Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

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

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Playing BlueZ on the D-Bus

Marcel Holtmann
BlueZ Project
marcel@holtmann.org

Abstract

The integration of the Bluetooth technology into the Linux kernel and the major Linux distributions has progressed really fast over the last two years. The technology is present almost everywhere. All modern notebooks and mobile phones are shipped with built-in Bluetooth. The use of Bluetooth with a Linux based system is easy and in most cases it only needs an one-time setup, but all the tools are still command line based. In general this is not so bad, but for a greater success it is needed to seamlessly integrate the Bluetooth technology into the desktop. There have been approaches for the GNOME and KDE desktops. Both have been quite successful and made the use of Bluetooth easy. The problem however is that both implemented their own framework around the Bluetooth library and its daemons and there were no possibilities for programs from one system to talk to the other. With the final version of the D-Bus framework and its adaption into the Bluetooth subsystem of Linux, it will be simple to make all applications Bluetooth aware.

The idea is to establish one central Bluetooth daemon that takes care of all task that can't or shouldn't be handled inside the Linux kernel. These jobs include PIN code and link key management for the authentication and encryption, caching of device names and services and also central control of the Bluetooth hardware. All possible tasks and configuration options are accessed via the D-Bus interface. This will allow to abstract the internals of GNOME and KDE applications from any technical details of the Bluetooth specification. Even other application will get access to the Bluetooth technology without any hassle.

1 Introduction

The Bluetooth specification [1] defines a clear abstraction layer for accessing different Bluetooth hardware options. It is called the *Host Controller Interface* (HCI) and is the basis of all Bluetooth protocols stacks (see Figure 1).

This interface consists of commands and events that provide support for configuring the local device and creating connections to other Bluetooth devices. The commands are split into six different groups:

- Link Control Commands
- Link Policy Commands
- Host Controller and Baseband Commands
- Informational Parameters
- Status Parameters
- Testing Commands

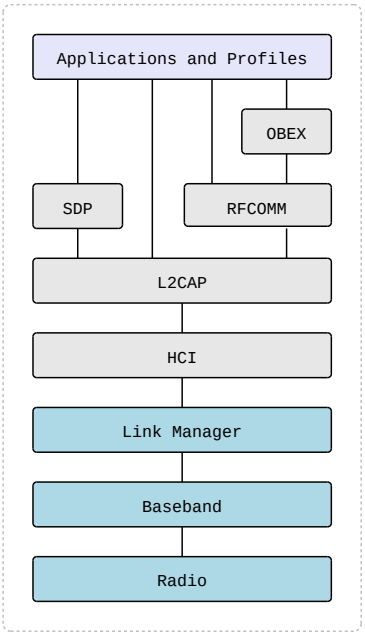

Figure 1: Simple Bluetooth stack

With the *Link Control Commands* it is possible to search for other Bluetooth devices in range and to establish connections to other devices. This group also includes commands to handle authentication and encryption. The *Link Policy Commands* are controlling the established connections between two or more Bluetooth devices. They also control the different power modes. All local settings of a Bluetooth device are modified with commands from the *Host Controller and Baseband Commands* group. This includes for example the friendly name and the class of device. For detailed information of the local device, the commands from the *Informational Paramters* group can be used. The *Status Parameters* group provides commands for detailed information from the remote device. This includes the link quality and the RSSI value. With the group *Testing Commands* the device provides commands for Bluetooth qualification testing. All commands are answered by an event that returns the requested value or information. Some events can also arrive at any time. For example to request a PIN code or to notify of a changed power state.

Every Bluetooth implementation must implement the *Host Controller Interface* and for Linux a specific set of commands has been integrated into the Linux kernel. Another set of commands are implemented through the Bluetooth library. And some of the commands are not implemented at all. This is because they are not needed or because they have been deprecated by the latest Bluetooth specification. The range of commands implemented in the kernel are mostly dealing with Bluetooth connection handling. The commands in the Bluetooth library are for configuration of the local device and handling of authentication and encryption.

While the *Host Controller Interface* is a clean hardware abstraction, it is not a clean or easy programming interface. The Bluetooth library provides an interface to HCI and an application programmer has to write a lot of code to get Bluetooth specific tasks done via HCI. To make it easy for application programmers and also end users, a task based interface to Bluetooth has been designed. The definition of this tasks has been done from an application perspective and they are exported through D-Bus via methods and signal.

2 D-Bus integration

The `hcid` daemon is the main daemon when running Bluetooth on Linux. It handles all device configuration and authentication tasks. All configuration is done via a simple configuration file and the PIN code is handled via PIN helper script. This means that every the configuration option needed to be changed, it was needed to edit the configuration file (`/etc/bluetooth/hcid.conf`) and to restart `hcid`. The configuration file still configures the basic and also default settings of `hcid`, but with the D-Bus

integration all other settings are configurable through the D-Bus API. The current API consists of three interfaces:

- org.bluez.Manager
- org.bluez.Adapter
- org.bluez.Security

The *Manager* interface provides basic methods for listing all attached adapter and getting the default adapter. In the D-Bus API terms an adapter is the local Bluetooth device. In most cases this might be an USB dongle or a PCMCIA card. The *Adapter* interface provides methods for configuration of the local device, searching for remote device and handling of remote devices. The *Security* interface provides methods to register passkey agents. These agents can provide fixed PIN codes, dialog boxes or wizards for specific remote devices. All Bluetooth applications using the D-Bus API don't have to worry about any Bluetooth specific details or details of the Linux specific implementation (see Figure 2).

Besides the provided methods, every interface contains also signals to broadcast changes or events from the HCI. This allows passive applications to get the information without actively interacting with any Bluetooth related task. An example for this would be an applet that changes its icon depending on if the local device is idle, connected or searching for other devices.

Every local device is identified by its path. For the first Bluetooth adapter, this would be `/org/bluez/hci0` and this path will be used for all methods of the *Adapter* interface. The best way to get this path is to call `DefaultAdapter()` from the *Manager* interface. This will always return the current default adapter or in error if no Bluetooth adapter is attached. With `ListAdapters()` it is possible to get a complete list of paths of the attached adapters.

If the path is known, it is possible to use the full *Adapter* interface to configure the local device or handle tasks like pairing or searching for other devices. An example task would be the configuration of the device name. With `GetName()` the current name can be retrieved and with `SetName()` it can be changed. Changing the name results in storing it on the filesystem and changing the name with an appropriate HCI command. If the local device already supports the Bluetooth Lisbon specification, then the *Extended Inquiry Response* will be also modified.

With the `DiscoverDevices()` method it is possible to start the search for other Bluetooth devices in range. This method call actually doesn't return any remote devices. It only starts the inquiry procedure of the Bluetooth chip and every found device is returned via the `RemoteDeviceFound` signal. This allows all applications to handle new devices even if the discovery procedure has been initiated by a different application.

3 Current status

The methods and signals for the D-Bus API for Bluetooth were chosen very carefully. The goal was to design it with current application needs in mind. It also aims to fulfill the needs of current established desktop frameworks like the *GNOME Bluetooth subsystem* and the *KDE Bluetooth framework*. So it covers the common tasks and on purpose not everything that might be possible. The API can be divided into the following sections:

- Local

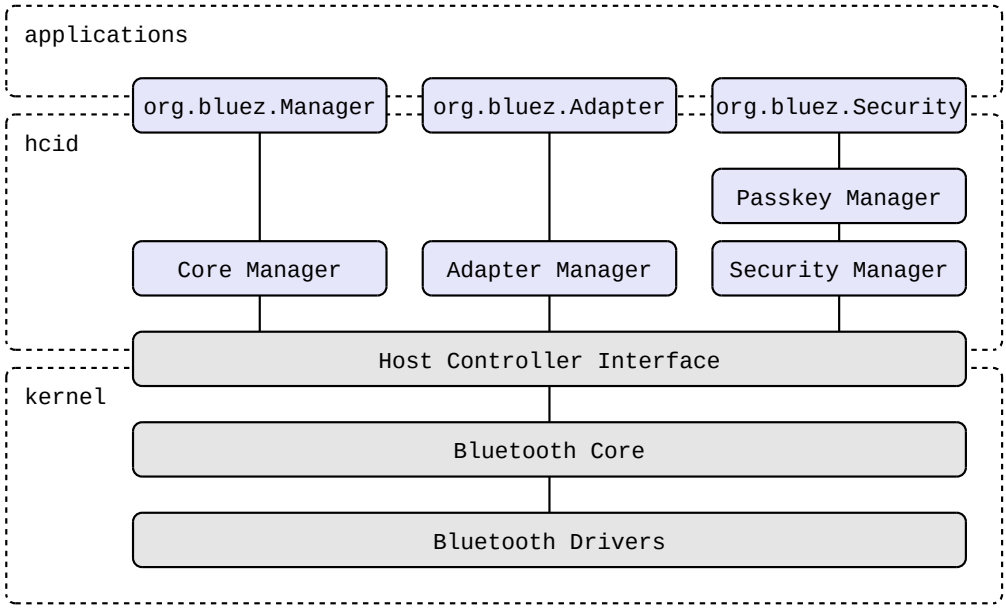

Figure 2: D-Bus API overview

  - version, revision, manufacturer
  - mode, name, class of device
- Remote
  - version, revision, manufacturer
  - name, class of device
  - aliases
  - device discovery
  - pairing, bondings
- Security
  - passkey agent

With these methods and signals all standard tasks are covered. The *Manager*, *Adapter* and *Security* interfaces are feature complete at the moment.

4 Example application

The big advantage of the D-Bus framework is that it has bindings for multiple programming languages. With the integration of D-Bus into the Bluetooth subsystem, the use of Bluetooth from various languages becomes reality. The Figure 3 shows an example of changing the name of the local device into *My Bluetooth dongle* using the Python programming language.

The example in Python is straight forward and simple. Using the D-Bus API within a C program is a little bit more complex, but it is still easier than using the native Bluetooth library API. Figure 4 shows an example on how to get the name of the local device.

5 Conclusion

The integration of a D-Bus API into the Bluetooth subsystem makes it easy for applications to access the Bluetooth technology. The current API is a big step into the right direction, but it is still limited. The Bluetooth technology is complex and Bluetooth services needs to be extended with an easy to use D-Bus API.

```
#!/usr/bin/python

import dbus

bus = dbus.SystemBus();

obj = bus.get_object('org.bluez',
            '/org/bluez')

manager = dbus.Interface(obj,
            'org.bluez.Manager')

obj = bus.get_object('org.bluez',
            manager.DefaultAdapter())

adapter = dbus.Interface(obj,
            'org.bluez.Adapter')

adapter.SetName('My Bluetooth dongle')
```

Figure 3: Example in Python

```
#include <stdio.h>
#include <stdlib.h>

#include <dbus/dbus.h>

int main(int argc, char **argv) {
 DBusConnection *conn;
 DBusMessage *msg, *reply;
 const char *name;

 conn = dbus_bus_get(DBUS_BUS_SYSTEM, NULL);
 msg  = dbus_message_new_method_call(
        "org.bluez",
        "/org/bluez/hci0",
        "org.bluez.Adapter", "GetName");

 reply =
   dbus_connection_send_with_reply_and_block(
        conn, msg, -1, NULL);

 dbus_message_get_args(reply, NULL,
        DBUS_TYPE_STRING, &name,
        DBUS_TYPE_INVALID);

 printf("%s\n", name);

 dbus_message_unref(msg);
 dbus_message_unref(reply);
 dbus_connection_close(conn);

 return 0;
}
```

Figure 4: Example in C

The next steps would be integration of D-Bus into the Bluetooth mouse and keyboard service. Another goal is the seamless integration into the Network Manager. This would allow to connect to Bluetooth access points like any other WiFi access point.

The current version of the D-Bus API for Bluetooth will be used in the next generation of the *Maemo platform* which is that basis for the *Nokia 770* Internet tablet.

References

[1] Special Interest Group Bluetooth: *Bluetooth Core Specification Version 2.0 + EDR*, November 2004.

[2] freedesktop.org: *D-BUS Specification Version 0.11*.