Hop on the D-Bus

Michaël Van Canneyt

August 31, 2010

Abstract

D-BUS is an interprocess communication protocol that has been around for some time, and has gradually replaced the existing IPC mechanisms used by the major packages on the Unix desktop. An introduction.

1 Introduction

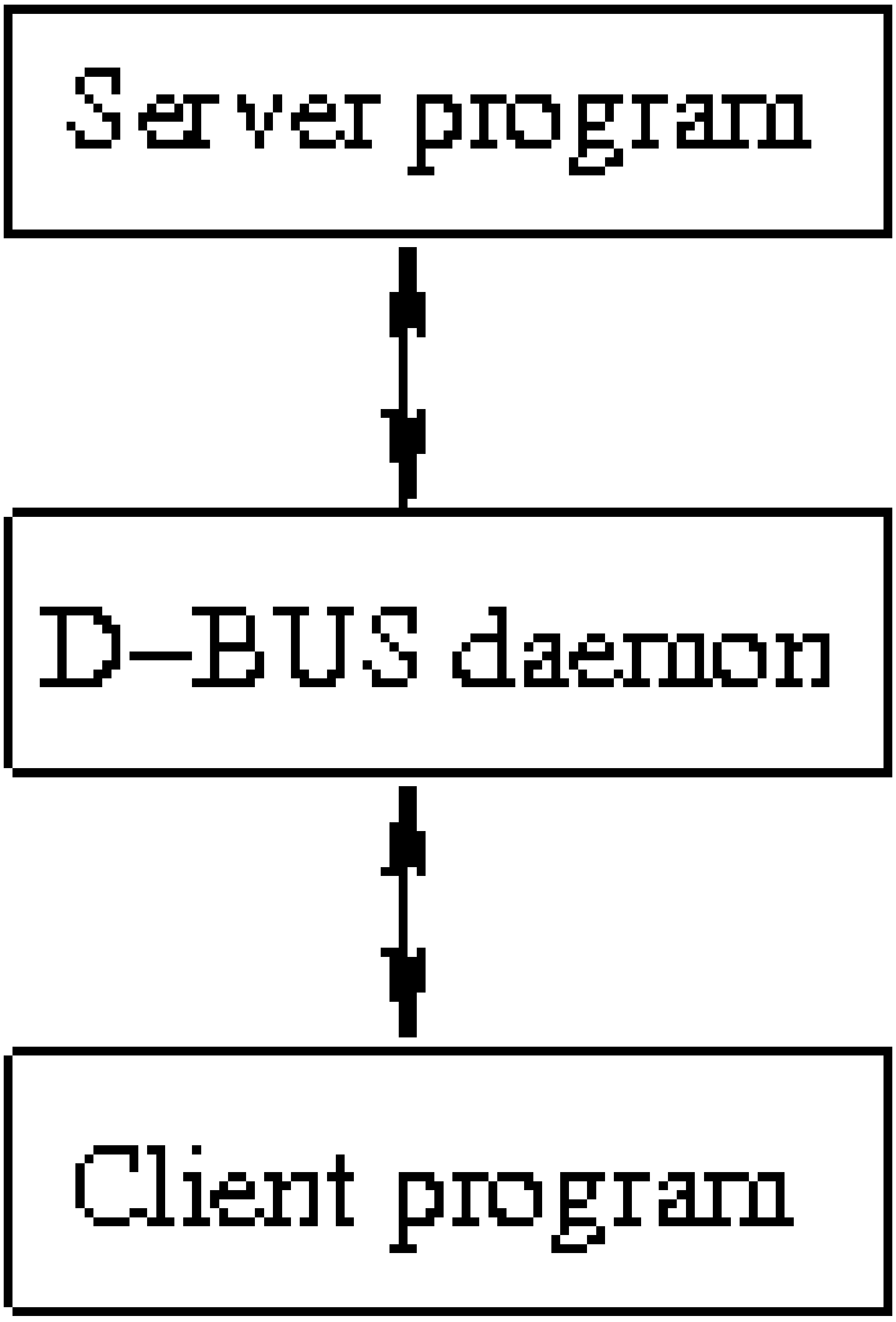
D-BUS implements a protocol which can be used by processes to communicate with one another. Communication can go from sending simple messages to letting the communication partner execute complex commands (remote method calling). Since it's initial development in 2002, it has replaced the mechanisms used by Gnome and KDE to achieve the same purpose. Many low-level unix programs - services which sit in the background - now also use this mechanism to communicate with all the other processes on the unix desktop: hal and udevd to name but a few.

D-Bus is implemented as a set of libraries and command-line tools. The library contains the actual implementation of the D-Bus protocol. The command-line tools contain the D-Bus daemon, and some tools to communicate through the D-Bus. In addition to this, some D-bus binding libraries exist: these are layers on top of the D-Bus protocol which make it easier to program the D-Bus. Bindings exist for glib, python, Qt, Ruby and Mono. Free Pascal also has low-level access to the D-Bus. The Qt interface is interesting to install, since it comes with an easy-to-use utility qdbus.

The D-BUS protocol is quite straightforward, and requires the presence of a daemon that runs in the background: dbus-daemon. Programs do not communicate directly with each other, they all communicate through the D-BUS daemon, which dispatches the messages to one or more recipients, as depicted in figure 1 on page 2. Usually, there are more than one dbus-daemon instances running on a machine: one system instance (the system bus), and one instance per logged-in user: the session bus. The system bus is meant to communicate with system services (daemons), whereas the session instance is meant for interaction between various programs that make up the desktop: the various programs of the KDE and GNOME desktop systems typically interact through the session desktop.

Whenever a D-BUS connection is made, the program must specify with which bus it wishes to connect: the system bus or the session bus. It is also possible to start a new bus: the system and session bus just create a bus with a well-known location: This is a unix socket, through which all communication is routed. The system bus, for instance, listens on a socket /var/run/dbus/system_bus_socket In fact, this is a configurable location: it is configured through the D-Bus configuration file, typically /etc/dbus-1/system.conf. The configuration file also handles various security settings.

Figure 1: Communication between client and server over the D-BUS



2 Signals and method calls

The D-BUS sends messages over the bus. It distinguishes mainly between 2 kinds of messages: signals and method calls. A signal is usually broadcast by a daemon to notify any listeners that something has changed on the system. A method call message is usually sent by a client to a daemon: the daemon executes the method call, and sends a reply to the client.

Each program that listens on the D-Bus for messages to which it answers, is called a service: each service has a (normally) unique name. For instance the dbus daemon itself listens on the D-Bus and is called 'org.freedesktop.DBus' (the name is case sensitive)

A program can contain multiple 'objects' which listen and respond to method calls. These objects are identified by their so-called Object Path. The D-Bus daemon exposes only one object: the root object /. Each object can expose various groups of method calls and signals, called interfaces: each interface is simply a name for a collection of methods. When sending a message with a method call, it is necessary to specify the interface name as well as the method name. The interface name can be omitted if the method name is unique over all exposed interfaces.

The D-Bus daemon exposes 2 interfaces:

org.freedesktop.DBus.Introspectable this interface contains just 1 method 'Introspect'. It will return a XML document that describes all interfaces, methods and signals exposed by the service.

org.freedesktop.DBus is an interface that allows to inspect and control the D-Bus daemon instance. One important method is `ListNames`, which lists all processes that are connected to the D-Bus.

As can be seen, the names look like domain names. There is no obligation to do this, it is just an unwritten agreement between developers; it also helps to prevent name clashes.

The `org.freedesktop.DBus.Introspectable` interface is expected to be implemented by all objects of all server processes that listen on the D-Bus, although this is not enforced. By implementing it, each object can be examined, and it's methods can be called.

3 dbus-send

The command-line tool dbus-send can be used to send messages over the d-bus. The qdbus tool is a similar tool, with a slightly easier interface. Both tools accept at least the following argument: –system or –session: they use it to decide to which bus they connect: the system bus or the session bus.

Executing qdbus with no arguments will send a message to the org.freedesktop.DBus service, and executes the `ListNames` method from the org.freedesktop.DBus interface. It results in output like the following:

```
 :1.0
 com.ubuntu.Upstart
:1.1
 org.freedesktop.Avahi
:1.10
 org.freedesktop.ConsoleKit
:1.11
:1.14
```

```
:1.15
 org.freedesktop.Hal
:1.16
:1.17
:1.18
:1.2
 org.freedesktop.NetworkManager
 org.freedesktop.NetworkManagerSystemSettings
```

The equivalent command for dbus-send is

```
dbus-send --print-reply --system --dest=org.freedesktop.DBus / org.freedesktop.DB
```

As can be seen, dbus-send is a bit more verbose, but it is also more clear what is happening: it sends a message to the service `org.freedesktop.DBus`, asking to execute method `org.freedesktop.DBus.ListNames` on object '/'. the `-print-reply` argument instructs dbus-send to print the reply of the method call. Without this argument, nothing would be printed. It also shows a glimpse of how D-Bus works internally: The result of a method call is just a new message sent back to the client executing the method.

4 The Introspectable interface

The `Introspect` method of the `org.freedesktop.DBus.Introspectable` interface is particularly interesting. Executing the following method

```
dbus-send --print-reply --system --dest=org.freedesktop.DBus / org.freedesktop.DB
```

results in a XML document being printed on the terminal:

```
    string "<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection
"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node>
  <interface name="org.freedesktop.DBus.Introspectable">
    <method name="Introspect">
      <arg name="data" direction="out" type="s"/>
    </method>
  </interface>
  <interface name="org.freedesktop.DBus">
    <method name="Hello">
      <arg direction="out" type="s"/>
    </method>
    <method name="RequestName">
      <arg direction="in" type="s"/>
      <arg direction="in" type="u"/>
      <arg direction="out" type="u"/>
    </method>
```

(The response has been shortened for display purposes). This XML document gives a lot of information about the D-Bus daemon in an easily understandable format. The same command can be executed for most objects of most services.

The `interface` tag describes an interface supported by an object. Each `method` tag describes a method, with `arg` elements to describe the various arguments to the call.

The `type` attribute of each argument describes the type of each argument. The various basic types are listed in the following table:

s	a string.
b	a boolean.
y	a byte.
i	a 32-bit signed integer.
u	a 32-bit unsigned integer.
n	a 16-bit signed integer.
q	a 16-bit unsigned integer.
x	a 64-bit signed integer.
t	a 64-bit unsigned integer.
d	a double-sized float.

In addition to the basic types, the D-BUS interface also supports arrays of types, they are designated by a combination: `aT`, where T is replaced by a basic type. Thus, `as` stands for an array of strings.

The XML document returned by the `Introspect` method starts with a `node` object: it stands for the object that is introspected. The `node` element can also contain a list of other `node` elements, which contain the names of the objects that are accessible below the current node. The following command

```
dbus-send --print-reply --system --dest=org.freedesktop.Hal /org/freedesktop/Hal
```

will result in 2 nodes:

```
!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"
"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">
<node>
  <interface name="org.freedesktop.DBus.Introspectable">
    <method name="Introspect">
      <arg name="data" direction="out" type="s"/>
    </method>
  </interface>
  <node name="Manager"/>
  <node name="devices"/>
</node>
```

From this we can deduce that the following objects exist inside the `org.freedesktop.Hal` service:

```
/org/freedesktop/Hal/Manager
/org/freedesktop/Hal/devices
```

The `org.freedesktop.DBus.Introspectable` interface together with the 'ListNames' method of the `org.freedesktop.DBus` interface of the DBus daemon makes it possible to inspect all objects exposed by the various services. In fact, the qdbusview program makes use of this. (this program is part of the qt4-dev-tools package on (K)Ubuntu). It

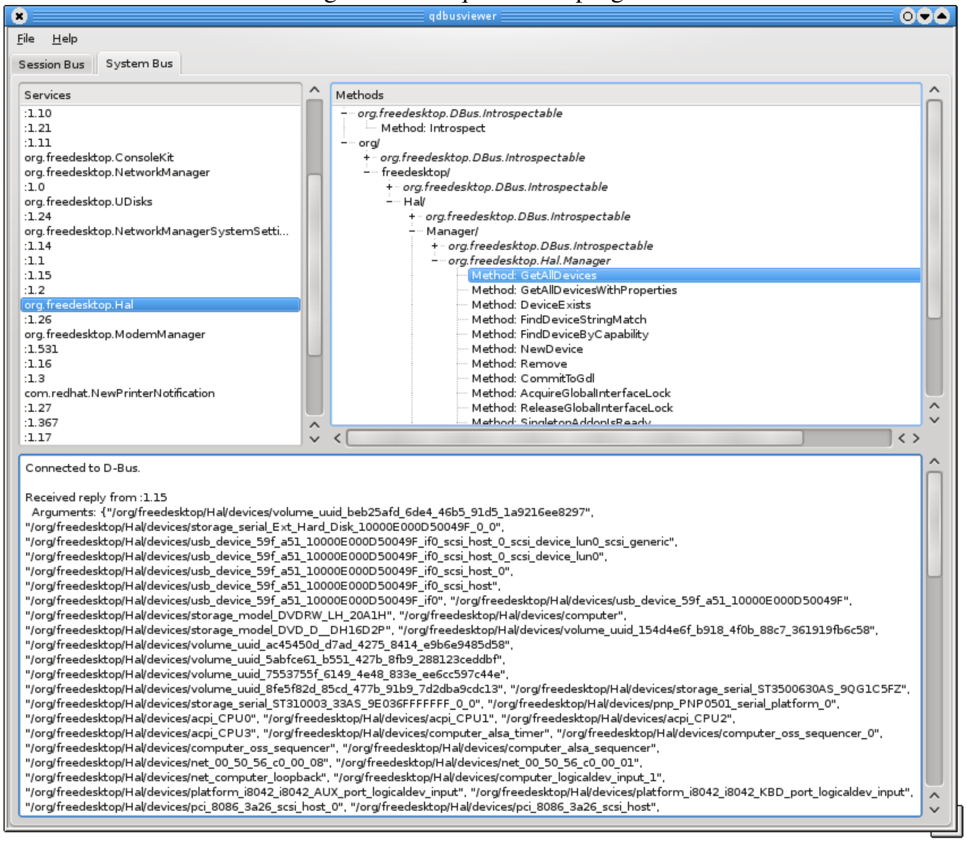


Figure 2: The qdbusview program

```
Services
:1.10
:1.21
:1.11
org.freedesktop.ConsoleKit
org.freedesktop.NetworkManager
:1.0
org.freedesktop.UDisks
:1.24
org.freedesktop.NetworkManagerSystemSetti...
:1.14
:1.1
:1.15
:1.2
org.freedesktop.Hal
:1.26
org.freedesktop.ModemManager
:1.531
:1.16
:1.3
com.redhat.NewPrinterNotification
:1.27
:1.367
:1.17
```

```
Methods
− org.freedesktop.DBus.Introspectable
      Method: Introspect
− org/
    + org.freedesktop.DBus.Introspectable
    − freedesktop/
        + org.freedesktop.DBus.Introspectable
        − Hal/
            + org.freedesktop.DBus.Introspectable
            − Manager/
                + org.freedesktop.DBus.Introspectable
                − org.freedesktop.Hal.Manager
                      Method: GetAllDevices
                      Method: GetAllDevicesWithProperties
                      Method: DeviceExists
                      Method: FindDeviceStringMatch
                      Method: FindDeviceByCapability
                      Method: NewDevice
                      Method: Remove
                      Method: CommitToGdl
                      Method: AcquireGlobalInterfaceLock
                      Method: ReleaseGlobalInterfaceLock
                      Method: SingletonAddonIsReady
```

```
Connected to D-Bus.

Received reply from :1.15
  Arguments: {"/org/freedesktop/Hal/devices/volume_uuid_beb25afd_6de4_46b5_91d5_1a9216ee8297",
"/org/freedesktop/Hal/devices/storage_serial_Ext_Hard_Disk_10000E000D50049F_0_0",
"/org/freedesktop/Hal/devices/usb_device_59f_a51_10000E000D50049F_if0_scsi_host_0_scsi_device_lun0_scsi_generic",
"/org/freedesktop/Hal/devices/usb_device_59f_a51_10000E000D50049F_if0_scsi_host_0_scsi_device_lun0",
"/org/freedesktop/Hal/devices/usb_device_59f_a51_10000E000D50049F_if0_scsi_host_0",
"/org/freedesktop/Hal/devices/usb_device_59f_a51_10000E000D50049F_if0_scsi_host",
"/org/freedesktop/Hal/devices/usb_device_59f_a51_10000E000D50049F_if0", "/org/freedesktop/Hal/devices/usb_device_59f_a51_10000E000D50049F",
"/org/freedesktop/Hal/devices/storage_model_DVDRW_LH_20A1H", "/org/freedesktop/Hal/devices/computer",
"/org/freedesktop/Hal/devices/storage_model_DVD_D__DH16D2P", "/org/freedesktop/Hal/devices/volume_uuid_154d4e6f_b918_4f0b_88c7_361919fb6c58",
"/org/freedesktop/Hal/devices/volume_uuid_ac45450d_d7ad_4275_8414_e9b6e9485d58",
"/org/freedesktop/Hal/devices/volume_uuid_5abfce61_b551_427b_8fb9_288123ceddbf",
"/org/freedesktop/Hal/devices/volume_uuid_7553755f_6149_4e48_833e_ee6cc597c44e",
"/org/freedesktop/Hal/devices/volume_uuid_8fe5f82d_85cd_477b_91b9_7d2dba9cdc13", "/org/freedesktop/Hal/devices/storage_serial_ST3500630AS_9QG1C5FZ",
"/org/freedesktop/Hal/devices/storage_serial_ST310003_33AS_9E036FFFFFFF_0_0", "/org/freedesktop/Hal/devices/pnp_PNP0501_serial_platform_0",
"/org/freedesktop/Hal/devices/acpi_CPU0", "/org/freedesktop/Hal/devices/acpi_CPU1", "/org/freedesktop/Hal/devices/acpi_CPU2",
"/org/freedesktop/Hal/devices/acpi_CPU3", "/org/freedesktop/Hal/devices/computer_alsa_timer", "/org/freedesktop/Hal/devices/computer_oss_sequencer_0",
"/org/freedesktop/Hal/devices/computer_oss_sequencer", "/org/freedesktop/Hal/devices/computer_alsa_sequencer",
"/org/freedesktop/Hal/devices/net_00_50_56_c0_00_08", "/org/freedesktop/Hal/devices/net_00_50_56_c0_00_01",
"/org/freedesktop/Hal/devices/net_computer_loopback", "/org/freedesktop/Hal/devices/computer_logicaldev_input_1",
"/org/freedesktop/Hal/devices/platform_i8042_i8042_AUX_port_logicaldev_input", "/org/freedesktop/Hal/devices/platform_i8042_i8042_KBD_port_logicaldev_input",
"/org/freedesktop/Hal/devices/pci_8086_3a26_scsi_host_0", "/org/freedesktop/Hal/devices/pci_8086_3a26_scsi_host",
```

offers a view of all services available on the system and session D-BUS in an file explorer-like interface, as shown in figure 2 on page 6. Since the number and type of arguments for a call are known from the Introspect call, it can also call a method and display the result - all this without any compiled-in knowledge of any of the service details. This tool is invaluable when developing applications that need to talk to services connected to the D-Bus.

5 Use in scripting

Armed with the knowledge presented here, how can the D-Bus be used in scripting ? The dbus-send command-tool (or qdbus) can obviously be used in scripting to send commands to services. One of these services is the KDE KLauncher service, which is listening on the session bus under the 'org.freedesktop.klauncher' service. The KLauncher API is documented on the KDE website:

`http://api.kde.org/4.x-api/kdelibs-apidocs/kinit/html/classKLauncher.html`

It can be called with dbus-send as follows:

```
dbus-send --type=method_call --dest=org.kde.klauncher \
 /KLauncher org.kde.KLauncher.start_service_by_desktop_name \
 string:"dolphin" array:string:"." array:string:'' \
 string:'' boolean:'false'
```

This will open the file manager 'dolphin' (first argument) on the 'Documents' folder (second argument). The meaning of other arguments are not very relevant for the working of the example, but they must be present.

The call returns (among other things) the process ID of the started instance of Dolphin. It can be caught for later use with some shell tricks:

```
PID=`dbus-send --type=method_call --print-reply\
 --dest=org.kde.klauncher \
 /KLauncher org.kde.KLauncher.start_service_by_desktop_name \
 string:"dolphin" array:string:"." array:string:'' \
 string:'' boolean:'false' \
| tail -1 | awk '{print $2}'`
```

After this command, the PID environment variable is filled with the process ID of the dolphin instance. It can be used for example to kill the process.

Another interesting service is the udisks-daemon: it can be used to get information about mounted disks. (an alternative is using the Hal service). The following can be used to list the disk devices:

```
dbus-send --print-reply --system \
  --dest=org.freedesktop.UDisks /org/freedesktop/UDisks \
  org.freedesktop.UDisks.EnumerateDevices
```

This will print a list of devices. Each device is of the form `/org/freedesktop/UDisks/devices/sdc1`.

Using the `Get` method of the `org.freedesktop.DBus.Properties` interface (this is a system interface to check properties of objects), it is possible to check whether the disk is mounted:

```
bus-send --print-reply --system \
  --dest=org.freedesktop.UDisks \
  /org/freedesktop/UDisks/devices/sdc1 \
  org.freedesktop.DBus.Properties.Get \
  string:org.freedesktop.UDisks.Device \
  string:DeviceIsMounted
```

This will print something like:

```
 variant       boolean true
```

meaning that `/dev/sdc1` is currently mounted.

Running this in a loop or a cron job, this can be used to check periodically if a removable disk is inserted, and if so, start e.g. a backup process.

6 monitoring signals

While it is easy to run a script periodically and check if a device is mounted, it is not very efficient. There is a better system. The HAL daemon monitors the system, and sends signals when new devices are attached or removed. When a new disk is attached to the system (such as a USB disk) it sends the signal 'DeviceAdded'. This signal can be listened for.

The dbus-monitor program (part of D-bus) can be used to monitor for signals. It takes a single argument, a filter rule for the signals it should watch out for. For the HAL daemon, the sender of the signal 'DeviceAdded' is 'org.freedesktop.Hal', and comes from the interface 'org.freedesktop.Hal.Manager'. All this is combined to a rule for dbus-monitor as follows:

```
dbus-monitor --system "type='signal',\
sender='org.freedesktop.Hal',\
interface='org.freedesktop.Hal.Manager',\
member='DeviceAdded'"
```

Running this command and adding a USB, will result in dbus-monitor printing several messages to the screen. Each message will contain the name of the device that corresponds to the disk: the disk will be known by several names: a raw device, a device with UUID (unique volume ID of each partition) and one with the label of the disk.

The following script will catch the output of dbus-monitor and will watch for messages that contain the uuid of the disk:

```
#!/bin/sh
#
# Monitoring only devices by UUID
MON=/org/freedesktop/Hal/devices/volume_uuid_
# Construct filter
FILTER="type='signal',\
sender='org.freedesktop.Hal',\
interface='org.freedesktop.Hal.Manager',\
member='DeviceAdded'"
# Read output of D-BUS in line:
dbus-monitor --system $FILTER | while read line
do
  # Extract device name:
  DEV=$(echo $line | grep $MON |\
        awk '{print $NF}' | sed s+$MON++)
  if [ ! -z "$DEV" ]; then
    # If it is a UUID device, print a message
    echo "Device $DEV appeared"
  fi
done
#
```

It will produce something like this:

```
Device "8fe5f82d_85cd_477b_91b9_7d2dba9cdc13" appeared
```

Instead of issuing an 'echo' command, a shell script can be started which mounts the device and for example starts a backup.

7 conclusion

The D-Bus is a powerful communication system. Various system daemons make good use of it, and a system administrator can also use it with some of the provided command-line tools and some scripts. But D-Bus offers also a convenient API to program it. A discussion of the API is left for a next contribution.