Programming the D-Bus

Michaël Van Canneyt

November 3, 2010

Abstract

In a previous article, the workings of D-Bus were explained, and it was shown how to use the D-Bus through command-line utilities and scripting. However, D-BUS offers a rich programming interface for a variety of programming languages, which is presented here.

1 Introduction

The D-BUS workings were explained in a the previous issue of FreeX: D-Bus is a system message bus, which can be used for RPC purposes as well. The messaging system is simple and yet powerful, and can be manipulated with the aid of some command-line utilities (dbus-send)- and some GUI tools as well (qdbusviewer).

The real power of D-Bus lies in it's rich programming interface. D-Bus itself is implemented in C, so the low-level C API is the 'natural' API for the D-BUS. However, it is rather cumbersome to use, so various bindings for the D-Bus exist:

- A Glib binding - also in C, but easier to use.
- A Qt wrapper (in C++)
- A Python wrapper
- A Perl wrapper
- A .NET wrapper (mainly for use in Mono)
- A ruby wrapper
- A Java wrapper
- A Pascal wrapper

Some of these wrappers are quite advanced, others are a simple shell around the C API.

The more advanced wrappers (Qt, Pascal) use the introspection methods to generate specialized source files that describe the interface in the language of the binding. For C, this binding is generated using dbus-binding-tool. When using C++ and Qt, the qdbusxml2cpp tool can be used to generate the necessary bindings. Both take a XML file as input and produce C or C++ code as output.

For Object Pascal, a similar tool exist: either a command-line tool, or a GUI tool, similar to qdbusviewer can be used to create D-BUS bindings. The principles of all these tools are the same, we'll study the Pascal version in this article.

2 The low-level API

To appreciate the work that the binding generators do, it is instructive to study the low-level C API. The Object Pascal implementation contains a one-on-one translation of the C API: all calls can be done in Pascal as they would be done in C, but string handling is less cumbersome, so that will be used. The demonstration will call the introspection interface to get the calls exposed by the DBUS daemon itself. The introspection for this interface looks as follows:

```
<interface name="org.freedesktop.DBus.Introspectable">
  <method name="Introspect">
    <arg name="data" direction="out" type="s"/>
  </method>
</interface>
```

This contains all that is needed to call this method, as will be shown below.

All D-BUS programs start with setting up a connection to the D-BUS. This is done with the `dbus_get_private` call, defined as follows:

```
function dbus_bus_get_private(
   atype: DBusBusType;
   error: PDBusError): PDBusConnection;
```

The first parameter identifies the bus to connect to, and is a pre-defined constant: One of `DBUS_BUS_SYSTEM`, `DBUS_BUS_SESSION`.

The call will return a pointer to an opaque connection record. Any errors are reported in `error`, and can be checked with `dbus_error_is_set`

```
function dbus_error_is_set(error: PDBusError): dbus_bool_t;
```

Which will return `True` if an error is reported. The error structure must be initialized with `dbus_error_init` before it can be used.

Putting these things together, it means that the following code can be used to set up a connection to the session bus:

```
var
  err: DBusError;
  conn: PDBusConnection;

begin
  dbus_error_init(@err);
  conn:=dbus_bus_get_private(DBUS_BUS_SESSION, @err);
  if dbus_error_is_set(@err) <> 0 then
    begin
    WriteLn('Connection Error: ' + err.message);
    dbus_error_free(@err);
    end;
```

Once a connection to the D-BUS is made, a message can be created and sent. D-BUS uses messages for everything. Depending on what one wants to do, a different kind of message must be created, and for each kind of message a API call exists to create the message. In the current example, a method call must be performed, and for this a 'method call message' exists: this message must be sent to the service, which will reply with a response message. The following call initializes such a method:

```
function dbus_message_new_method_call(
    const bus_name, path,
          interface_name, method: PChar): PDBusMessage;
```

The function takes 4 arguments:

bus_name Name of the program listening on the D-Bus. For the example, this is the D-Bus daemon itself, which uses the name 'org.freedesktop.DBus' (names are case sensitive).

path Object path: if the program exposes multiple objects, then a path to the object must be given. The D-bus daemon only exposes itself, so this is the root object '/'

interface_name The interface one wishes to use. For the example, this is the 'org.freedesktop.DBus.Introspectable' interface.

method Finally, the name of the method, in this example, this is the 'Introspect' method.

Once the message is created, the arguments (parameters) that the method expects, must be appended to it; arguments are a kind of attachments to the message, they are appended to the message using an 'iterator'. In the example, no arguments are expected, so none must be appended, but the same concept is used when the result of the method must be retrieved.

All this results in the following code:

```
  msg:=dbus_message_new_method_call('org.freedesktop.DBus',
                                    '/',
                                    'org.freedesktop.DBus.Introspectable',
                                    'Introspect');
  if (msg=nil) then
    begin
    WriteLn('Could not construct message: ' + err.message);
    dbus_error_free(@err);
    end;
```

Once constructed, the message must be sent to the D-Bus daemon. This can be done in a variety of ways:

```
 function dbus_connection_send(
    connection: PDBusConnection;
    message_: PDBusMessage;
    client_serial: Pdbus_uint32_t): dbus_bool_t;
function dbus_connection_send_with_reply(
    connection: PDBusConnection;
    message_: PDBusMessage;
    pending_return: PPDBusPendingCall;
    timeout_milliseconds: cint): dbus_bool_t;
function dbus_connection_send_with_reply_and_block(
    connection: PDBusConnection;
    message_: PDBusMessage;
    timeout_milliseconds: cint;
   error: PDBusError): PDBusMessage;
```

Despite the fact that they serve the same function, the three calls differ quite a lot. The first function sends a message, and returns a unique message identifier. It does not wait for a reply: receiving the reply must be done by waiting for incoming messages and intercepting the reply message (using the appropriate APIS functions). The second sends a message, and registers a callback which will be executed when a reply is received. The last one sends the message, and waits for a reply message: the reply message is returned by the function.

The last function is the easiest function to use for the example:

```
  reply:=dbus_connection_send_with_reply_and_block(conn,msg,
                                                    1000,@err);
  if (dbus_error_is_set(@err)<>0) then
    begin
    WriteLn('Error waiting for reply: ' + err.message);
    dbus_error_free(@err);
    end;
```

If a reply message was received, the return values must be processed: there can be more than one value, but in the case of 'Instrospect', there is only 1 value.

Processing the return values is done in the same manner as attaching values to a message when sending a message: the return values are 'arguments' of the reply message, and must be processed with an 'iterator'. The iterator is a structure which points to the 'arguments' of the message: the pointer can be moved to the next argument, or can be used to examine the type and value of the argument.

The iterator must be initialized before it can be used:

```
function dbus_message_iter_init(amessage: PDBusMessage;
                                iter: PDBusMessageIter): dbus_bool_t;
```

The function initializes the iterator `iter` for the message `amessage`: it then points to the first argument of the message. If there are no attachments, the return value of the function is 'False'.

Once the iterator is initialized, the type of the argument can be examined:

```
function dbus_message_iter_get_arg_type(
   iter: PDBusMessageIter): cint;
```

The function returns the type of the argument the iterator is currently pointing to. The return value is one of the predefined constants in the following table:

value	meaning
DBUS_TYPE_BYTE	8 bits signed integer
DBUS_TYPE_BOOLEAN	Boolean value
DBUS_TYPE_INT16	16 bits signed integer
DBUS_TYPE_UINT16	16 bits unsigned integer
DBUS_TYPE_INT32	32 bits signed integer
DBUS_TYPE_UINT32	32 bits unsigned integer
DBUS_TYPE_INT64	64 bits signed integer
DBUS_TYPE_UINT64	64 bits unsigned integer
DBUS_TYPE_DOUBLE	Double precision float
DBUS_TYPE_STRING	String
DBUS_TYPE_OBJECT_PATH	Object path
DBUS_TYPE_SIGNATURE	GUID
DBUS_TYPE_ARRAY	Array
DBUS_TYPE_VARIANT	Variant

These types correspond to the types shown in the introspection XML sample in the previous article. The size of the argument is very important, because retrieving an argument's value is performed using a plain memory buffer, using the following call:

```
procedure dbus_message_iter_get_basic(iter: PDBusMessageIter;
                                      value: Pointer);
```

No checking whatsoever is performed to see whether the buffer is of the right size; it is therefor very important to check the type of the argument before retrieving a value. In the case of the example, the `Introspect` method returns a string, which is retrieved as follows:

```
  if (dbus_message_iter_init(reply, @args) = 0) then
    writeln('No return value')
  else if (DBUS_TYPE_STRING<>dbus_message_iter_get_arg_type(@args)) then
    Writeln('Return value is not a string')
  else
    dbus_message_iter_get_basic(@args, @res);
  if (res<>Nil) then
    Writeln(res);
```

The code starts by initializing an iterator (`args`), and checking the type of the argument. If the argument is a string, its value is retrieved in `res`, which is a `pchar`. String values are not actually returned by D-BUS: they remain as part of the message: if the value must be manipulated somehow, then it must be copied first.

Both the message and it's attachments (the arguments) must be disposed of when the program is done with them. This can be done with the `dbus_message_unref` call: it de-references the message - as soon as no-one is referencing the message, it is disposed of by the D-bus implementation. In the case of our sample program both the sent message and the reply must be disposed of. After that, the connection can be closed:

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

The whole program is roughly over 50 lines. For a simple method call, with no parameters and a single return value, this is quite some code.

3 A proper binding

For pascal, there are 2 ways to operate the D-BUS without recourse to the C api. The first is using a set of classes which are simple wrappers around the C API: they do not hide alter the working of the C api, but offer a more native Object Pascal interface using classes. These classes are implemented in the dbuscomp unit. The same sample program implemented using the classes looks as follows:

```
program testdbuscomp;
{$mode objfpc}{$h+}
uses dbuscomp;

Var
  Conn : TDBUSConnection;
  S : String;
  M : TDBUSMethodCallMessage;
  R : TDBUSMessage;
```

```
begin
  Conn:=TDBUSConnection.Create(Nil);
  try
    Conn.kind:=ckSession;
    Conn.Connect;
    M:=TDBUSMethodCallMessage.Create(
       'org.freedesktop.DBus',
       '/',
       'org.freedesktop.DBus.Introspectable',
       'Introspect');
    R:=Conn.SendWithReplyAndBlock(M,1000);
    R.GetArgument(S);
    Writeln(S);
  finally
    R.Free;
    M.Free;
    Conn.Free;
  end;
end.
```

The same way of working is used: create a connection (using `TDBUSConnection`), a message `TDBUSMethodCallMessage` and then send the message, waiting for a reply.. Afterwards, the attachments are examined to read the return value.

This code is more pascal-ish than the original code. Still, it does not hide the mechanisms of D-BUS: creating a message, sending it, waiting for a reply message, and then extracting the result from the reply message.

To hide this complexity, a third layer exists. This layer works using introspection and a tool to generate code. 2 tools exist; one command-line tool, and one GUI tool (fpdbusview). The GUI tool is displayed in figure 1 on page 7. The setup is rather similar to the qdbusviewer: the top-left displays the services on the D-BUS. Selecting a service will display the interfaces exposed by the service's root object in the right treeview, as well as any objects it contains (a double click on an object will load the interfaces for that object).

Once an object was selected, Pascal code can be generated which describes the interface. For the introspectable interface, the following code is generated:

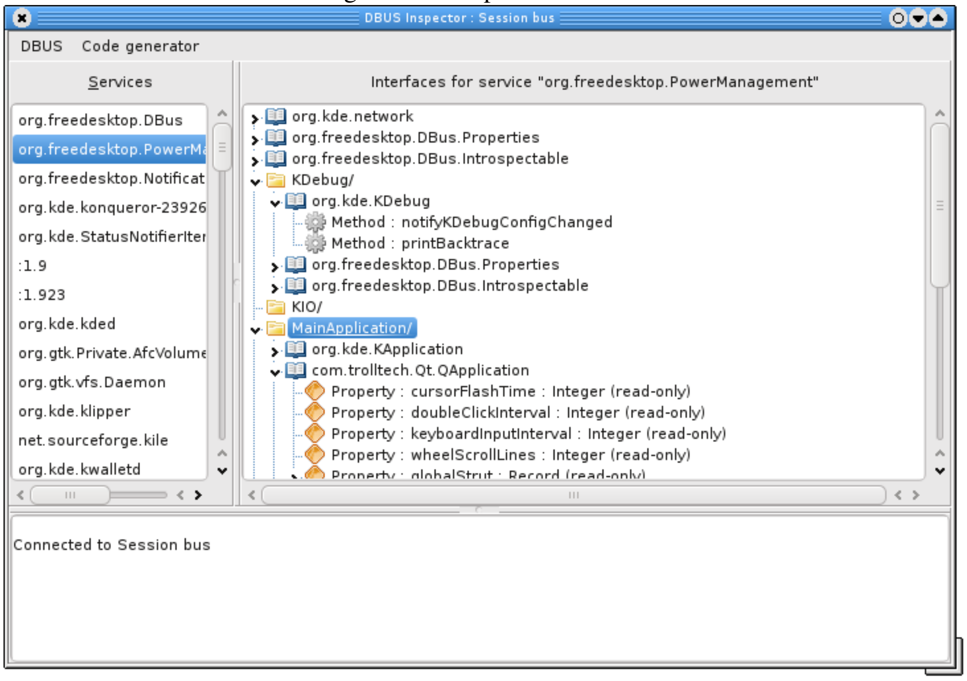
```
 Const
   Sorg_freedesktop_DBus_Properties_Name =
      'org.freedesktop.DBus.Properties';

Type
  { org.freedesktop.DBus.Introspectable
    -> org_freedesktop_DBus_Introspectable }

  Iorg_freedesktop_DBus_Introspectable = Interface
    Function Introspect : String;
  end;

  Torg_freedesktop_DBus_Introspectable_proxy = Class(TDBUSProxy)
  Public
    Function Introspect : String;
  end;
```

Figure 1: The fpdbusview tool


DBUS Inspector : Session bus
DBUS
Code generator
Services
Interfaces for service "org.freedesktop.PowerManagement"
org.freedesktop.DBus
org.freedesktop.PowerMa
org.freedesktop.Notificat
org.kde.konqueror-23926
org.kde.StatusNotifierIter
:1.9
:1.923
org.kde.kded
org.gtk.Private.AfcVolume
org.gtk.vfs.Daemon
org.kde.klipper
net.sourceforge.kile
org.kde.kwalletd
org.kde.network
org.freedesktop.DBus.Properties
org.freedesktop.DBus.Introspectable
KDebug/
org.kde.KDebug
Method : notifyKDebugConfigChanged
Method : printBacktrace
org.freedesktop.DBus.Properties
org.freedesktop.DBus.Introspectable
KIO/
MainApplication/
org.kde.KApplication
com.trolltech.Qt.QApplication
Property : cursorFlashTime : Integer (read-only)
Property : doubleClickInterval : Integer (read-only)
Property : keyboardInputInterval : Integer (read-only)
Property : wheelScrollLines : Integer (read-only)
Property : globalStrut : Record (read-only)

Connected to Session bus

As can be seen, 2 things are created: an interface definition corresponding to the Introspectable interface, and then a proxy object. The proxy object exposes the interface: An instance of this object can be created, and offers the methods that the interface defines (in the case of the example, it has an Introspect function).

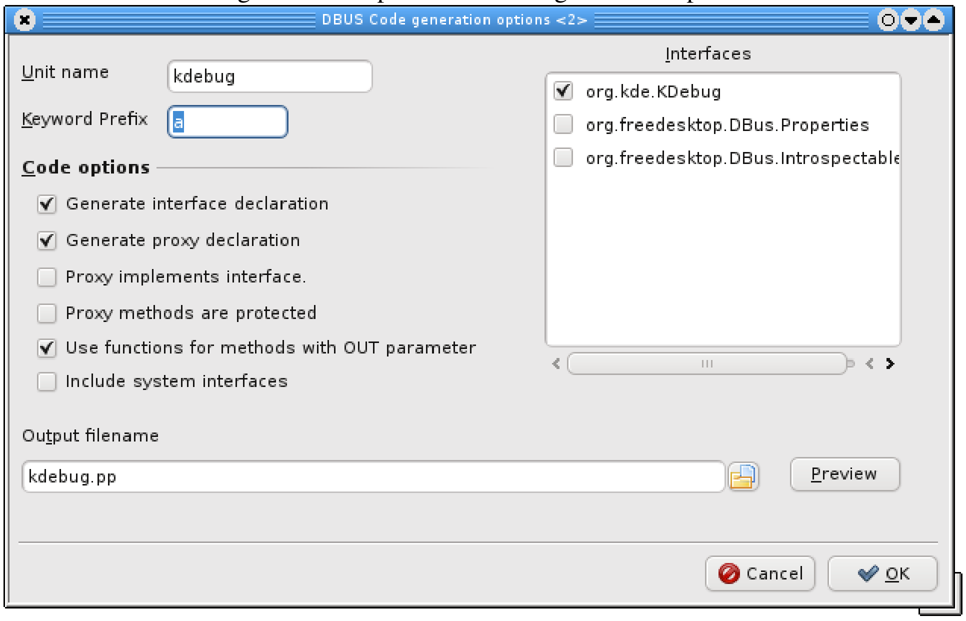
The generated code can be saved to a unit, and used as follows:

```
Conn:=TDBUSConnection.Create(Self);
try
  Conn.Kind:=ckSession;
  Conn.Connect;
  P:=Torg_freedesktop_DBus_Introspectable_proxy.create(Self);
  try
    P.Connection:=Conn;
    P.ObjectPath:='/';
    P.Destination:='org.freedesktop.DBus';
    Writeln(P.Introspect);
  finally
    P.Free;
  end;
finally
  Conn.Free;
end;
```

The proxy object hides the details of sending a messageand waiting for a reply. Instead, it exposes a method, just as the original service would expose it. All communication details are hidden from the user: the creation of the method, waiting for a reply end extracting results is taken care of by the generated code.

For the example, the needed code is not much less than when using the pascal classes, but for interfaces that have lots of methods with many parameters, the reduction of the number

Figure 2: The fpdbusview code generation options


DBUS Code generation options <2>

Unit name: kdebug

Keyword Prefix: a

Code options

- [✓] Generate interface declaration
- [✓] Generate proxy declaration
- [ ] Proxy implements interface.
- [ ] Proxy methods are protected
- [✓] Use functions for methods with OUT parameter
- [ ] Include system interfaces

Interfaces

- [✓] org.kde.KDebug
- [ ] org.freedesktop.DBus.Properties
- [ ] org.freedesktop.DBus.Introspectable

Output filename: kdebug.pp

of lines of code is significant.

There is also a command-line tool (dbus2pas) that performs the same function. It can connect to a service and create an interface, or it can read the introspection data from a file: the result is the same: a file describing the interface. There are various options that influence the generated code, the dialog is presented in figure 2 on page 8 The options in the dialog are mostly self-explanatory: a list of interfaces for which to generate code, the name of the unit in which to save the code, a prefix to use in case method or argument names are reserved words, and some options. The code can be previewed to see the effect of the various options, and pressing 'Ok' will generate the unit and save it to the indicated filename.

4 conclusion

Programming the D-BUS is easy, and may be done in a variety of ways. The tool presented here shows how to do this in pascal, but similar tools exist for C, C++ and other languages: the way of working is approximately the same. The discussion here presents only a mechanism for programming a D-BUS client: a program that uses methods exposed by existing services. An explanation of how to program a D-BUS server is left to a future contribution.