Reprinted from the

Proceedings of the Linux Symposium

Volume Two

July 21th–24th, 2004
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Jes Sorensen, *Wild Open Source, Inc.*
Matt Domsch, *Dell*
Gerrit Huizenga, *IBM*
Matthew Wilcox, *Hewlett-Packard*
Dirk Hohndel, *Intel*
Val Henson, *Sun Microsystems*
Jamal Hadi Salimi, *Znyx*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*

On a Kernel Events Layer and User-space Message Bus System

Robert Love
Novell
rml@ximian.com

Abstract

Various Linux usage scenarios, particularly the widely accepted server and the rapidly growing desktop, require a lightweight, simple, asynchronous mechanism for kernel to user-space communication. Such a mechanism is crucial for the transmissions of events to user-space in a type-safe and clean manner. Further, a system-level messaging bus, which can deliver messages up the system stack on both a system-wide and per-user level, is required to further the integration of the Linux system.

This talk will discuss the design and implementation for two specific solutions, the Kernel Events Layer and D-BUS, to these two problems. Finally, useful solutions built on the sum of these technologies will be discussed—such as a fully integrated Linux desktop, from the kernel up through the GNOME desktop.

1 Introduction

Usually considered a plus of open source development, the Linux system is developed piece-meal, resulting in cleanly separated layers and properly defined interfaces. This separation, however, also results in a lack of integration among the various components comprising the system stack. In particular, the lack of integration is readily manifest between the lower levels of the stack—kernel and system-level components—and the upper levels of the system, such as the desktop environment on desktop machines.

A particularly important, but missing, component of the Linux system is an ubiquitous IPC mechanism and events system. Such a component would facilitate the dissemination of information up the system stack, better integrating the Linux system from the kernel up through the system layers, the desktop, and the end user applications and daemons. With well defined interfaces, such integration could occur while continuing the current separation and interoperability of Linux components.

What would such an IPC mechanism and event system allow? Quite a bit. Photo applications could start automatically in response to camera insertion. The volume of your music player could automatically lower in response to your phone ringing. System shutdown, reboot, and suspend messages could be trasmitted up the stack. HA applications could receive instant notifications from the kernel. No longer need components in the system live separate lives from the kernel, the layers below them, and themselves. Now, applications can communicate, listen, and evolve.

Such a system may be broken into three requirements:

- Kernel support implementing a kernel-to-user event mechanism

- A user-space message transport and IPC mechanism
- Applications sending and receiving such messages

This paper will discuss two specific implementations of these requirements:

- The Kernel Events Layer
- D-BUS

2 The Kernel Events Layer

2.1 Goals and Design

Current user-space grokking of the kernel typically requires some combination of periodic polling, parsing of unformatted text files from `/proc`, and luck. The Linux kernel currently lacks a mechanism for kernel to user-space communication.

The requirements for such a system include:

- simple and clean
- low overhead and scalable
- asynchronous transport accessible without polling
- type-safe
- generic enough for use in multiple usage scenarios
- support for formalized sender interfaces, allowing standardized messaging

Event systems have been proposed and even implemented, but they generally receive minimal community buyin, presumably due to a lack of one or moe of these requirements (more than likely, the “simple” bit).

2.2 Implementation

The Kernel Events Layer implements an event system satisfying these requirements.

Usage is simple:

```
send_event(int type, char
*interface, char *fmt, ...)
```

The `type` parameter specifies a constant value representing the type of message being sent. The `interface` value specifies the originator of the message. It is used to provide an interface object for object-based component and IPC systems such as CORBA and D-BUS. Finally, `fmt` and any following arguments provide the usual `va_list` of format and arguments.

Example:

```
send_event (DBUS_NORMAL,
"org.kernel.arch.cpu",
"overheating")
```

This specifies a message from the `org.kernel.arch.cpu` interface with a value of `overheating`.

The actual implementation of the Kernel Events Layer uses netlink. In fact, the Kernel Event Layer is simply specific netlink socket into user-space in which the event is formated and then reconstructed by user-space. Netlink is fast, simple, and already in the kernel. Thus it was a natural choice.

The Kernel Events Layer code uses `netlink_broadcast()` internally.

2.3 Real World Usage

The Kernel Events Layer is independent of any specific user-space transport mecha-

nism. The assumed use case is to create a new daemon (or modify an existing daemon, like the D-BUS system message bus, `dbus-system-1`). This daemon listens on the netlink socket, reading each event as it occurs. The events are parsed and reconstructed into the format native to the user-space transport mechanism.

In the case of D-BUS, the `dbus-system-1` daemon sends the kernel events out the system message bus. Components up the system stack may then receive the kernel events right off the D-BUS system bus, along with other system-wide messages.

3 D-BUS

D-BUS is a user-space IPC system.

D-BUS varies from other IPC mechanisms in that it provides a bus system (as opposed to point-to-point) over which messages (as opposed to byte streams) are transported. Messages include a header containing metadata about the message itself and a body containing the data. The bus system is created by forming a point-to-point connection between the D-BUS daemon and each listener. The daemon acts as the hub and the listeners as the spokes of a wheel.

D-BUS provides both a system-wide and a per-user session bus. The system-wide bus is used to dissemenate information on a machine-global scale. A single system daemon provides this service, allowing applications up the stack to receive messages from components down the stack. A security system implements access control.

The per-user session bus exists on a per-user basis, with one daemon created for each user session. The per-user daemon is used for general application IPC and is physically separate from the system-wide bus. The per-user daemon is generally used for traditional point-to-point IPC.

D-BUS is the name given to this system. It is composed of several architectural layers:

- The message bus daemon
- The D-BUS library, `libdbus`, which connects to applications together
- Wrapper libraries and bindings that wrap `libdbus` for direct use on various application frameworks, such as Glib or QT, and various languages, such as C# and Python. The wrapper libraries and bindings provide the API that most programmers should use as they both simplify the rather low-level `libdbus` API and provide an API more familiar and fit for that particular environment.

3.1 D-BUS Concepts

D-BUS introduces various concepts that comprise the IPC system.

- The **bus** is either the system-wide global bus or the per-user session bus.
- **Objects** represent an instance of a specific listener of a D-BUS message. Objects are contained within the applications that use D-BUS, and generally map to objects in object-oriented languages. Because D-BUS would not find using a pointer or reference to identify an object very friendly, it introduces a name for each object. The name resembles a UNIX filesystem path, such as /org/kernel/fs/filesystem.
- **Interfaces** represent methods or signals implemented on an object. Each object supports at least one interface.

- **Messages** are sent to and from a defined method or signal. D-BUS supports multiple message types: method invocation, method return, error message, and signal.

3.2 Use of D-BUS

D-BUS's simplicity, performance, and use of the message and bus paradigm set it up for use across the entire Linux system and make it a perfect replacement for CORBA, DCOP, and other IPC mechanisms.

Multiple projects are taking advantage of D-BUS. They include:

- Project Utopia uses D-BUS as the IPC mechanism to link the kernel, udev, HAL, and the GNOME desktop.
- A CUPS patch uses D-BUS to transmit information about the printer spool.
- Jamboree uses D-BUS to automatically mute the volume.
- A Gconf patch uses D-BUS as the Gconf transport mechanism.

4 The Kernel Events Layer, D-BUS, and Project Utopia

D-BUS is used as the backbone of Project Utopia, an umbrella project aiming to bring improved hardware management and system integration to the Linux system and GNOME desktop. Project Utopia uses D-BUS to link the kernel, up through hotplug, udev and HAL to the rest of the system. Libraries utilizing D-BUS and built on top of HAL provide enhanced hardware support. Applications at the desktop level can then reap the benefits.

4.1 Example: libinput

`libinput` is a simple library for managing input devices that sits on top of HAL and communicates to HAL beneath it and the applications above it via D-BUS. `libinput` is used to enumerate all input devices on the system. `libinput` also provides an interface for applications to register callbacks, and integrate these callbacks into its mainloop. The callbacks are invoked when input devices are added to or removed from the system.

Sample usage of enumerating all input devices on the system:

```
struct input *devices;

if (input_init ())
        /* error ... */

devices = input_devices_get ();
while (devices) {
        /* ... */
        devices = devices->next;
}
input_devices_put (devices);
```

Given a specific `struct input`, the library provides wrappers for opening and closing the device via `open`(2) and `close`(2). This is not strictly required, but furthers the abstracting of device nodes not only from the user but even from the application.

Example:

```
fd = input_device_open (device, 0);

/* ... */

input_device_close (device);
```

Registering of the callbacks is also easy:

```
void my_mainloop
```

```
(DBusConnection *dbus_connection)
{
  dbus_connection_setup_with_g_main
  (dbus_connection, NULL);
}

void my_added
(struct input *device)
{
  printf
  ("%s was just "
          "hotplugged!\n",
          device->product);
}

void my_removed
(struct input *device)
{
  printf
  ("%s was just "
         "hotunplugged!\n",
         device->product);
}

  /* ... */
  input_init_with_callbacks
        (&my_mainloop,
         &my_added,
         &my_removed);

  gtk_main ();
```

When an input device is added or removed from the system, `my_added` and `my_removed` are invoked as appropriate.

The goals behind such a library are twofold:

- Abstract away concepts of device nodes and low-level system-specific behavior and allow application developers to search for enumerate the devices on a system through simple interfaces.

- Allow asynchronous poll-free hack-free callbacks into the application to notify the program of changes in events, such as a new joystick on the system.

5 Conclusion

The Kernel Events Layer and D-BUS are two crucial components in better unifying and integrating the Linux system. They provide the infrastructure required for a future rich with information exchange. Where all levels of the desktop can communicate—talking, listening, evolving.