Introduction to D-Bus

An introduction to the core concepts of D-Bus from an application developer's perspective.

Abstract

D-Bus is a free/open source inter-process communication (IPC) mechanism that is part of the freedesktop.org project. It is used in a wide range of applications and many freedesktop.org standards such as Desktop Notifications, media player control and XDG portals are built on it.

The term "IPC" can be used to describe methods of getting information from one process to another. This includes exchanging data, calling methods and listening to events.

The use cases for IPC on the desktop range from scripting (allowing a user to execute a script in a common environment to interact with or control various running applications), to providing access to centralized services, to coordination between multiple instances of cooperative applications.

A good example of using D-Bus is the freedesktop.org notifications specification. Applications send their notifications to a central server (e.g. Plasma) which then displays the notifications and sends back events such as the notification being closed or an action being invoked.

Another example of IPC usage is unique application support. When such an application is started it first checks for other running instances of the same application and if there is one it sends a message to the running instance via IPC to show itself and terminates.

D-Bus is language- and toolkit-agnostic and thus allows apps and services from all providers to interact. Qt provides a set of classes and tools to interact with D-Bus. This series of tutorials introduces the high-level concepts of D-Bus and their implementation in Qt and KDE software.

Buses

D-Bus provides multiple message "buses" for applications to use in communicating with each other. Each bus provides its own connection facilities, which allow for the separation of different categories of messages: messages sent on one bus can not be accessed from another bus, but applications connected to the same bus may all communicate with each other. Multiple applications can be connected to any given bus at any given time, and an application can be connected to multiple buses simultaneously. This allows different security policies to be applied to different buses while also allowing for the efficient sharing of both global and local messages.

D-Bus offers two predefined buses, which cover most of the typical D-Bus usage.

The **system bus** is used for system-global services such as hardware management. It is shared between users and usually comes with strict security policies.

Each desktop session (e.g. each logged in user) has an additional **session bus**, which is the one that desktop applications will tend to use most often.

Additionally, an application may create any number of own buses if necessary.

Messages

Messages form the base unit of communication on a bus. All information passed over the bus is done in the form of messages, similar to the way information transmitted using TCP/IP is done via packets. Unlike network packets, however, each D-Bus message is guaranteed to contain the entire set of data being sent or received. In addition to the data being sent, messages also record who the sender and intended receiver are to allow for proper routing.

Messages may be method calls, signal emissions or method return values and may contain error information.

Namespaces and Addresses

Since multiple applications can be on the same bus, and one application may provide multiple objects to which messages can be sent, it is necessary to have a means to effectively and unambiguously address any given object on any given bus, similar to the way a street address uniquely identifies any given residence or office. There are 3 pieces of information which, when taken together, create a unique address for any given object on a bus: interface, service and object name.

Interfaces

An **interface** is a set of callable methods and signals that are advertised on the bus. An interface provides a "contract" between the applications passing messages that defines the name, parameters (if any) and return values (if any) of the interface. These methods may not map directly in a one-to-one fashion to methods or API in the application that is advertising the interface, though they often do. This allows multiple applications to provide similar or the same interfaces, regardless of internal implementation, while allowing applications to use these interfaces without worrying about the internal design of the applications.

Interfaces can be described for documentation and code re-use purposes using XML. Not only can users and programmers reference the XML description of the interface, but developers can use classes that are auto-generated from the XML — making using D-Bus much easier and less error-prone (e.g. the compiler can check the syntax of messages at compile time).

Services

A **service** represents an application connection to a bus.

Service here corresponds to "well-known" Bus names in the D-Bus specification terminology. The term "Bus name" is a bit confusing. Regardless of how it sounds like, **Bus names** are the names of connections(!) on the bus, not names of buses(!). So here the term **service** will be used, as the Qt Documentation calls it.

These are kept unique by using a "reverse domain name" approach, as can be seen in many other systems that need to namespace for multiple components. Most services provided by applications from the KDE project itself use the `org.kde prefix` to their service name. So one may find `org.kde.screensaver` advertised on the session bus.

You should use the domain name for your organization or application for your service names. For example if your domain is `awesomeapps.org` and the name of your application is `wickedwidget` you would probably use `org.awesomeapps.wickedwidget` as the service name on the bus.

If an application has more than one connection to a bus, or if multiple instances of the same application may be active at once, it will need to use a unique service name for each connection. Often this is done by appending the process ID to the service name.

Objects

Of course, an application is likely to advertise access to more than one object on the bus. This many-to-one relationship between objects and services is accommodated by providing a **path** component to the address. Each path associated with a service represents a different, unique object. An example might be `/MainInterface` or `/Documents/Doc1`. The actual path structure is completely arbitrary and is up to the application providing the service as to what the paths should be. These paths simply provide a way to identify and logically group objects for applications that send messages to the application.

Some libraries export object paths with their "reverse domain" prepended to it, so as to properly namespace their objects. This is quite common for libraries and plugins that join an arbitrary service and must therefore avoid all clashing with objects exported by the application and other components. However, this practice is not in use in KDE applications and libraries.

Objects provide access to interfaces. In fact, a given object can provide access to multiple interfaces at the same time.

Putting it all together

A D-Bus message contains an address made up of all the above components so that it can be routed to the correct application, object and method call. Such an address might look like this:

```
org.kde.krunner /App org.kde.krunner.App.display
```

In this case `org.kde.krunner` is the service, `/App` is the path to the object, `org.kde.krunner.App` is the interface the object exports and `display` is a method in the interface. If the `/App` object only provides the `org.kde.krunner.App` interface (or the `display` method is unique amongst the services it implements) then this would work equally well as an address:

```
org.kde.krunner /App display
```

In this way each possible destination is uniquely and reliably addressable.

Calling and Being Called

Now that we have a way to address any given end point on the bus, we can examine the possibilities when it comes to actually sending or receiving messages.

Methods

Methods are messages that are sent to cause code to be executed in the receiving application. If the method is not available because, for instance, the address was wrong or the requested application is not running, an error will be returned to the calling application. If the method is successfully called, an optional return value will be returned to the calling application. Even if there is no return value provided, a success message will be returned. This round trip does have overhead, and it is important to keep this in mind for performance critical code.

Such method calls are always initiated by the calling application and the resulting messages have exactly one source and one destination address.

Signals

Signals are like method calls except that they happen in the "reverse" direction and are not tied to a single destination. A signal is emitted by the application which is exporting the interface and is available to any application on the same bus. This allows an application to spontaneously advertise changes in state or other events, to any applications which may be interested in tracking those changes.

If this sounds a lot like the signal and slots mechanism in Qt, that's because it is. For all intents and purposes it is a non-local version of the same functionality.

Useful Tools

There are several useful tools for exploring the D-Bus buses as well as developing applications that use D-Bus. We will now look briefly at end-user tools as the articles that follow cover the development tools in greater detail and context.

qdbus

`qdbus` is a command line tool which can be used to list the services, objects and interfaces on a given bus as well as send messages to a given address on the bus. It can be used to explore both the system and the default session bus. If the `--system` switch is passed, qdbus will connect to the system bus, otherwise it uses the session bus.

`qdbus` uses the rest of the supplied arguments on the command as an address and, if any, parameters to pass to a given object. If a full address is not supplied, then it lists all the objects available from that point on the bus. For instance, if no addresses are provided a list of available services is listed. If a service name is provided, object paths will be provided. If a path is also provided all methods in all interfaces will be listed. In this way one can quite easily explore and interact with objects on the bus, making `qdbus` very useful for testing, scripting and even idle exploration.

qdbusviewer

`qdbusviewer` is a Qt application that provides a graphical interface to essentially the same set of features that qdbus provides on the command line, thus providing a more user friendly mechanism to interact with the bus. `qdbusviewer` ships with Qt itself and is easy for anyone who is familiar with the basic D-Bus concepts, such as object addresses, to use.

Next
Accessing D-Bus Interfaces