

Comunicación entre procesos con D-Bus y HAL

A TODA PASTILLA

¡Es el final de CORBA! Actualmente Gnome se basa en el sistema de mensajes D-Bus y KDE está en proceso de migración. **POR OLIVER FROMMEL**

A nadie le gusta que las aplicaciones se pasen todo el día en una esquina del escritorio sin intención de comunicarse entre ellas. Lo menos que se puede esperar es que intercambien datos con otras aplicaciones del escritorio utilizando simplemente la técnica de arrastrar y soltar. Pero muchos usuarios esperan que sus programas demuestren habilidades de comunicación incluso más avanzadas a cualquier nivel. Desde luego, éstos quieren utilizar memorias USB sin importarles qué programa van a usar. Además, los teléfonos software de VOIP deberían hacerse amigos del nuevo hardware cuando se cambien los auriculares sin tener que reiniciar el sistema.

Para ello, un sistema Linux necesita un mecanismo de comunicación que permita a las aplicaciones del escritorio hablar entre ellas y con los sistemas

subyacentes, así como con el kernel y el hardware. Y si los desarrolladores de Freedesktop tienen algo que decir, D-Bus [1] (que se basa en la capa de abstracción hardware HAL [2]) será el sistema de comunicación para la próxima generación de Linux.

Háblame

D-Bus es un sistema de comunicación interproceso (IPC); es decir, proporciona la infraestructura que permite a las aplicaciones hablar las unas con las otras y con el sistema operativo. Aunque los mecanismos de IPC se presentaron en UNIX hace años, están restringidos a las señales, a los canales y a elementos similares.

Esto puede que le resulte familiar; después de todo, las soluciones competitivas han estado rondando desde el principio. Tan sólo hay que pensar en CORBA, en DCOM de Microsoft o en cientos de otros proyectos. Tanto KDE como Gnome han experimentado con sus propias implementaciones de CORBA. KDE presentó su propio sistema DCOP, y el legado de Gnome lo representa el sistema de componentes de Bonobo. A pesar de las opiniones personales sobre CORBA, la mayoría de los desarrolladores, que lo único

Oliver pasó varios años como sysop y programador en el Centro de Electrónica Ars en Linz/Austria.



Tras finalizar sus estudios en Filosofía, Lenguas y Ciencias de la Computación, llegó a ser editor de la Corporación Bavarian Broadcasting. Actualmente es el jefe de la Centro Editorial de Software y Programación en Linux New Media AG.

EL AUTOR

que desean es programar una aplicación de escritorio, están sobrecargados por el sistema. Y esto posiblemente explique por qué Bonobo lleva tanto tiempo vegetando detrás de Gnome.

D-Bus está diseñado como un sistema simple y ligero. La librería básica Libdbus simplemente proporciona las funciones que permiten a dos aplicaciones comunicarse. Los desarrolladores de aplicaciones no utilizan normalmente la librería, prefieren usar la API basada en Glib Libdbus-Glib, que proporciona una API de C orientada a objetos. Es en este nivel donde las capacidades de D-Bus se extienden para proporcionar un sistema de bus genuino haciendo referencia a su nombre. El proceso servidor, *dbus-daemon*, se ejecuta en segundo plano y se queda a la escucha de peticiones de conexiones de aplicaciones que se registran para diferentes tipos de eventos, como la conexión y desconexión de hardware específico. Cuando se produce el evento, el servicio de D-Bus envía un mensaje por el bus y la aplicación responde de acuerdo al mensaje.

Sistema Global o Por Sesión

En los sistemas donde se utiliza D-Bus, cada proceso servidor implementa dos buses: el bus del sistema y el bus de la sesión. El bus del sistema se ejecuta cuando se arranca el equipo y se mantiene en ejecución incluso aunque no haya ningún usuario conectado. Cuando un usuario se

conecta y inicia una sesión del escritorio, se ejecuta un proceso servidor para la sesión. El binario *dbus-daemon* tiene parámetros de la línea de comandos para ambos modos: *--system* o *--session*. El paquete D-Bus incluye *dbus-launch* para ejecutar el servicio y establecer las variables de entorno requeridas. La mayoría de las distribuciones ejecutan el servicio D-Bus en modo sesión junto con la sesión X.

La Figura 1 muestra el papel que juegan los dos buses en la comunicación entre los componentes del sistema operativo. El bus de sesión permite a las aplicaciones pertenecientes a la misma sesión del escritorio hablar entre ellas. Desde luego, estas aplicaciones pueden ser servicios proporcionados por el propio entorno del escritorio. En contraste con esto, el bus del sistema principalmente se asegura de que los programas del escritorio puedan hablar con las capas inferiores. Por ejemplo, una aplicación puede utilizar el bus del sistema para registrarse a una clase específica de hardware, como las cámaras digitales.

Gestión del Hardware con HAL

D-Bus no proporciona su propia gestión del hardware; por el contrario, se basa en la capa de abstracción del hardware, HAL. Aunque HAL es independiente de D-Bus, los dos componentes funcionan mano a mano: HAL utiliza a D-Bus para pro-

porcionar servicios y D-Bus fue principalmente programado para HAL.

Además del kernel, las distribuciones modernas usan el subsistema Udev para la gestión del hardware. Desde la versión 0.59, Udev reemplaza al sistema hotplug, que simplemente se había establecido así mismo como un mecanismo para soportar el hardware que se conecta en caliente ejecutando */sbin/hotplug*. Además de la información del kernel y Udev, HAL posee ahora detalles adicionales sobre dispositivos almacenados como ficheros FDI (Device Information Files) en formato XML. El Listado 1 muestra una sección de un fichero FDI para una cámara digital.

Gnome NetworkManager [3] es un buen ejemplo de cómo cooperan los componentes. Utiliza el servicio de HAL para monitorizar el subsistema de red. Cuando se produce un cambio, como cuando un usuario conecta o desconecta un dispositivo de red inalámbrico USB, el servicio utiliza D-Bus para notificar a NetworkManager. Además de los dispositivos genuinos,

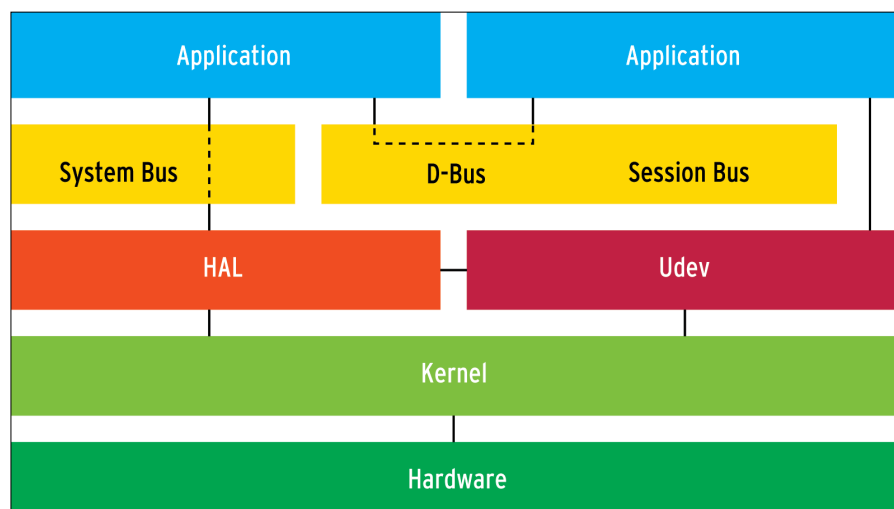


Figura 1: D-Bus y HAL en el contexto total de componentes de un sistema Linux. Las aplicaciones utilizan D-Bus para preguntarle a HAL sobre el hardware subyacente.

Listado 1: 10-camera-ptp.fdi

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

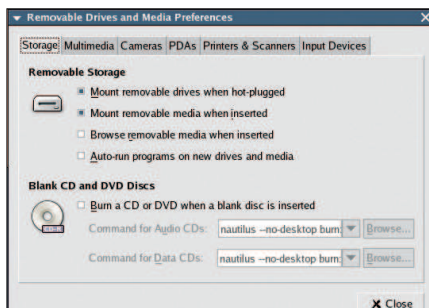


Figura 2: Trabajando con HAL y D-Bus: El Gnome Volume Manager, que es configurable por el usuario mediante `gnome-volume-properties`. (ver más abajo).

HAL también puede manejar el sistema de ficheros; tiene la habilidad de identificar tipos de sistemas de ficheros, incluyendo las particiones LUKS-encrypted [4]. En Gnome, HAL maneja ahora la gestión del hardware, particularmente los dispositivos de conexión en caliente. El proceso `gnome-volume-manager`, que se ejecuta en segundo plano, hace que esto funcione. Para configurar el proceso, los usuarios de Gnome ejecutan la interfaz `gnome-volume-properties` (Figura 2).

También hay otra interfaz para HAL que proporciona a los usuarios una vista en árbol de los dispositivos conectados (Figura 3). Los usuarios de Fedora encontrarán el `hal-device-manager` escondido en el paquete `hal-gnome`.

Listado 2: cups.conf

```
01 <busconfig>
02 <!-- Solo root puede enviar
este mensaje -->
03 <policy user="root">
04 <allow
send_interface="com.redhat.PrinterSpooler"/>
05 </policy>
06
07 <!-- Permitir que cualquier
conexión reciba el mensaje -->
08 <policy context="default">
09 <allow
receive_interface="com.redhat.PrinterSpooler"/>
10 </policy>
11 </busconfig>
for dev in dev_list: print
dev,"\\n"
```

Utilizando D-Bus

La especificación del protocolo D-Bus está disponible en [1]. El protocolo define cuatro tipos de mensajes que los usuarios pueden enviar por el bus. Por ejemplo, una aplicación puede llamar a los métodos proporcionados por otra aplicación. Para ello, el segundo tipo representa una respuesta a la petición de llamada. El tercero se utiliza para el manejo de los errores por la aplicación servidora, mientras que el cuarto tipo de mensaje proporciona las señales transmitidas por las aplicaciones por el bus que no requieren respuesta. Los programadores pueden llamar a los métodos de D-Bus tanto de forma síncrona como asíncrona.

D-Bus utiliza un esquema de nombres multinivel para identificar las fuentes y los destinos de los mensajes. Cada aplicación contiene uno o más objetos que pueden ser accesibles por las rutas formadas por los nombres de dominio invertidos, con el nombre del objeto añadido, por ejemplo, `/org/freedesktop/DBus`. Los objetos proporcionan servicios con aspecto similar, pero están separados por puntos: `org.freedesktop.DBus`. La interfaz agrupa los métodos y las señales de un objeto; de nuevo, una notación con puntos se utiliza de forma similar a como lo hacen los interfaces Java.

Seguridad

No hubiese sido una buena idea permitirles a los usuarios sin privilegios

Listado 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()
10 for dev in dev_list:
11     print dev,"\\n"
```

acceder a D-Bus. De acuerdo con los desarrolladores, la seguridad ha sido uno de los puntos clave. En un caso simple, la UID sería evaluada para el control de acceso. Si el servidor del bus y el cliente pertenecen al mismo usuario, no se aplica ninguna restricción. D-Bus también implementa políticas de seguridad que definen los privilegios de los usuarios para permitirles un control más preciso (Listado 2).

D-Bus puede también usarse con SE Linux, probablemente porque Red Hat es el principal desarrollador.

Aplicaciones Prácticas

Otras aplicaciones, además de Gnome, han empezado a utilizar D-Bus. Una lista bastante actualizada se encuentra disponible en [5]. Aunque los usuarios están avisados para que no cambien los sistemas en funcionamiento, se puede utilizar D-Bus para controlar los reproductores de audio BMPx y Banshee. Algo útil para los programas de red es la versión actual

Listado 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()
```

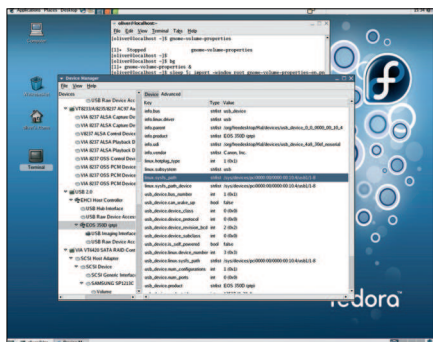


Figura 3: El Gestor de Dispositivos HAL muestra una vista en árbol del hardware.

del paquete Avahi Zero-Conf que soporta D-Bus. Esto significa que una aplicación puede ser notificada cuando aparezcan los servidores en la red.

Programación D-Bus

Hay buenas y malas noticias para los programadores de D-Bus. Las buenas noticias son que está disponible para diversos lenguajes de programación [6], desde la API Glib-C, hasta Python, Ruby, C# y Java. Las malas noticias son que la API ha cambiado con tanta frecuencia en el pasado, que muchos de los programas de ejemplo que hay disponibles en Internet no se ejecutarán en las versiones actuales de D-Bus. Hay una carencia general de documentación para interactuar con D-Bus. La mejor apuesta es investigar el código fuente de los programas que funcionan, como el NetworkManager de Gnome, que está escrito en Python.

La solución básica es la misma para todos los lenguajes de programación soportados: conectarse al bus, tomar una referencia del objeto remoto y la interfaz y realizar la petición o registrar el manejador de señal. Cuando un programa registra un manejador de señal, necesita que un bucle principal compruebe regularmente las señales entrantes. Es aconsejable utilizar el objeto bucle principal de Glib en los programas D-Bus, ya se esté programando en C o con un lenguaje de script.

Listado 5: BMPx.service

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

El siguiente código de ejemplo listado en Python muestra cómo utilizar D-Bus en un programa. Utilizando construcciones orientadas a objetos como objetos e interfaces es mucho más fácil en Python que en los programas Glib-C. Obsérvese lo sencillo que es importar el módulo D-Bus en la Línea 1 del Listado 3. Se han hecho algunos cambios a partir de la versión 0.41: si se quiere utilizar el bucle principal de Glib, téngase en cuenta que los objetos y métodos Glib residen ahora en el módulo *dbus.glib*.

El acceso basado en interfaces es otra característica que se ha desarrollado con el tiempo. Muchos programas de ejemplo de Internet muestran el uso de los métodos anticuados, *get_service*. Actualmente, se necesita un objeto proxy que utilice una interfaz para encapsular el acceso. El método *get_object* del objeto *bus* proporciona esta funcionalidad, suministrando los argumentos *org.freedesktop.Hal* y */org/freedesktop/Hal/Manager* (Listado 3, Línea 4). El objeto estático proxy *dbus.Interface* genera una interfaz a la que el programador puede llamar.

Algunas de las características del API D-Bus requieren expresiones especiales en los diversos lenguajes de programación. Por ejemplo, la API de Python utiliza los nuevos decoradores 2.4 para identificar las señales y los métodos de los servicios (véase 4).

Como en el cliente del ejemplo, el programa comienza conectándose a D-Bus. El constructor del objeto *HelloWorldObject* es entonces llamado para ejecutar el método *_init*. El decorador *@dbus.service.method* especifica los métodos de la interfaz.

Ejecución Automática

Hemos supuesto que una aplicación se conecta por sí misma a D-Bus y luego se ejecuta como un cliente. Un programa que proporcione servicios D-Bus tiene que ser ejecutado al arrancar el sistema o bien tendrá que ejecutarlo el servidor D-Bus. Para ello, el servidor tiene que conocer el nombre del servicio del ejecutable binario: el servidor procesa el fichero de configuración *.service* para encontrar el nombre. El Listado 5 muestra un ejemplo de fichero de configuración para el reproductor de audio BMPx.

Como se puede ver en los ejemplos, no es muy difícil utilizar D-Bus para permitirle a las aplicaciones comunicarse con el resto, aunque puede ser una tarea complicada encontrar la función adecuada en la jungla de la API.

D-Bus se ha extendido, como un fuego por el bosque, entre las distribuciones Linux; a pesar de que aún está en desarrollo y los interfaces cambian de una versión a otra. Si se está interesado en experimentar con D-Bus en esta fase, hay que tener en cuenta que la API de D-Bus puede cambiar sin aviso previo.

Regreso al Futuro

Por darle un poco de ironía a la historia del desarrollo de D-Bus, hay que decir que sus inventores consideran ahora importante tener un sistema que funcione en red. Ante esto, es probable que en cuestión de tiempo comience a mutar a alguna clase de monstruo al estilo CORBA. Si ocurriese, esperemos que algún grupo nuevo de desarrolladores de Novell o Red Hat empiecen de nuevo a partir de cero. En el momento de escribir este artículo, Gnome se basa fuertemente en D-Bus para gestionar los dispositivos de conexión en caliente como las cámaras, los discos duros o escáneres, por poner un ejemplo. Pero los desarrolladores de KDE están migrando muy lentamente sus aplicaciones a D-Bus e incluso Qt habla actualmente de D-Bus, así que los desarrolladores de aplicaciones deberían quizás tomarse algún tiempo antes de pasarse, también, a D-Bus. ■

Recursos

- [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 para HAL: <http://www.redhat.com/magazine/012oct05/features/hal>
- [5] Software D-Bus: http://www.freedesktop.org/wiki/Software_2fDBusProjects
- [6] Language bindings para D-Bus: http://www.freedesktop.org/wiki/Software_2fDBusBindings