Modern IPC in Linux

A brief introduction to D-Bus

Author: Martin Sehnoutka
Operating System course, University of Patras

Contents

Abstract

Inter Process Communication (IPC) is an essential part of every operating system. Classic mechanisms like shared memory, unix-domain socket or pipes provide necessary IPC primitives, but their usage is complicated and way too low level for common use cases. D-Bus is a software bus abstraction, that is designed specifically for IPC in Linux desktop environment.

1 Theoretical background

In this section I briefly introduce theoretical background necessary for understanding D-Bus.

1.1 What is Linux?

Linux itself is just a kernel [1]. It is not a small project, in fact it is probably the biggest open-source project in the world, but to create an operating system (OS), kernel itself is not enough. This is where Linux distributions are introduced. A distribution is a set of software packages, that together create a functional operating system. There is not just one distribution, in fact there are hundreds of them, because every developer or system administrator has different needs from their OS. This is the reason for introducing distributions with short life time, but newest packages (Fedora, Ubuntu) or stable, tested, but a little bit outdated packages (Debian, Red Hat Enterprise Linux) or distributions with third-party closed source components or without them etc.

A D-Bus is a package as any other component, so it is not necessarily included in every distribution, but there is a high chance, that it will be included as a part of a desktop OS.

1.2 Classic IPC mechanisms

Traditional IPC mechanisms are shared memory or message passing (Figure 1), where unix-domain sockets are example of message passing mechanism. These techniques are very flexible and can be used in many different ways, so they are suitable for creating more high-level abstractions on top of them.

[1] Source: `https://www.kernel.org/category/about.html`

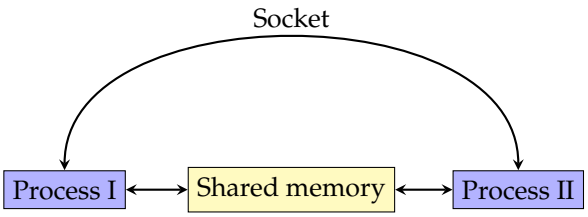

Figure 1: Classic IPC mechanisms

2 D-Bus introduction

In this chapter, D-Bus is introduced from ideas behind it up to various implementations.

2.1 Motivation for a new IPC mechanism

In order to build a modern desktop operating system, many processes need to run at the same time. For instance, dhcp client, graphical desktop environment, firewall or daemon responsible for printers. In fact hundreds of processes are running at the same time. It can be counted easily using the **ps** command, which displays all running processes and then count lines using **wc**:

```
~ $ # Count running processes
~ $ ps -A | wc -l
310
```

All of these processes need to communicate with each other, in order to share important information like user notifications, new hardware addition or change in network connection. There is much more possible applications for it, but they are less obvious, for those not interested in Linux desktop.

But D-Bus is not restricted to desktop environment. It can also be used for server side deployment and controlled over ssh.

2.2 D-Bus big picture

D-Bus, as the name suggests, is an implementation of a software bus abstraction (Figure 2). All processes, that are connected to a bus, can see each other and communicate over the bus. The bus can be used for Remote Procedure Call (RPC) and 1-to-many publish-subscribe mechanism [2].

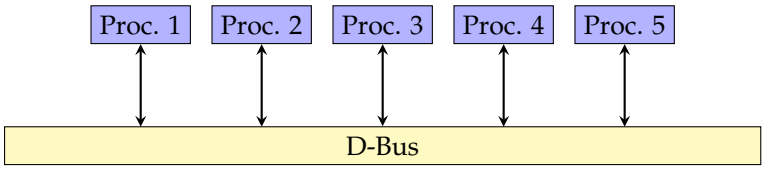

Figure 2: D-Bus idea

There can be multiple instances of the D-Bus running at the same time at one OS. One system-wide instance is always running, but user specific instances are created on demand, one for each login session. This comes from the obvious requirement not to mix different user sessions together. For example, one used do not want to be notified when a different one receives a new email.

2.3 D-Bus concepts

Terminology used in D-Bus might be little confusing at first sight. Basic building block is called a **service**, this is a running process, that is connected to a D-Bus instance and provide some **objects**. Each **service** can have multiple **objects** associated with it. For example login **service** exports one **object** for each running session.

An **object** provides one or more **interfaces**. Similarly to C# or Java, where a programmer can define multiple interfaces for different purposes and implement them all for one class. This **interface** consist of **methods**,

[2] Source: `https://www.freedesktop.org/wiki/Software/dbus/`

signals and **properties**. **Methods** can be used to call a remote procedure (RPC), **signals** are used to subscribe process for some notifications and **properties** define some state of a service.

Method arguments are typed using the D-Bus type system. It provides basic data types like integers or strings and basic collection types like lists and dictionaries.

Here is a list of concepts used in D-Bus [3]:

- **Service**(name in reverse domain notation)
- A program that offers some API over a bus
- e.g. *com.example.test*
  - **Object** (path notation)
  - Each service can provide more objects, that are in tree-like hierarchy
  - e.g. */com/example/test*
    * **Interface** - a collection of **methods**, **signals** and **properties**
    * Again in reverse domain notation
    * e.g. *com.example.test.calc*

And overview of interface members:

- Methods
  - RPC mechanism
  - Methods has a signature and a return value
  - e.g. ss for two strings or {is} for dictionary{int=>string}
- Signals
  - 1-to-many publish-subscribe mechanism
  - Client can subscribe for signals that are emitted by an object
- Properties
  - Like member variables in OOP
  - with set/get methods

3 Command line tools and code examples

There is not just one implementation of D-Bus specification. Different projects like GNOME, KDE or systemd has its own implementation and some programming languages like Python has their own native implementation. If you want to start exploring D-Bus, you can do so using command line tools, that are probably already included in your Linux desktop.

Of course there are also graphical tools like D-Feet, but I am not going to mention them, as their usage is similar to the command line tools.

3.1 Tools

busctl[4] is a tool, that comes with systemd init system. If you are using mainstream distributions like Fedora or Ubuntu, you are most likely using systemd as well. On the other hand, if you are using Gentoo or Alpine Linux, there is probably no need for me to explain this further.

The nice thing about **busctl** is its tab completion feature. You can use it to write keywords, names and even argument's types.

```
$ busctl # press tab twice
call     get-property  introspect  monitor       status
capture  help          list        set-property  tree
```

For example you can use it to list all services and their objects attached to the user session by using:

[3] Source: `https://pythonhosted.org/txdbus/dbus_overview.html`

[4] Upstream: `https://github.com/systemd/systemd/tree/master/src/busctl`

```
$ busctl --user tree
```

Or list methods and signals exported by an object. In this case the Notification object of running desktop environment:

```
$ # The output is cut off
$ busctl --user introspect org.freedesktop.Notifications /org/freedesktop/Notifications
NAME                                TYPE      SIGNATURE     RESULT/VALUE FLAGS
org.freedesktop.DBus.Introspectable interface -             -            -
.Introspect                         method    -             s            -
...
org.freedesktop.Notifications       interface -             -            -
.CloseNotification                  method    u             -            -
.GetCapabilities                    method    -             as           -
.GetServerInformation               method    -             ssss         -
.Notify                             method    susssasa{sv}i u            -
...
```

There is also a family of tools, that starts with "dbus-". These can be also used to control, monitor and debug services:

```
$ # This output is generated using tab completion of the Fish shell (alternative to bash)
$ dbus-monitor --session
dbus-binding-tool                          (C language GLib bindings generation utility.)
dbus-cleanup-sockets                          (Clean up leftover sockets in a directory)
dbus-daemon                                                        (Message bus daemon)
dbus-launch                              (Utility to start a message bus from a shell script)
dbus-monitor                                (Debug probe to print message bus messages)
dbus-run-session                              (Start a process as a new D-Bus session)
dbus-send                                            (Send a message to a message bus)
dbus-test-tool                                (D-Bus traffic generator and test tool)
dbus-update-activation-environment  (Update environment used for D-Bus session services)
dbus-uuidgen                                                  (Utility to generate UUIDs)
```

3.2 Programming interfaces

As mentioned before, there is multiple implementations of the D-Bus specification. Most of the implementations are in C, but binding exist for many other languages [5].

3.3 Code examples

In this section, two code examples are shown. One very simple D-Bus client and one service exporting an calculator object.

Based on the analysis of the Notification service above, I can write a simple client, that invokes the "Notify" method. I chose pydbus [6] library to make it simple. First of all, the method takes a lot of arguments as can be seen from the signature. It is necessary to read documentation for them:

```
.Notify                             method    susssasa{sv}i u            -
```

Now I can write the code, which given the dynamic nature of Python is really simple:

```
from pydbus import SessionBus

bus = SessionBus()
notifications = bus.get('.Notifications')

notifications.Notify('test', 0, 'dialog-information', \
        "Hey!", "It works!", [], {}, 50000)
```

And that is it, working D-Bus client.

Writing a service is a bit more complicated. Again, it could be done using the Python pydbus or any other library. For example Vala has very nice D-Bus support, because it was created specifically for the GObject and

[5] Source: https://www.freedesktop.org/wiki/Software/DBusBindings/

[6] Upstream: https://github.com/LEW21/pydbus

GNOME ecosystem. Nevertheless I decided to use the dbus-rs [7] library. It is written in the Rust programming language and seems to have almost stable API. It is necessary to build a D-Bus service piece by piece starting from methods and signals, followed by interfaces and objects up to the service itself. Of course it is a lot of typing to write code like this, but I think it represents the concepts defined by D-Bus nicely. It is also possible to extend this library using the Rust macro system and get a nice and simple service definition, but I am going to use it directly.

```
extern crate dbus;
extern crate calc;

use dbus::{Connection, BusType, NameFlag};
use dbus::tree::Factory;
use std::sync::Arc;

// Wrapper function, so that we can use ?
fn run_dbus_service() -> Result<(), dbus::Error> {
    // Acquire session bus instance
    let connection = Connection::get_private(BusType::Session)?;

    // Register a service name
    connection.register_name(
        "cz.sehny.service",
        NameFlag::ReplaceExisting as u32)?;

    // Factory is an abstract construction used for creation of DBus components like methods,
    // objects and interfaces
    let factory = Factory::new_fnmut::<()>();

    let signal = Arc::new(
        factory
            .signal("CalcExecuted", ())
            .sarg::<&str,_>("expression")
    );
    let signal_clone = signal.clone();

    // New method "Eval"uate expression
    let method = factory.method("Eval", (),
        // I use move to transfer ownership of the signal variable
        move |m|{
            let n: &str = m.msg.read1()?;
            let res = match calc::eval(n) {
                Ok(val) => format!("{}={}", n, val),
                Err(_) => format!("There was an error in the input expression!"),
            };
            let s = format!("[Calculator service] {}", res);
            let sig = signal
                .msg(m.path.get_name(), m.iface.get_name())
                .append(format!("{}", res));
            Ok(vec!(m.msg.method_return().append1(s), sig))
    })  .inarg::<&str,_>("args")
        .outarg::<&str,_>("result");

    let interface = factory.interface("cz.sehny.calc", ())
        .add_m(method)
        .add_s(signal_clone);

    let object = factory.object_path("/calc", ())
        .introspectable()
        .add(interface);
```

[7] Upstream: `https://github.com/diwic/dbus-rs`

```
    // In order to create an object hierarchy, I create a tree, which will than
    // contain objects specified by their paths
    let tree = factory.tree(())
        .add(object);
    tree.set_registered(&connection, true)?;
    connection.add_handler(tree);
    loop {
        connection.incoming(1000).next();
    }
}

fn main() {
    if let Err(e) = run_dbus_service() {
        eprintln!("There was an error in DBus service: {}", e);
    }
}
```

4 Alternatives

I am not aware of any other software bus implementation for Linux except for D-Bus and Bus1, but Bus1 seems to be under heavy development and its goals are not clearly stated. Finally one can use any application layer protocol over the loopback interface for IPC. JSON-RPC is a popular choice, for instance Xi-editor is using it. Or single purpose protocols can be used, like Language Server Protocol [8].

5 Final notes

This document is available on my Github together with the source codes [9].

[8] Source: `https://github.com/Microsoft/language-server-protocol`
[9] Link: `https://github.com/msehnout/dbus-article`