications don't need to know which specific notification service is running on the system - they can just send them to whoever claimed that name and implements the corresponding API.

How do you use DBus as a user?

As an end user, you don't really need to care about DBus. As long as a DBus daemon is running on your system (and this will be the case by default on almost every Linux distribution), applications using DBus should just work.

If you're curious, though, you can use a DBus introspection tool such as QDBusViewer or D-Spy to have a look at what sort of APIs the programs on your system provide. Just be careful not to send anything through it without researching it first - you can break things this way!

How do you use DBus as a developer?

You'll need a DBus protocol client. There are roughly two options:

1. Bindings to libdbus for the language you are using, or
2. A client implementation that's written directly in the language you are using (eg. `dbus-next` in JS)

You could also write your own client, as DBus typically just works over a local socket, but note that the serialization format is a little unusual, so it'll take some time to implement it correctly. Using an existing implementation is usually a better idea.

Note that you use a DBus *client* even when you want to *provide* an API over DBus; the 'server' in this arrangement is the DBus daemon, not your application.

How the protocol works

DBus implements a few different kinds of interaction mechanisms:

- **Properties:** These are (optionally read-only) values that can be read or written. They're usually used to *check* or *change* something.
- **Methods:** These are callable and can produce a result. They're usually used to *do* something.
- **Signals:** These are like events, and can be subscribed to. They're usually emitted when something *happens* on the other side.

All of these - properties, methods and signals - are addressable by pre-defined names. However, it takes a few steps to get there:

- First, you need to select a **bus name** - this is kind of like a process name (or, in the case of a "well-known" API, the standard name), although technically one process can present multiple bus names. Its components are delimited by dots.
- Then, on the resulting bus, you select an **object path** - essentially, this is the specific 'object' (or object *type*) within the process that you wish to access. Its components are delimited by slashes.
- Finally, on the selected object, you then select an **interface** - you can think of this as the 'service' that you wish to access. Custom DBus APIs often only implement a single interface, in addition to the standard DBus-specified interfaces for introspection (see below).

After these steps, you will end up with an interface that you can interact with - it has properties, methods, and/or signals. Don't worry too much about how exactly the hierarchy works here - the division between bus name, object path and interface can be (and in practice, is) implemented in many different ways depending on requirements, and if you merely wish to *use* a DBus API from some other application, you can simply specify whatever its documentation tells you for all of these values.

Some more information and context about this division can be found here, though keep in mind that you'll often encounter exactly one possible value for bus name, object path and interface, for any given application that exposes an API over DBus, so it's not required reading.

Introspection

An additional feature of DBus is that it allows introspection of DBus APIs; that is, you can use the DBus protocol itself to interrogate an API provider about its available API surface, the argument types, and so on. The details of this are currently not covered here.

Some well-known DBus APIs

- **MPRIS** - The 'media player control' API.
- **FreeDesktop Notifications** - The standard API for displaying desktop notifications.