www.photocase.com

Interprocess communication with D-Bus and HAL

FAST TRAFFIC

It's the end of the line for CORBA! Gnome now relies on the D-Bus messaging system, and KDE is in the process of migrating.

BY OLIVER FROMMEL

Nobody likes applications that spend all day sitting in the corners of the desktop and refusing to talk to anything else. The least you might expect would be for them to exchange data with other desktop residents using simple drag & drop techniques. But many users expect their programs to demonstrate more advanced communication skills at all levels. Of course, users want pluggable USB disks no matter what program they are using. And VOIP softphones should make friends with the new hardware when you change the headset without demanding a reboot.

To allow this to happen, a Linux system needs a communication system that lets desktop applications talk to one another and to the underlying levels right down through the kernel to the hardware. And if the Freedesktop developers have any say in the matter, D-Bus [1], which relies on Hardware Abstraction Layer HAL [2], will be the communication system for future generations of Linux.

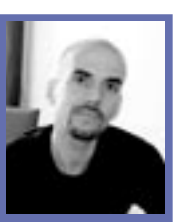
Talk to Me

D-Bus is an Inter Process Communication (IPC) system; that is, it provides the infrastructure that lets applications talk to one another and to parts of the operating system. Although IPC mechanisms were introduced to Unix years ago, they are restricted to signals, pipes, and similar things.

This may sound familiar to you; after all, competitive approaches have been around right from the beginning – just think of CORBA, Microsoft's DCOM, or hundreds of other projects. Both KDE and Gnome have experimented with their own CORBA implementations from the outset. KDE introduced its own DCOP system, and Gnome's CORBA legacy is evidenced by the Bonobo component system. Regardless of their personal opinions of CORBA, most developers who just want to code a desktop application are overtaxed by the system. And this possibly explains why Bonobo has been vegetating behind the Gnome scenes for so long.

D-Bus is designed as a simple system with a small footprint. The basic Lib-

THE AUTHOR



Oliver spent several years as a sysop and programmer at Ars Electronica Center in Linz/Austria. After finishing his studies in Philosophy, Linguistics and Computer Science, he became an editor for the Bavarian Broadcasting Corporation. Today he is head of the Editorial Competence Center for Software and Programming at Linux New Media AG.

dbus library simply provides functions that allow two applications to communicate. Application developers don't typically use the library, preferring, instead, the Glib-API-based Libdbus-Glib, which provides an object-oriented C-API. It is at this level that D-Bus's capabilities extend to provide a genuine bus system living up to its name. The server process, *dbus-daemon*, runs in the background and listens for connection requests from applications that register for various event types, such as plugging and unplugging specific hardware. When the event occurs, the D-Bus daemon sends a message along the bus, and the applications respond accordingly.

System Global or Per Session

On systems that use D-Bus, each server process implements two buses: the system bus and the session bus. The system bus launches at boot time and keeps on running even if no users are logged on to the system. When a user runs the GUI-based login manager to launch a desktop session, a server process for the session bus is launched. The *dbus-daemon* binary has command line parameters for both modes: *--system* or *--session*. The D-Bus package includes the *dbus-launch* for starting the daemon and setting the required environmental variables. Most distributions launch the session mode D-Bus daemon along with the X session.

Figure 1 shows the role the two buses play in communications between the operating system components. The session bus lets applications belonging to a desktop session talk to each other. Of course, these applications can be services provided by the desktop environment. In contrast to this, the system bus mainly ensures that desktop programs can talk to the underlying layers. For example, an application can use the system bus to register for a specific hardware class, such as, say, digital cameras.

Hardware Management with HAL

D-Bus does not provide its own hardware management; instead it relies on the Hardware Abstraction Layer, HAL. Although HAL is independent of D-Bus, the two components work hand in hand: HAL uses D-Bus to provide services, and D-Bus was mainly programmed for HAL.

Besides the kernel, modern distributions use the Udev subsystem for user-space hardware management. As of version 0.59, Udev replaces the hotplug system, which has only recently established itself as a mechanism for supporting pluggable hardware by running */sbin/hotplug*. Besides kernel and Udev information, HAL now has additional details on devices stored as FDI files (Device Information Files) in an XML format. Listing 1 shows a section from an FDI file for a digital camera.

Gnome NetworkManager [3] is a good example of how components cooperate. It uses the HAL daemon to monitor the network subsystem. When changes occur, such as a user plugging in or unplugging a wireless USB stick, the daemon uses D-Bus to notify NetworkManager. Besides genuine devices, HAL can also handle filesystems; it has the ability to identify filesystem types, including LUKS-encrypted partitions [4]. In Gnome, HAL now handles the lion's share of the hardware management, particularly hot-pluggable devices. The *gnome-volume-manager* process, which runs in the background, makes this happen; To configure the process, Gnome users run the *gnome-volume-properties* front-end (Figure 2).

There is also a front-end for HAL that gives users a tree view of the attached devices (Figure 3). Fedora users will find the *hal-device-manager* hidden away in the *hal-gnome* package.

Using D-Bus

The D-Bus protocol specification is available from [1]. The protocol defines four message types that users can send over the bus. For example, one application can call the methods provided by an-

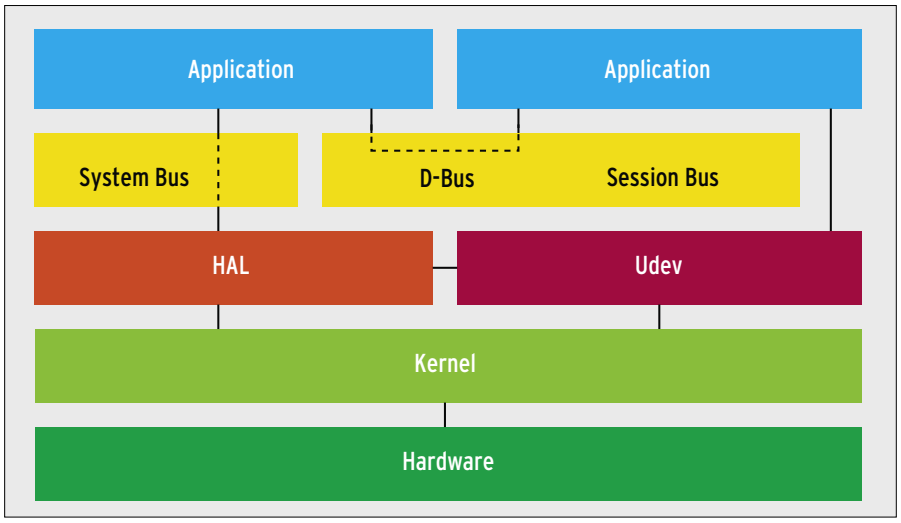


Figure 1: D-Bus and HAL in the overall context of a Linux system's components. Applications use D-Bus to query HAL about the underlying hardware.

Listing 1: 10-camera-ptp.fdi

```
01 <deviceinfo version="0.2">
02   <device>
03     <match key="info.bus"
   string="usb">
04       <match key="usb.
   interface.class" int="0x06">
05         <match key="usb.
   interface.subclass"
   int="0x01">
06           <match key="usb.
   interface.protocol"
   int="0x01">
07             <merge key="info.
   category"
   type="string">camera</merge>
08             <append key="info.
   capabilities"
   type="strlist">camera</
   append>
09             <merge key="camera.
   access_method"
   type="string">ptp</merge>
10           </match>
11         </match>
12       </match>
13     </match>
14   </device>
15 </deviceinfo>
```

Figure 2: Working with HAL and D-Bus: the Gnome Volume Manager, which is user-configurable via gnome-volume-properties. (shown above)

other application. To reflect this, the second type represents a response to a call request. The third message type is used for error handling by the server application. The fourth message type provides signals transmitted by applications over the bus that do not require a response. Programmers can call D-Bus methods synchronously or asynchronously.

D-Bus uses a multi-level naming schema to identify message sources and targets. Each application contains one or more objects that can be addressed via paths made up of inverted domain names, with the object name itself appended, for example, */org/freedesktop/DBus*. Objects provide services that look similar, but are dot-separated: *org.freedesktop.DBus*. The interface groups the methods and signals for an object; again, a dotted notation is used in a similar approach to a Java interface.

Listing 2: cups.conf

```
01 <busconfig>
02   <!-- Only root can send this message -->
03   <policy user="root">
04     <allow send_interface="com.redhat.PrinterSpooler"/>
05   </policy>
06
07   <!-- Allow any connection to receive the message -->
08   <policy context="default">
09     <allow receive_interface="com.redhat.PrinterSpooler"/>
10   </policy>
11 </busconfig>
```

Security

It would not be a good idea to let non-privileged users access D-Bus. According to the developers, security has been a major concern right from the outset. In a simple case, the UID would be evaluated to control access. If the bus daemon and the clients belong to the same user, no restrictions are applied. D-Bus also implements security policies that define a user's privileges to allow more granular control (Listing 2).

D-Bus can also be used with SE Linux, probably because Red Hat is the main developer.

Practical Applications

Other applications besides Gnome have started to use D-Bus. A fairly up-to-date list is available at [5]. Although users are advised not to change working systems, you can use D-Bus to control the BMPx and Banshee audio players. A useful thing for network-aware programs is the current version of the Avahi Zero-Conf package which supports D-Bus. This means that applications can be notified when servers appear on the network.

Listing 3: hal.py

```
01 import dbus
02
03 bus = dbus.SystemBus()
04 proxy_obj = bus.get_object('org.freedesktop.Hal',
05                           '/org/freedesktop/Hal/Manager')
06 hal_manager = dbus.Interface(proxy_obj, 'org.freedesktop.Hal.Manager')
07
08 dev_list = hal_manager.GetAllDevices()
09
10 for dev in dev_list:
11     print dev,"\n"
```

D-Bus Programming

There is good news and bad news for D-Bus programmers. The good news is that bindings are available for a variety of programming languages [6], from the Glib-C API, through Python, to Ruby, C#, and Java. The bad news is that the API has changed so frequently in the past that many sample programs on the Internet will not run on the current D-Bus versions. There is a general lack of documentation on D-Bus interaction. Your best bet is to investigate the source code of working program, such as the Gnome

Listing 4: server.py

```
01 import gobject
02 import dbus
03 import dbus.glib
04 import dbus.service
05
06 class HelloWorldObject(dbus.service.Object):
07     def __init__(self, bus_name, object_path):
08         dbus.service.Object.__init__(self, bus_name, object_path)
09     @dbus.service.method('org.firstfloor.HelloWorldIFace')
10     def hello(self):
11         return "blabla"
12
13 session_bus = dbus.SessionBus()
14 bus_name = dbus.service.BusName('org.firstfloor.HelloWorld', bus=session_bus)
15 object = HelloWorldObject(bus_name, '/org/firstfloor/HelloWorldObject')
16
17 mainloop = gobject.MainLoop()
18 mainloop.run()
```

Network Manager, which was written in Python.

The basic approach is the same for all supported programming languages: connect to the bus, pick up a reference for the remote object and the interface, and issue requests, or register signal handlers. When a program registers a signal handler, it requires a main loop that regularly checks for incoming signals. It is advisable to use the Glib mainloop object for D-Bus programs, no matter whether you use C or a scripting language.

The following short sample code listing in Python shows you how to use D-Bus in your own programs. Using object-oriented constructs such as objects and interfaces is far easier in Python than in Glib-C programs. Just look how easy it is to import the D-Bus module in Line 1 of Listing 3. Some changes have occurred in D-Bus version 0.41 or newer: if you want to use the Glib mainloop, note that the Glib objects and methods now reside in the *dbus.glib* module.

Interface-based access is another feature that has developed over time. Many sample programs on the Internet demonstrate the use of the deprecated methods, *get_service*. Today, you need a proxy object that uses an interface to place a wrapper round the access. The *bus* object's *get_object* method gives you this ability, providing the arguments *org.freedesktop.Hal* and */org/freedesktop/Hal/Manager* (Listing 3, Line 4). The proxy object's static *dbus.Interface* method generates an interface, which the programmer can then call.

Some D-Bus API features require special expressions in the various programming languages. For example, the Python API uses Python 2.4's new decorators to identify signals and service methods (see Listing 4).

Just like in the client example, the sample program starts by connecting to the D-Bus. The *HelloWorldObject* object's constructor is then called to run the method *__init__* . The *@dbus.service.method* decorator specifies the interface methods.

Listing 5: BMPx.service

```
01 [D-BUS Service]
02 Name=org.beepmediaplayer.bmp
03 Exec=/usr/libexec/
   beep-media-player-2-bin
```

Figure 3: The HAL Device Manager showing a tree-view of the hardware

Automatic Launch

Thus far, we have just assumed that an application connects to D-Bus of its own accord and then runs as a client. A program that provides D-Bus services either needs to be launched at boot time, or the D-Bus server has to launch it. To allow this to happen, the server needs to know the service name of the binary to execute; the server parses a *.service* configuration file to discover the name. Listing 5 gives an example of a configuration file for the BMPx audio player.

As you have seen from the examples, it is not really difficult to use D-Bus to let your own applications talk to the outside world, although finding the right function in the API jungle can be a pain.

D-Bus has spread like wildfire to many Linux distributions, despite the fact that it is still under development and the interfaces change from one release to the next. If you are interested in experimenting with D-Bus at this stage, be aware that major changes to the D-Bus API can occur without notice.

Back to the Future

In a step that adds a touch of irony to the D-Bus development story, the inventors of D-Bus now find it important to have a system that works throughout a network. In this light, it is probably just a question of time until the system starts to mutate into the kind of monster that CORBA has become. If this happens, let's hope a new group of developers at Novell or Red Hat steps into to start again from scratch.

As of this writing, Gnome draws most heavily on D-Bus, to manage hotplug devices such as cameras, hard disks, or scanners, for example. But the KDE developers are slowly migrating their applications to D-Bus, and even Qt now speaks D-Bus, so application developers should maybe take some time out to ride the D-Bus, too. ■

INFO

[1] D-Bus: *http://www.freedesktop.org/wiki/Software/dbus*

[2] HAL: *http://www.freedesktop.org/wiki/Software/hal*

[3] Gnome NetworkManager: *http://www.gnome.org/projects/NetworkManager*

[4] LUKS for HAL: *http://www.redhat.com/magazine/012oct05/features/hal*

[5] D-Bus software: *http://www.freedesktop.org/wiki/Software_2fDbusProjects*

[6] Language bindings for D-Bus: *http://www.freedesktop.org/wiki/Software_2fDBusBindings*