Connect desktop apps using D-BUS

Helping applications talk to one another

Ross Burton (r.burton@180sw.com), Software Engineer, OneEighty Software, Ltd.

Summary: D-BUS is an up-and-coming message bus and activation system that is set to achieve deep penetration in the Linux® desktop. Learn why it was created, what it can be used for, and where it is going.

Date: 27 Jul 2004
Level: Intermediate

Comments: 3 (View | Add comment - Sign in)

★★★★☆ Average rating (105 votes)
Rate this article

D-BUS is essentially an implementation of *inter-process communication* (IPC). However, several features distance D-BUS from the stigma of being "Yet Another IPC Implementation." There are many different IPC implementations because each aims to solve a particular well-defined problem. CORBA is a powerful solution for complex IPC in object-orientation programming. DCOP is a lighter IPC framework with less power, but is well integrated into the K Desktop Environment. SOAP and XML-RPC are designed for Web services and therefore use HTTP as the transport protocol. D-BUS was designed for desktop application and OS communication.

Desktop application communication

The typical desktop has multiple applications running, and they often need to talk to each other. DCOP is a solution for KDE, but is tied to Qt and so is not used in other desktop environments. Similarly, Bonobo is a solution for GNOME, but is quite heavy, being based on CORBA. It is also tied to GObject, so it is not used outside of GNOME. D-BUS aims to replace DCOP and Bonobo for simple IPC and to integrate these two desktop environments. Because the dependencies for D-BUS are kept as small as possible, other applications that would like to use D-BUS don't have to worry about bloating dependencies.

Desktop/Operating System communication

The term "operating system" here includes not only the kernel but also the system daemons. For example, with a D-BUS-enabled `udev` (the Linux 2.6 replacement for `devfs`, providing dynamic /dev directories), a signal is emitted when a device (such as a USB camera) is inserted. This allows for tighter integration with the hardware in the desktop, leading to an improved user experience.

D-BUS features

D-BUS has several interesting features that make it look like a very promising candidate.

The protocol is low-latency and low-overhead, designed to be small and efficient to minimize round-trips. In addition, the protocol is binary, not textual, which removes the costly serialization process. The use cases are biased towards processing on the local machine, so all messages are sent in the native byte ordering. The byte ordering is stated in each message, so if a D-BUS message travels over a network to a remote host, it can still be parsed correctly.

D-BUS is easy to use from a developer's point of view. The wire protocol is simple to understand, and the client library wraps it in an intuitive manner.

The library has also been designed to be wrapped by other systems. It is expected that GNOME will create wrappers around D-BUS using GObject (indeed these partially exist, integrating D-BUS into their event loop), and that KDE will create similar wrappers using Qt. There is already a Python wrapper that has a much simpler interface, due to Python's object-orientation and flexible typing.

Finally, D-BUS is being developed under the umbrella of freedesktop.org, where interested members from GNOME, KDE, and elsewhere participate in the design and implementation.

The inner workings of D-BUS

A typical D-BUS setup will consist of several buses. There will be a persistent *system bus*, which is started at boot time. This bus is used by the operating system and daemons and is tightly secured so that arbitrary applications cannot spoof system events. There will also be many *session buses*, which are started when a user logs in and are private to that user. It is a session bus that the user's applications will use to communicate. Of course, if an application wants to receive messages from the system bus, it can connect to it as well -- but the messages it can send will be restricted.

Once applications are connected to a bus, they have to state which messages they would like to receive by adding *matchers*. Matchers specify a set of rules for messages that will be received based on interfaces, object paths, and methods (see below). This enables applications to concentrate on handling what they want to handle, to allow efficient routing of messages, and to keep the anticipated multitude of messages across buses from grinding the performance of all of the applications down to a crawl.

Objects

At its heart, D-BUS is a peer-to-peer protocol -- every message has a source and a destination. These addresses are specified as *object paths*. Conceptually, all applications that use D-BUS contain a set of *objects*, and messages are sent to and from specific objects -- not applications -- that are identified by an object path.

Additionally, every object can support one or more *interfaces*. These interfaces appear at first to be similar to interfaces in Java or pure virtual classes in `C++`. However, there is not an option to check if objects implement the interfaces they claim to implement, and there is no way of introspecting an object to list the interfaces it supports. Interfaces are used to namespace the method names, so a single object can have multiple methods with the same name but with different interfaces.

Messages

There are four types of messages in D-BUS: method calls, method returns, signals, and errors. To perform a method on a D-BUS object, you send the object a method call message. It will do some processing and return either a method return message or an error message. Signals are different in that they cannot return anything: there is neither a "signal return" message, nor any other type of error message.

Messages can also have arbitrary arguments. Arguments are strongly-typed, and the types range from fundamental primitives (booleans, bytes, integers) to high-level structures (strings, arrays, and dictionaries).

Services

Services are the highest level of abstraction in D-BUS, and their implementation is currently in flux. An application can register a service with a bus, and if it succeeds, the application has *acquired* the service. Other applications can check whether a particular service exists on the bus and can ask the bus to start it if it doesn't. The details of the service abstraction -- particularly service activation -- are under development at the moment and are liable to change.

Use cases

Even though D-BUS is relatively new, it has been adopted very quickly. As I mentioned earlier, `udev` can be built with D-BUS support so that it sends a signal when a device is hot-plugged. Any application can listen to these events and perform actions when they are received. For example, gnome-volume-manager can detect the insertion of a USB memory stick and automatically mount it; or, it can automatically download photos when a digital camera is plugged in.

A more amusing but far less useful example is the combination of Jamboree and Ringaling. Jamboree is a simple music player that has a D-BUS interface so that it can be told to play, go to the next song, change the volume, and so on. Ringaling is a small program that opens /dev/ttyS0 (a serial port) and watches what is received. When Ringaling sees the text "RING," it uses D-BUS to tell Jamboree to turn down the volume. The net result is that if you have a modem plugged into your computer and your phone rings, the music is turned down for you. *This* is what computers are for!

Code examples

Now, let's walk through a few example uses of D-BUS code.

dbus-ping-send.c sends a signal over the session bus every second with the string "Ping!" as an argument. I'm using GLib to manage the bus so that I don't need to deal with the details of the bus connection myself.

Listing 1. dbus-ping-send.c

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

static gboolean send_ping (DBusConnection *bus);

int
main (int argc, char **argv)
{
  GMainLoop *loop;
  DBusConnection *bus;
  DBusError error;
  /* Create a new event loop to run in */
  loop = g_main_loop_new (NULL, FALSE);

  /* Get a connection to the session bus */
  dbus_error_init (&error);
  bus = dbus_bus_get (DBUS_BUS_SESSION, &error);
  if (!bus) {
    g_warning ("Failed to connect to the D-BUS daemon: %s", error.message);
    dbus_error_free (&error);
    return 1;
  }

  /* Set up this connection to work in a GLib event loop */
  dbus_connection_setup_with_g_main (bus, NULL);
  /* Every second call send_ping() with the bus as an argument*/
  g_timeout_add (1000, (GSourceFunc)send_ping, bus);

  /* Start the event loop */
  g_main_loop_run (loop);
  return 0;
}

static gboolean
send_ping (DBusConnection *bus)
{
  DBusMessage *message;

  /* Create a new signal "Ping" on the "com.burtonini.dbus.Signal" interface,
   * from the object "/com/burtonini/dbus/ping". */
  message = dbus_message_new_signal ("/com/burtonini/dbus/ping",
                                     "com.burtonini.dbus.Signal", "Ping");
  /* Append the string "Ping!" to the signal */
  dbus_message_append_args (message,
                            DBUS_TYPE_STRING, "Ping!",
                            DBUS_TYPE_INVALID);
  /* Send the signal */
  dbus_connection_send (bus, message, NULL);
  /* Free the signal now we have finished with it */
  dbus_message_unref (message);
  /* Tell the user we send a signal */
  g_print("Ping!\n");
  /* Return TRUE to tell the event loop we want to be called again */
  return TRUE;
}
```

The `main` function creates a GLib event loop, gets a connection to the session bus, and integrates the D-BUS event handling into the Glib event loop. Then it creates a one-second timer that calls `send_ping`, and starts the event loop.

`send_ping` constructs a new Ping signal, coming from the object path /com/burtonini/dbus/ping and interface `com.burtonini.dbus.Signal`. Then the string "Ping!" is added as an argument to the signal and sent across the bus. A message is printed on standard output to let the user know a signal was sent.

Of course, it is not good to fire signals down the bus if there is nothing listening to them... which brings us to:

Listing 2. dbus-ping-listen.c

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

static DBusHandlerResult signal_filter
      (DBusConnection *connection, DBusMessage *message, void *user_data);

int
main (int argc, char **argv)
{
  GMainLoop *loop;
  DBusConnection *bus;
  DBusError error;

  loop = g_main_loop_new (NULL, FALSE);
  dbus_error_init (&error);
  bus = dbus_bus_get (DBUS_BUS_SESSION, &error);
  if (!bus) {
    g_warning ("Failed to connect to the D-BUS daemon: %s", error.message);
    dbus_error_free (&error);
    return 1;
  }
  dbus_connection_setup_with_g_main (bus, NULL);

  /* listening to messages from all objects as no path is specified */
  dbus_bus_add_match (bus, "type='signal',interface='com.burtonini.dbus.Signal'");
  dbus_connection_add_filter (bus, signal_filter, loop, NULL);

  g_main_loop_run (loop);
  return 0;
}

static DBusHandlerResult
signal_filter (DBusConnection *connection, DBusMessage *message, void *user_data)
{
  /* User data is the event loop we are running in */
  GMainLoop *loop = user_data;

  /* A signal from the bus saying we are about to be disconnected */
  if (dbus_message_is_signal
        (message, DBUS_INTERFACE_ORG_FREEDESKTOP_LOCAL, "Disconnected")) {
    /* Tell the main loop to quit */
    g_main_loop_quit (loop);
    /* We have handled this message, don't pass it on */
    return DBUS_HANDLER_RESULT_HANDLED;
  }
  /* A Ping signal on the com.burtonini.dbus.Signal interface */
  else if (dbus_message_is_signal (message, "com.burtonini.dbus.Signal", "Ping")) {
    DBusError error;
    char *s;
    dbus_error_init (&error);
    if (dbus_message_get_args
       (message, &error, DBUS_TYPE_STRING, &s, DBUS_TYPE_INVALID)) {
      g_print("Ping received: %s\n", s);
      dbus_free (s);
    } else {
      g_print("Ping received, but error getting message: %s\n", error.message);
      dbus_error_free (&error);
    }
    return DBUS_HANDLER_RESULT_HANDLED;
  }
  return DBUS_HANDLER_RESULT_NOT_YET_HANDLED;
}
```

This program listens for the signals dbus-ping-send.c is emitting. The `main` function starts as before, creating a connection to the bus. Then it states that it would like to be notified when signals with the `com.burtonini.dbus.Signal` interface are sent, sets `signal_filter` as the notification function, and enters the event loop.

`signal_func` is called when a message that meets the matches is sent. However, it will also receive bus management signals from the bus itself. Deciding what to do when a message is received is a simple case of examining the message header. If the message is a bus disconnect signal, the event loop is terminated, as there is no point in listening to a non-existent bus. (The bus is told that the signal was handled). Next, the incoming message is compared to the message we are expecting, and, if successful, the argument is extracted and output. If the incoming message is neither of those, the bus is told that we did not handle the message.

Those two examples used the low-level D-BUS library, which is complete but can be long-winded to use when you want to create services and many objects. This is where the higher-level bindings come in. There are C# and Python wrappers in development that present a programming interface far closer to the logical model of D-BUS. As an example, here is a more sophisticated reworking of the ping/listen example in Python. Because the Python bindings model the logical interface, it is not possible to send a signal without it coming from a service. So this example also creates a service:

Listing 3. dbus-ping-send.py

```
#! /usr/bin/env python

import gtk
import dbus

# Connect to the bus
bus = dbus.Bus()

# Create a service on the bus
service = dbus.Service("com.burtonini.dbus.SignalService", bus)

# Define a D-BUS object
class SignalObject(dbus.Object):
    def __init__(self, service):
        dbus.Object.__init__(self, "/", [], service)

# Create an instance of the object, which is part of the service
signal_object = SignalObject(service)

def send_ping():
    signal_object.broadcast_signal("com.burtonini.dbus.Signal", "Ping")
    print "Ping!"
    return gtk.TRUE

# Call send_ping every second to send the signal
gtk.timeout_add(1000, send_ping)
gtk.main()
```

Most of the code is self-explanatory: a connection to the bus is obtained, and the service `com.burtonini.dbus.SignalService` is registered. Then a minimal D-BUS object is created and every second a signal is broadcast from the object. This code is clearer than the corresponding `C code`, but the Python bindings still need work. (For instance, there is no way to add arguments to signals.)

Listing 4. dbus-ping-listen.py

```
#! /usr/bin/env python

import gtk
import dbus

bus = dbus.Bus()

def signal_callback(interface, signal_name, service, path, message):
    print "Received signal %s from %s" % (signal_name, interface)

# Catch signals from a specific interface and object, and call signal_callback
# when they arrive.
bus.add_signal_receiver(signal_callback,
                        "com.burtonini.dbus.Signal", # Interface
                        None, # Any service
                        "/" # Path of sending object
                        )

# Enter the event loop, waiting for signals
gtk.main()
```

This code is more concise than the equivalent `C code` in dbus-ping-listen.c and is easier to read. Again, there are areas where the bindings still need work (when calling `bus.add_signal_receiver`, the user must pass in an interface and an object path; otherwise, malformed matchers are created). This is a trivial bug, and once it is fixed, the service and object path arguments could be removed, improving the readability of the code even more.

Conclusion

D-BUS is a lightweight yet powerful remote procedure call system with minimal overhead costs for the applications that wish to use it. D-BUS is under active public development by a group of very experienced programmers. Acceptance of D-BUS by early adopters is rapid, so it appears to have a rosy future on the Linux desktop.

Resources

- You'll find info, downloads, documentation, and more at the D-BUS home page.
- D-BUS is developed as part of freedesktop.org/.
- CORBA is a powerful, standardized remote procedure call specification.
- ORBit is the CORBA implementation used in GNOME. (The GNOME component system Bonobo is built on top of ORBit).
- KDE's remote procedure call implementation is DCOP.
- Project Utopia aims to build seamless integration of hardware in Linux and uses D-BUS to achieve this.
- Previously for developerWorks, Ross wrote Wrap GObjects in Python (developerWorks, March 2003), which shows how you can use a C-coded GObject in Python whenever you like, whether or not you're especially proficient in `C`.
- The tutorials Bridging XPCOM/Bonobo: Techniques (developerWorks, May 2001) and Bridging XPCOM/Bonobo: Implementation (developerWorks, May 2001) discuss concepts and techniques required for bridging two component architectures so that the components from one architecture can be used in another environment.
- Connect KDE applications using DCOP (developerWorks, February 2004) introduces KDE's inter-process communication protocol and how to script it.
- Synchronizing processes and threads (developerWorks, October 2001) looks at inter-process synchronization primitives as a way to control two processes' access to the same resource.
- CORBA Component Model (CCM) outlines the CORBA specification and CORBA interoperability with other component models.
- Find more resources for Linux developers in the developerWorks Linux zone.
- Browse for books on these and other technical topics.
- Download no-charge trial versions of selected developerWorks Subscription products that run on Linux, including WebSphere Studio Site Developer, WebSphere SDK for Web services, WebSphere Application Server, DB2 Universal Database Personal Developers Edition, Tivoli Access Manager, and Lotus Domino Server, from the Speed-start your Linux app section of developerWorks. For an even speedier start, help yourself to a product-by-product collection of how-to articles and tech support.

About the author

Ross Burton is an average computer science graduate who by day codes Java and embedded systems. By night, to get away from the horrors, he prefers Python, C, and GTK+. You can contact Ross at r.burton@180sw.com.

information at any time. Your first name, last name (unless you choose to hide them), and display name will accompany the content that you post.

All information submitted is secure.

Close [x]

Choose your display name

The first time you sign in to developerWorks, a profile is created for you, so you need to choose a display name. Your display name accompanies the content you post on developerWorks.

Please choose a display name between 3-31 characters. Your display name must be unique in the developerWorks community and should not be your email address for privacy reasons.

Display name: (Must be between 3 – 31 characters.)

By clicking **Submit**, you agree to the developerWorks terms of use.

Submit Cancel

All information submitted is secure.

★★★★☆ Average rating (105 votes)

○ 1 star ★☆☆☆☆ 1 star
○ 2 stars ★★☆☆☆ 2 stars
○ 3 stars ★★★☆☆ 3 stars
○ 4 stars ★★★★☆ 4 stars
○ 5 stars ★★★★★ 5 stars

Submit

Add comment:

Sign in or register to leave a comment.

Note: HTML elements are not supported within comments.

☐ Notify me when a comment is added1000 characters left

Post

Total comments (3)

Here is a patch for dbus-ping-listen.c for newer versions of dbus:

```
2c2
< #include <dbus/dbus-glib.h>
---
> #include <dbus/dbus.h>
26c26
< dbus_bus_add_match (bus, "type='signal',interface='com.burtonini.dbus.Signal'");
---
> dbus_bus_add_match (bus, "type='signal',interface='com.burtonini.dbus.Signal'",&error);
41c41
< (message, DBUS_INTERFACE_ORG_FREEDESKTOP_LOCAL, "Disconnected")) {
```

```
---
> (message, "org.freedesktop.Local", "Disconnected")) {
55d54
< dbus_free (s);
```

And to compile these programs:

```
gcc $(pkg-config --libs --cflags glib-2.0) $(pkg-config --libs --cflags dbus-glib-1) $(pkg-config --libs --cflags dbus-1) dbus-ping-listen.c -o dbus-ping-listen

gcc $(pkg-config --libs --cflags glib-2.0) $(pkg-config --libs --cflags dbus-glib-1) $(pkg-config --libs --cflags dbus-1) dbus-ping-send.c -o dbus-ping-send
```

Posted by **chadl** on 22 February 2011

Report abuse

Here are a couple patches to update the C files to work with new versions of dbus:

```
--- dbus-ping-send.c.old 2011-02-21 21:54:01.543097200 -0500
+++ dbus-ping-send.c 2011-02-21 23:59:57.901601960 -0500
@@ -1,5 +1,5 @@
#include <glib.h>
-#include <dbus/dbus-glib.h>
+#include <dbus/dbus.h>

static gboolean send_ping (DBusConnection *bus);

@@ -42,8 +42,9 @@
message = dbus_message_new_signal ("/com/burtonini/dbus/ping",
"com.burtonini.dbus.Signal", "Ping");
/* Append the string "Ping!" to the signal */
+ char *str_ping = "Ping!";
dbus_message_append_args (message,
- DBUS_TYPE_STRING, "Ping!",
+ DBUS_TYPE_STRING, &str_ping,
DBUS_TYPE_INVALID);
/* Send the signal */
dbus_connection_send (bus, message, NULL);
```

Posted by **chadl** on 21 February 2011

Report abuse

These programs do not compile. dbus-ping-send.c is missing the following include:

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

Once added, the compiler reports:

```
undefined reference to dbus_connection_setup_with_g_main
```

Posted by **pys** on 30 December 2010

Report abuse

Industries

Integrated Service Management

Students and faculty

Business partners

Accessibility