OBus user manual

Jérémie Dimino

July 30, 2012

Abstract

`D-Bus` is an inter-processes communication protocol, or IPC for short, which has recently become a standard on desktop oriented computers. It is now possible to talk to a lot application using `D-Bus`. Moreover, it has many bindings/implementations for differents languages, which make it easily accessible. `OBus` is a pure OCaml implementation of this protocol. What makes it different from other bindings/implementations is that it is the only one using cooperative threads, which make it very simple to fully exploit the asynchronous nature of D-Bus.

Note: it is advised to have some knowledge about the `Lwt` library before reading this manual.

Contents

1 Introduction

1.1 Overview of OBus

1.1.1 Packages

The main packages of the OBus distribution is the OBus package, available via findlib. It contains the core library. Moveover, OBus although provides packages for using a bunch of services of the Freedesktop project:

- obus.hal
- obus.notification
- obus.network-manager
- obus.policykit
- obus.udisks
- obus.upower

The use of these packages is straightforward and you need to know almost nothing about D-Bus or OBus. For example, here is a program which open a popup notification:

```
open Notification

lwt () =
  lwt id = Notification.notify ~summary:"Hello, world!" () in
  return ()
```

Lastly OBus also provides a syntax extension (package obus.syntax) and a parser/printer for the IDL language (package obus.idl).

1.1.2 Modules

OBus contains about 30 public modules. But do not be scared, most of the time you will need a very small subset of them. These modules can be divided in two categories:

- the high-level API
- the low-level API

The low-level API is described in the section 8 of this manual. Note that you must have a good knowledge of D-Bus to use it.

2 Quick start

In this section we explain how to quickly uses a D-Bus service using OBus.

- The first step is to obtain the introspection of the service. Some applications put theses file into /usr/share/dbus-1/interfaces/. Otherwise you can get it by introspecting a running service, for example:

  ```
  $ obus-introspect -rec org.foo.bar / > foo.xml
  ```

  will recursivelly introspect the service named org.foo.bar and put all the interfaces it implements into foo.xml.

- The second step is to turn this file into an ocaml module which contains the description of the interface:

  ```
  $ obus-gen-interface foo.xml
  ```

  This will create the two files `foo_interfaces.ml` and `foo_interfaces.ml`.

- The final step is to turn the introspection file into a module for client-side use:

  ```
  $ obus-gen-client foo.xml
  ```

  This will produce the two files `foo_client.mli` and `foo_client.ml`. These two files can be edited, and must be compiled with the `lwt.syntax` syntax extension.

After that, you can use `Foo_client` module to access the service. Methods are mapped to functions returning a `lwt` thread, signals are mapped to values of type `OBus_signal.t`, and properties to values of type `OBus_property.t`. For example:

```
lwt () =
  (* Connect to the session bus *)
  lwt bus = OBus_bus.session () in

  (* Create a proxy for a remote object *)
  let proxy =
    OBus_proxy.make
      (OBus_peer.make bus "org.foo.bar")
      ["org"; "foo"; "bar"]
  in

  (* Call a method of the servivce *)
  lwt result = Foo_client.Org_foo_bar.plop proxy ... in

  (* Connect to a signal of the service *)
  lwt () =
    Lwt_react.E.notify (fun args -> ...)
    =|< OBus_signal.connect (Foo_client.Org_foo_bar.plip proxy)
  in

  (* Read the contents of a property *)
  lwt value = OBus_property.get (Foo_client.Org_foo_bar.plap proxy) in

  ...
```

3 Basis

In this section we will describe the minimum you must know to use `OBus` and interfaces for `D-Bus` services written with `OBus` (like the ones provided in the `OBus` distribution: `obus.notification`, `obus.upower`, ...).

3.1 Connections and message buses

A *connection* is a way of exchanging messages with another application speaking the `D-Bus` protocol. Most of the time applications use connection to a special application called a *message bus*. A message bus act as a router between several applications. On a desktop computer, there are two well-known instances: the *system* message bus, and the user *session* message bus.

The first one is unique given a computer, and use security policies. The second is unique given a user session. Its goal is to allow programs running in the session to talk to each other. `OBus` offers two function for connecting to these message buses: `OBus_bus.session` and `OBus_bus.system`.

The session bus exists for the life-time of a user session. It exits when the session is closed, and any programs using it should exit to, that is why `OBus` will exit the program when the connection to the session bus is lost. However this behavior can be changed.

On the other hand the system bus can be restarted and program using it may try to reopen the connection. System-wide application should handle the lost of the connection with the system bus.

Here is a small example which connects the session bus and prints its id:

```
open Lwt

lwt () =
  (* Open a connection to the session message bus: *)
  lwt bus = OBus_bus.session () in

  (* Obtain its id: *)
  lwt id = OBus_bus.get_id bus in

  Lwt_io.printlf "The session bus id is %d." (OBus_uuid.to_string id)
```

3.2 Names

On a message bus, applications are referenced using names. There is a special category of names called *unique names*. Each time an application connects to a bus, the bus give it a unique name. Unique name are of the form `:1.42` and cannot be changed. You can think of a unique name as an *ip* (such as `192.168.1.42`).

Once connected, the unique name can is returned by the function `OBus_bus.name`. Here is an example of a program that prints its unique name:

```
open Lwt

lwt () =
  (* Connects to the session bus: *)
  lwt bus = OBus_bus.session () in

  (* Read our unique name: *)
  let name = OBus_bus.name bus in

  Lwt_io.printlf "My unique connection name is %s." name
```

Unique name are usefull to uniquelly identify an application. However when you want to use a specific service you may prefer using a well-known name such as `org.freedesktop.Notifications`. `D-Bus` allows applications to own as many non-unique names as they want. You can think of a non-unique name as a *dns* (such as “obus.forge.ocamlcore.org”).

Names can be requested or resolved using functions of the `OBus_bus` module.

Here is an example:

```
open Lwt

lwt () =
  lwt bus = OBus_bus.session () in

  lwt () =
```

```
    try_lwt
      (* Try to resolve a name, this may fail if nobody owns it: *)
      lwt owner = OBus_bus.get_name_owner bus "org.freedesktop.Notifications" in
      Lwt_io.printlf "The owner is %d."
    with OBus_bus.Name_has_no_owner msg ->
      Lwt_io.printlf "Cannot resolve the name: %s." msg
  in

  (* Request a name: *)
  OBus_bus.request_name bus "org.foo.bar" >>= function
    | `Primary_owner ->
        Lwt_io.printl "I own the name org.foo.bar!"
    | `In_queue ->
        Lwt_io.printl "Somebody else owns the name, i am in the queue."
    | `Exists ->
        Lwt_io.printl "Somebody else owns the name\
                       and does not want to loose it :(."
    | `Already_owner
        (* Cannot happen *)
        Lwt_io.printl "I already owns this name."
```

Note that the `OBus_resolver` module offer a better way of resolving names and monitoring name owners. See section 5.4 for details.

3.3 Peers

A *peer* represent an application accessible through a `D-Bus` connection. To uniquelly identify a peer one needs a connection and a name. The module `OBus_peer` defines the type type of peers. There are two requests that should be available on all peers: `ping` and `get_machine_id`. The first one just ping the peer to see if it is alive, and the second returns the id of the machine the peer is currently running on.

3.4 Objects and proxies

In order to export services, `D-Bus` uses the concept of *objects*. An application may holds as many objects as it wants. From the inside of the application, `D-Bus` objects are generally mapped to language native objects. From the outside, objects are refered by *object-paths*, which looks like "`/org/freedesktop/DBus`". You can think of an object path as a pointer.

Objects may have members which are organized by interfaces (such as "`org.freedesktop.DBus`"). There are three types of members:

- Methods
- Signals
- Properties

Methods act like functions. Clients can call methods of objects. Signals are spontaneous events that may occurs at any time. Clients may register to these signals and then be notified when a signal arrive. Properties act as variable, that can be read and/or written and sometimes monitored.

In order to uniquelly identify an object, we need its path and the peer that owns it. We call such a thing a *proxy*. Proxies are defined in the module `OBus_proxy`.

Here is a simple example on how to call a method on a proxy (we will explain latter what means the `C.seq...` things):

```
open Lwt
open OBus_value

lwt () =
  lwt bus = OBus_bus.session () in

  (* Create the peer: *)
  let peer = OBus_peer.make ~name:"org.freedesktop.DBus" ~connection:bus in

  (* Create the proxy: *)
  let proxy = OBus_proxy.make ~peer ~path:["org"; "freedesktop"; "DBus"] in

  (* Call a method: *)
  lwt id =
    OBus_proxy.call proxy
      ~interface:"org.freedesktop.DBus"
      ~member:"GetId"
      ~i_args:C.seq0
      ~o_args:(C.seq1 C.basic_string)
      ()
  in

  Lwt_io.printlf "The bus id is: %s" id
```

4 Interaction between the OCaml world and the D-Bus world

4.1 Value mapping

`D-Bus` defines its own type system, which is used to serialize and deserialize messages. These types are defined in the module `OBus_value.T` and `D-Bus` values that are defined in the module `OBus_value.V`. When a message is received, its contents is represented as a value of type `OBus_value.V.sequence`. Simillary, when a message is sent, it is first converted into this format.

Manipulating boxed `D-Bus` values is not very handy. To make the interaction more transparent, `OBus` defines a set of type combinators which allow to easilly switch between the `D-Bus` representation and the ocaml representation. These convertors are defined in the module `OBus_value.C`.

Here is an example of convertion (in the toplevel):

```
# open OBus_value;;

(* Make a D-Bus value from an ocaml one: *)
# C.make_sequence (C.seq2 C.basic_int32 (C.array C.basic_string)) (42l, ["foo"; "ba
- : OBus_value.V.sequence =
[OBus_value.V.Basic (OBus_value.V.Int32 42l);
 OBus_value.V.Array (OBus_value.T.Basic OBus_value.T.String,
  [OBus_value.V.Basic (OBus_value.V.String "foo");
   OBus_value.V.Basic (OBus_value.V.String "bar")])]

(* Cast a D-Bus value to an ocaml one: *)
# C.cast_sequence (C.seq1 C.basic_string) [V.basic(V.string "foobar")];;
- : string = "foobar"
```

```
(* Try to cast a D-Bus value to an ocaml one with the wrong type: *)
# C.cast_sequence (C.seq1 C.basic_string) [V.basic(V.int32 0l)];;
Exception: OBus_value.C.Signature_mismatch.
```

4.2 Errors mapping

A call to a method may fails. In this case the service sends an error to the caller. `D-Bus` errors are mapped to ocaml exceptions by the `OBus_error` module. Basically, to defines a mapping between an exception and a `D-Bus` error, here is what you have to do:

```
exception My_exn of string

let module M = OBus_error.Register(struct
                                     exception E = My_exn
                                     let name = "org.foo.bar.MyError"
                                   end)
in ()
```

Or, if you use the syntax extension:

```
exception My_exn of string
  with obus("org.foo.bar.MyError")
```

5 Using D-Bus services

In this section we describe the canonical way of using a `D-Bus` service with `OBus`.

5.1 Defining and using members

For all types of members (methods, signals and properties), `D-Bus` provides types to defines them and functions to use these definitions. A member definition contains all the information about a member. For example, here is the definition of a method call named "foo" on interface "org.foo.bar" which takes a string and returns an 32-bits signed integer:

```
open OBus_member

let m_Foo = {
  Method.interface = "org.foo.bar";
  Method.member = "Foo";
  Method.i_args = C.seq1 C.basic_string;
  Method.o_args = C.seq1 C.basic_int32;
  Method.annotations = [];
}
```

Once a member is defined, it can be used by the corresponding modules:

```
open Lwt
open OBus_members

(* Definition of a method *)
let m_GetId = {
  Method.interface = "org.freedesktop.DBus";
  Method.member = "GetId";
```

```
  Method.i_args = C.seq0;
  Method.o_args = C.seq1 C.basic_string;
  Method.annotations = [];
}

(* Definition of a signal *)
let s_NameAcquired = {
  Signal.interface = "org.freedesktop.DBus";
  Signal.member = "NameAcquired";
  Signal.args = C.seq1 (C.basic C.string);
  Signal.annotations = [];
}

lwt () =
  lwt bus = OBus_bus.session () in
  let proxy =
    OBus_proxy.make
      (OBus_peer.make bus "org.freedesktop.DBus")
      ["org"; "freedesktop"; "DBus"]
  in

  (* Call the method we just defined: *)
  lwt id = OBus_method.call m_GetId proxy () in

  (* Register to the signal we just defined: *)
  lwt event = OBus_signal.connect (OBus_signal.make s_NameAcquired proxy) in

  Lwt_react.E.notify_p
    (fun name ->
       Lwt_io.printlf "name acquired: %s" name)
    event;

  Lwt_io.printlf "The message bus id is %s" id
```

Of course, writting definitions by hand may be very boring and error-prone. To avoid that `OBus` can automatically convert introspection data into ocaml definitions.

5.2 Using tools to generate member definitions

There are two tools that are usefull for client-side code: `obus-gen-interface` and `obus-gen-client`. The first one converts an xml introspection document (or an idl file) into an ocaml module containing all the camlized definitions. This generated file is in fact also needed for server-side code. Note that fiels produced by `obus-gen-interface` are not meant to be edited.

The second tool maps members into their ocaml counterpart: methods are mapped to functions, signals to value of type `OBus_signal.t` and properties to values of type `OBus_property.t`. This generated file is meant to be edited. For example, you can edit it in order to change the type of values taken/returned by methods.

5.3 The `OBus` IDL language

Since editing XML is horrible, `OBus` provides a intermediate language to write `D-Bus` interfaces. Moreover this language allow you to automatically converts integers to ocaml variants when needed.

The syntax is pretty simple. Here is an example, taken from OBus sources (file src/oBus_interfaces.obus):

```
interface org.freedesktop.DBus {
  (** A method definition: *)
  method Hello : () -> (name : string)

  (** Bitwise flags definition: *)
  flag request_name_flags : uint32 {
    0b001: allow_replacement
    0b010: replace_existing
    0b100: do_not_queue
  }

  (** Definition of an enumeration: *)
  enum request_name_result : uint32 {
    1: primary_owner
    2: in_queue
    3: exists
    4: already_owner
  }

  (** A method that use newly defined types: *)
  method RequestName :
    (name : string, flags : request_name_flags)
    ->
    (result : request_name_result)
}
```

All OBus tools that accept XML files also accept IDL files. Moreover it is possible to convert them by using obus-idl2xml and obus-xml2idl.

5.4 Name tracking

The owner of a on-unique name may change over the time. OBus provides the OBus_resolver module to deals with it. The owner is mapped into a React's signal holding the current owner of a name.

6 Writing D-Bus services

In this section we describe the canonical way of writing D-Bus services with OBus.

Local D-Bus objects are represented by values of type OBus_object.t. The main operations on objects are: adding an interface and exporting it on a connection. Exporting an object means making it available to all peers reachable from the connection.

In order to add callable methods to objects you have to create interfaces descriptions (of type 'a OBus_object.interface) and add them to objects.

The canonical way to create interfaces with OBus is to first write its signature in an XML introspection file or in an OBus idl file, then convert it into an ocaml definition module with obus-gen-interface and in a template ocaml source file with obus-gen-server.

Here is a small example of interface:

```
interface org.Foo.Bar {
  method GetApplicationName : () -> (name : string)
    (** Returns the name of the application *)
}
```

It is converted with:

```
$ obus-gen-interface foobar.obus -o foobar_interfaces
file "foobar_interfaces.ml" written
file "foobar_interfaces.mli" written
$ obus-gen-server foobar.obus -o foobar
file "foobar.ml" written
```

Now all that you have to do is to edit the file generated by `obus-gen-server` and replace the "Not implemented" errors by your code.

Once it is done, here is how to actually create the object, add the interface and export it:

```
lwt () =
  lwt bus = OBus_bus.session () in

  (* Request a name: *)
  lwt _ = OBus_bus.request_name bus "org.Foo.Bar" in

  (* Create the object: *)
  let obj =
    OBus_object.make
      ~interfaces:[Foobar.Org_Foo_Bar.interface]
      ["plip"]
  in

  (* Attach it some data: *)
  OBus_object.attach obj ();

  (* Export the object on the connection *)
  OBus_object.export bus obj;

  (* Wait forever *)
  fst (wait ())
```

Note the you can attach custom data to the object with `OBus_object.attach`.

7 One-to-one communication

Instead of connection to a message bus, you may want to directly connects to another application. This can be done with `OBus_connection.of_addresses`.

If you want to allow other applications to connect to your application then you have to start a server. Starting a server is very simple, all you have to do is to call `OBus_server.make` with a callback that will receive new connections.

8 Low-level use of D-Bus

This section describes the low-level part of `OBus`.

8.1 Message filters

Message filters are function that are applied to all incomming/outgoing messages. Filters are of type:

```
type filter = OBus_message.t -> OBus_message.t option
```

Each filter may use and/or modify the message. If `None` is returned the message is dropped.

8.2 Matching rules

When using a message bus, an application do not receive messages that are not destined to it. In order to receive such messages, one need to add rules on the message bus. All messages matching a rule are sent to the application which defined that rule.

There are two ways of adding matching rules: by using the module `OBus_bus`, or by using `OBus_match`. The functions `OBus_bus.add_match` and `OBus_bus.remove_match` are directly mapped to the corresponding methods of the message bus. The function `OBus_match.export` is more clever:

- it exports only one time duplicated rules,
- it exports only the most general rules.

We say that a rule `r1` is more general that a rule `r2` if all messages matched by `r2` are also matched by `r1`. For example a rule that accept all messages with interface field equal to `foo.bar` is more general that a rule that accept all messages with interface field equal to `foo.bar` and with member field equal to `plop`.

Note that you must be carefull if you use both modules that automatically manage rules (such as `OBus_signal`, `OBus_resolver` or `OBus_property`) and `OBus_bus.add_match` or `OBus_bus.remove_match`.

8.3 Defining new transports

A transport is a way of receiving and sending messages. The `OBus_transport` allow to defines new transports. If you want to create a new transport that use the same serialization format as default transport, then you can use the `OBus_wire` module.

By definning new transports, you can for example write an application that forward messages over the network in a very few lines of code.

8.4 Defining new authentication mechanisms

When openning a connection, before we can send and receive message over it, `D-Bus` requires a authentication procedure. `OBus` implements both client and server side authentication. The `OBus_auth` allow to write new client and server side authentication mechanisms.