Get on the D-BUS

Robert Love

Abstract

Programs, the kernel and even your phone can keep you in touch and make the whole desktop work the way you want. Here's how D-BUS works, and how applications are using it.

D-BUS is an interprocess communication (IPC) system, providing a simple yet powerful mechanism allowing applications to talk to one another, communicate information and request services. D-BUS was designed from scratch to fulfill the needs of a modern Linux system. D-BUS' initial goal is to be a replacement for CORBA and DCOP, the remote object systems used in GNOME and KDE, respectively. Ideally, D-BUS can become a unified and agnostic IPC mechanism used by both desktops, satisfying their needs and ushering in new features.

D-BUS, as a full-featured IPC and object system, has several intended uses. First, D-BUS can perform basic application IPC, allowing one process to shuttle data to another—think UNIX domain sockets on steroids. Second, D-BUS can facilitate sending events, or signals, through the system, allowing different components in the system to communicate and ultimately to integrate better. For example, a Bluetooth dæmon can send an incoming call signal that your music player can intercept, muting the volume until the call ends. Finally, D-BUS implements a remote object system, letting one application request services and invoke methods from a different object—think CORBA without the complications.

Why D-BUS Is Unique

D-BUS is unique from other IPC mechanisms in several ways. First, the basic unit of IPC in D-BUS is a message, not a byte stream. In this manner, D-BUS breaks up IPC into discrete messages, complete with headers (metadata) and a payload (the data). The message format is binary, typed, fully aligned and simple. It is an inherent part of the wire protocol. This approach contrasts with other IPC mechanisms where the lingua franca is a random stream of bytes, not a discrete message.

Second, D-BUS is bus-based. The simplest form of communication is process to process. D-BUS, however, provides a dæmon, known as the message bus dæmon, that routes messages between processes on a specific bus. In this fashion, a bus topology is formed, allowing processes to speak to one or more applications at the same time. Applications can send to or listen for various events on the bus.

A final unique feature is the creation of not one but two of these buses, the system bus and the session bus. The system bus is global, system-wide and runs at the system level. All users of the system can communicate over this bus with the proper permissions, allowing the concept of system-wide events. The session bus, however, is created during user login and runs at the user, or session, level. This bus is used solely by a particular user, in a particular login session, as an IPC and remote object system for the user's applications.

D-BUS Concepts

Messages are sent to objects. Objects are addressed using path names, such as /org/cups/printers/queue. Processes on the message bus are associated with objects and implemented interfaces on that object.

D-BUS supports multiple message types, such as signals, method calls, method returns and error messages. Signals are notification that a specific event has occurred. They are simple, asynchronous, one-way heads-up messages. Method call messages allow an application to request the invocation of a method on a remote object. Method return messages provide the return value resulting from a method invocation. Error messages provide exceptions in response to a method invocation.

D-BUS is fully typed and type-safe. Both a message's header and payload are fully typed. Valid types include byte, Boolean, 32-bit integer, 32-bit unsigned integer, 64-bit integer, 64-bit unsigned integer, double-precision floating point and string. A special array type allows for the grouping of types. A DICT type allows for dictionary-style key/value pairs.

D-BUS is secure. It implements a simple protocol based on SASL profiles for authenticating one-to-one connections. On a bus-wide level, the reading of and the writing to messages from a specific interface are controlled by a security system. An administrator can control access to any interface on the bus. The D-BUS dæmon was written from the ground up with security in mind.

Why D-BUS?

These concepts make nice talk, but what is the benefit? First, the system-wide message bus is a new concept. A single bus shared by the entire system allows for propagation of events, from the kernel (see The Kernel Event Layer sidebar) to the uppermost applications on the system. Linux, with its well-defined interfaces and clear separation of layers, is not very integrated. D-BUS' system message bus improves integration without compromising fine engineering practices. Now, events such as disk full and printer queue empty or even battery power low can bubble up the system stack, available for whatever application cares, allowing the system to respond and react. The events are sent asynchronously, and without polling.

The Kernel Event Layer

The Kernel Event Layer is a kernel-to-user communication mechanism that uses a high-speed netlink socket to communicate asynchronously with user space. This mechanism can be tied into D-BUS, allowing the kernel to send D-BUS signals!

The Kernel Event Layer is tied to sysfs, the tree of kobjects that lives at /sys on modern Linux systems. Each directory in sysfs is tied to a kobject, which is a structure in the kernel used to represent objects; sysfs is an object hierarchy exported as a filesystem.

Each Kernel Event Layer event is modeled as though it originated from a sysfs path. Thus, the events appear as if they emit from kobjects. The sysfs paths are easily translatable to D-BUS paths, making the Kernel Event Layer and D-BUS a natural fit. This Kernel Event Layer was merged into the 2.6.10-rc1 kernel.

Second, the session bus provides a mechanism for IPC and remote method invocation, possibly providing a unified system between GNOME and KDE. D-BUS aims to be a better CORBA than CORBA and a better DCOP than DCOP, satisfying the needs of both projects while providing additional features.

And, D-BUS does all this while remaining simple and efficient.

Adding D-BUS Support to Your Application

The core D-BUS API, written in C, is rather low-level and large. On top of this API, bindings integrate with programming languages and environments, including Glib, Python, Qt and Mono. On top of providing language wrappers, the bindings provide environment-specific features. For example, the Glib bindings treat

D-BUS connections as GObjects and allow messaging to integrate into the Glib mainloop. The preferred use of D-BUS is definitely using language and environment-specific bindings, both for ease of use and improved functionality.

Let's look at some basic uses of D-BUS in your application. We first look at the C API and then poke at some D-BUS code using the Glib interface.

The D-BUS C API

Using D-BUS starts with including its header:

```
#include <dbus/dbus.h>
```

The first thing you probably want to do is connect to an existing bus. Recall from our initial D-BUS discussion that D-BUS provides two buses, the session and the system bus. Let's connect to the system bus:

```
DBusError error;
DBusConnection *conn;

dbus_error_init (&error);
conn = dbus_bus_get (DBUS_BUS_SYSTEM, &error);
if (!conn) {
    fprintf (stderr, "%s: %s\n",
             err.name, err.message);
    return 1;
}
```

Connecting to the system bus is a nice first step, but we want to be able to send messages from a well-known address. Let's acquire a service:

```
dbus_bus_acquire_service (conn, "org.pirate.parrot",
                          0, &err);
if (dbus_error_is_set (&err)) {
    fprintf (stderr, "%s: %s\n",
             err.name, err.message);
    dbus_connection_disconnect (conn);
    return;
}
```

Now that we are on the system bus and have acquired the org.pirate.parrot service, we can send messages originating from that address. Let's send a signal:

```
DBusMessage *msg;
DBusMessageIter iter;

/* create a new message of type signal */
msg = dbus_message_new_signal(
          "org/pirate/parrot/attr",
          "org.pirate.parrot.attr", "Feathers");

/* build the signal's payload up */
dbus_message_iter_init (msg, &iter);
dbus_message_iter_append_string (&iter, "Shiny");
dbus_message_iter_append_string (&iter,
```

```
                                    "Well Groomed");

/* send the message */
if (!dbus_connection_send (conn, msg, NULL))
        fprintf (stderr, "error sending message\n");

/* drop the reference count on the message */
dbus_message_unref (msg);

/* flush the connection buffer */
dbus_connection_flush (conn);
```

This sends the Feathers signal from org.pirate.parrot.attr with a payload consisting of two fields, each strings: Shiny and Well Groomed. Anyone listening on the system message bus with sufficient permissions can subscribe to this service and listen for the signal.

Disconnecting from the system message bus is a single function:

```
if (conn)
        dbus_connection_disconnect (conn);
```

The Glib Bindings

Glib (pronounced gee-lib) is the base library of GNOME. It is on top of Glib that Gtk+ (GNOME's GUI API) and the rest of GNOME is built. Glib provides several convenience functions, portability wrappers, a family of string functions and a complete object and type system—all in C.

The Glib library provides an object system and a mainloop, making object-based, event-driven programming possible, even in C. The D-BUS Glib bindings take advantage of these features. First, we want to include the right header files:

```
#include <dbus/dbus.h>
#include <dbus/dbus-glib.h>
```

Connecting to a specific message bus with the Glib bindings is easy:

```
DBusGConnection *conn;
GError *err = NULL;

conn = dbus_g_bus_get (DBUS_BUS_SESSION, &err);
if (!conn) {
        g_printerr ("Error: %s\n", error->message);
        g_error_free (error);
}
```

In this example, we connected to the per-user session bus. This call associates the connection with the Glib mainloop, allowing multiplexed I/O with the D-BUS messages.

The Glib bindings use the concept of proxy objects to represent instantiations of D-BUS connections associated with specific services. The proxy object is created with a single call:

```
DBusGProxy *proxy;

proxy = dbus_g_proxy_new_for_service (conn,
```

```
                                "org.fruit.apple",
                                "org/fruit/apple",
                                "org.fruit.apple");
```

This time, instead of sending a signal, let's execute a remote method call. This is done using two functions. The first function invokes the remote method; the second retrieves the return value.

First, let's invoke the Peel remote method:

```
DBusGPendingCall *call;

call = dbus_g_proxy_begin_call (proxy,
                    "Peel", DBUS_TYPE_INVALID);
```

Now let's retrieve-check for errors and retrieve the results of the method call:

```
GError *err = NULL;
int ret;

if (!dbus_g_proxy_end_call (proxy, call,
                            &err, DBUS_TYPE_INT32,
                            &ret, DBUS_TYPE_INVALID)) {
        g_printerr ("Error: %s\n", err->message);
        g_error_free (err);
}
```

The Peel function accepts a single parameter, an integer. If this call returned nonzero, it succeeded, and the variable ret holds the return value from this function. The data types that a specific method accepts are determined by the remote method. For example, we could not have passed DBUS_TYPE_STRING instead of DBUS_TYPE_INT32.

The main benefit of the Glib bindings is mainloop integration, allowing developers to manage multiple D-BUS messages intertwined with other I/O and UI events. The header file <dbus/dbus-glib.h> declares multiple functions for connecting D-BUS to the Glib mainloop.

Conclusion

D-BUS is a powerful yet simple IPC system that will improve, with luck, the integration and functionality of Linux systems. Users are encouraged to investigate new D-BUS utilizing applications. With this article in hand, D-BUS shouldn't be a scary new dependency, but a shining new feature. The on-line Resources list some interesting applications that use D-BUS. Developers are encouraged to investigate implementing D-BUS support in their applications. There are also some Web sites that provide more information on using D-BUS. Of course, the best reference is existing code, and thankfully there is plenty of that.

Resources for this article: http://www.linuxjournal.com/article/7926.