Free, one day ...

Wednesday, November 7 2012

Journey into the world of DBus for C++ (part 2) - attempt around libdbus-c++

By Emilien Kia on Wednesday, November 7 2012, 08:00 - dev

This entry is following the first part of the journey.

We will try to create an example with a little client and a little server around a little service called *com.example.Hello* which will propose to say hello: the client will call a method *sayHello* passing it a name in parameter and the server will perform the action by dispalying it in the terminal.

Immersion in the doc

libdbus-c+ + providing an inline documentation (http://dbus-cplusplus.sourceforge.net/index.html) generated by Doxygen (http://www.stack.nl/~dimitri/doxygen/) (also provided by installing the *libdbus-c++-doc* package), we will start by trying to write our little test inspired from it.

First reef: the documentation does not provide introduction, tutorial nor usage related pages. The doc is composed only from few comments related to few classes or functions, but nothing really usable.

So we look at the class list (http://dbus-cplusplus.sourceforge.net/annotated.html) . We find a class called DBus::Connection (http://dbus-cplusplus.sourceforge.net/structDBus_1_1Connection.html) which provides 3 static functions *SystemBus()*, *SessionBus()* and *ActivationBus()* which returns a *Connection* object.

Interesting! They should provide a way to connect to different busses (session, system and activation).

But the simple instantiation to the session bus by a simple call to *DBus::Connection::SessionBus()* miserably fails because there is no default *dispatcher* set. We remain skeptical.

Some searches on the net does not satisfy us. Lack of documentation really penalizing us.

The unique really usable example on the net

After catastrophic immersion in the (lack of) doc, I have intended to search an example on the net.

Surprise ... only one example is really documented : 3 pages on a wiki of a project I did not known called Apertium (http://www.apertium.org/) (automatic translation) including:

- DBus usage examples in many languages including the C++ with libdbus-c++
- One detailled example (http://wiki.apertium.org/wiki/Compiling_a_C%2B%2B_D-Bus_program) of a program using libdbus-c++

And here is the revelation: we must use the *dbusxx-xml2cpp* tool to produce C++ header files declaring classes representing our DBus objects.

Moreover a little `man dbusxx-xml2cpp` says:

```
DBUSXX-XML2CPP(1)                                      DBUSXX-XML2CPP(1)

NAME
       dbusxx-xml2cpp - generate proxy and adapter interfaces.

SYNOPSIS
       dbusxx-xml2cpp [xmlfile] [--proxy=outfile.h] [--adaptor=outfile.h]

DESCRIPTION
       dbusxx-xml2cpp  generates proxy and adapter interfaces from modified D-
       Bus XML introspection documents.

OPTIONS
       --proxy=outfile.h
              Generate a proxy interface to outfile.h.

       --adapter=outfile.h
              Generate an adapter interface to outfile.h.
```

```
SEE ALSO
       dbusxx-introspection(1)

AUTHOR
       This manual page was written by Vincent Cheng <Vincentc1208@gmail.com>,
       for the Debian project (and may be used by others).

libdbus-c++-dev                    May 2011                  DBUSXX-XML2CPP(1)
```

Just have to write a XML file describing wanted service and to process it to generate needed stub codes.

So, this is our XML service file:

```
 <?xml version="1.0" encoding="UTF-8" ?>
 <node name="/com/example/Hello">
   <interface name="com.example.Hello">
     <method name="SayHello">
       <arg type="s" name="name" direction="in" />
     </method>
   </interface>
 </node>
```

And files produced by the call `dbusxx-xml2cpp service.xml - -proxy=proxy.hpp - -adaptor=adaptor.hpp`:

- A wonderful proxy class with a nice function *void SayHello(const std::string& name)*:

```
/*
 *      This file was automatically generated by dbusxx-xml2cpp; DO NOT EDIT!
 */

#ifndef __dbusxx__proxy_hpp__PROXY_MARSHAL_H
#define __dbusxx__proxy_hpp__PROXY_MARSHAL_H

#include <dbus-c++/dbus.h>
#include <cassert>

namespace com {
namespace example {

class Hello_proxy
: public ::DBus::InterfaceProxy
{
public:

    Hello_proxy()
    : ::DBus::InterfaceProxy("com.example.Hello")
    {
    }

public:

    /* properties exported by this interface */
public:

    /* methods exported by this interface,
     * this functions will invoke the corresponding methods on the remote objects
     */
    void SayHello(const std::string& name)
    {
        ::DBus::CallMessage call;
        ::DBus::MessageIter wi = call.writer();

        wi << name;
        call.member("SayHello");
        ::DBus::Message ret = invoke_method (call);
    }


public:
    /* signal handlers for this interface
     */

private:

    /* unmarshalers (to unpack the DBus message before calling the actual signal handler)
     */
};

} }
#endif //__dbusxx__proxy_hpp__PROXY_MARSHAL_H
```

- Another wonderful adaptor class with a nice abstract method *virtual void DisBonjour(const std::string& name) = 0* to define in order to specify its behavior:

```
/*
 *      This file was automatically generated by dbusxx-xml2cpp; DO NOT EDIT!
 */

#ifndef __dbusxx__adaptor_hpp__ADAPTOR_MARSHAL_H
#define __dbusxx__adaptor_hpp__ADAPTOR_MARSHAL_H

#include <dbus-c++/dbus.h>
#include <cassert>

namespace com {
namespace example {

class Hello_adaptor
: public ::DBus::InterfaceAdaptor
{
public:

    Hello_adaptor()
    : ::DBus::InterfaceAdaptor("com.example.Hello")
    {
        register_method(Hello_adaptor, SayHello, _SayHello_stub);
    }

    ::DBus::IntrospectedInterface *introspect() const
    {
        static ::DBus::IntrospectedArgument SayHello_args[] =
        {
            { "name", "s", true },
            { 0, 0, 0 }
        };
        static ::DBus::IntrospectedMethod Hello_adaptor_methods[] =
        {
            { "SayHello", SayHello_args },
            { 0, 0 }
        };
        static ::DBus::IntrospectedMethod Hello_adaptor_signals[] =
        {
            { 0, 0 }
        };
        static ::DBus::IntrospectedProperty Hello_adaptor_properties[] =
        {
            { 0, 0, 0, 0 }
        };
        static ::DBus::IntrospectedInterface Hello_adaptor_interface =
        {
            "com.example.Hello",
            Hello_adaptor_methods,
            Hello_adaptor_signals,
            Hello_adaptor_properties
        };
        return &Hello_adaptor_interface;
    }
public:

    /* properties exposed by this interface, use
     * property() and property(value) to get and set a particular property
     */

public:

    /* methods exported by this interface,
     * you will have to implement them in your ObjectAdaptor
     */
    virtual void SayHello(const std::string& name) = 0;

public:

    /* signal emitters for this interface
     */

private:

    /* unmarshalers (to unpack the DBus message before calling the actual interface method)
     */
    ::DBus::Message _SayHello_stub(const ::DBus::CallMessage &call)
    {
        ::DBus::MessageIter ri = call.reader();

        std::string argin1; ri >> argin1;
        SayHello(argin1);
        ::DBus::ReturnMessage reply(call);
        return reply;
    }
};

} }
#endif //__dbusxx__adaptor_hpp__ADAPTOR_MARSHAL_H
```

Whaou ! That’s great. It produces stubs which look really easilly usable and in the good spirit of C++.

But a problem arises: how do we use them ?

The use of examples

Finally, I resigned myself to download the library source archive and … surprise … it contains a subdirectory *examples* and a sub-subdirectory *echo* which contains client and server example.

Overloading of generated classes

Looking a little more precisely in the examples, just have to create a class which override the generated proxy class (adaptor class respectively) and *DBus::IntrospectableProxy* and *DBus::ObjectProxy* classes (*DBus::IntrospectableAdaptor* and *DBus::ObjectAdaptor* classes respectively), and which implement abstract methods if any. Moreover, some parameters can be specified in constructors.

```
class Hello : public com::example::Hello_proxy,
                    public DBus::IntrospectableProxy,
                    public DBus::ObjectProxy
{
public:
     Hello(DBus::Connection &connection, const char *path, const char *name):
     DBus::ObjectProxy(connection, path, name)
     {
     }
};

class Hello : public com::example::Hello_adaptor,
                    public DBus::IntrospectableAdaptor,
                    public DBus::ObjectAdaptor
{
public:
     Bonjour(DBus::Connection &connection):
          DBus::ObjectAdaptor(connection, "/com/example/Hello")
     {
     }

     virtual void SayHello(const std::string& name)
     {
          std::cout << "Hello '" << name << "'" << std::endl;
     }

};
```

Invocation

Invocation is not more complicated, just:

1. instantiate a *DBus::BusDispatcher* object then specify it that we will use it as default dispatcher;
2. retrieve an instance to the wanted bus (session or system);
3. then, depending of the side where the code is placed:
   - server side:
     1. instantiate adaptors and specify them the bus to connect to
     2. *enter* in the dispatcher, i.e. let it doing its job to wait for invocations
   - client side:
     1. instantiate proxies and specifying busses, object pathes and names
     2. call methods from proxy objects

So, there are bits of simple codes:

- client side:

```
DBus::BusDispatcher dispatcher;

int main(int argc, char** argv)
{
    DBus::default_dispatcher = &dispatcher;
    DBus::Connection bus = DBus::Connection::SessionBus();

    Hello hello(bus, "/com/example/Hello", "com.example.Hello");

    if(argc>=2)
        hello.SayHello(argv[1]);
    else
        hello.SayHello("world");

    return 0;
}
```

- server side:

```
DBus::BusDispatcher dispatcher;

int main()
{
    DBus::default_dispatcher = &dispatcher;
    DBus::Connection bus = DBus::Connection::SessionBus();

    bus.request_name("com.example.Hello");

    Hello hello(bus);

    dispatcher.enter();

    return 0;
}
```

That’s it !

Monday, November 5 2012

Journey into the world of DBus for C++ (part 1)

By Emilien Kia on Monday, November 5 2012, 13:55 - dev

- cpp
- dbus

DBus (http://dbus.fredesktop.org) , no need to present it anymore, is one of master piece of current systems.

It is everywhere, used by lots of applications. It makes application communication easy and provides a simple model for local service providing. Continuing the good old unix tradition, it does little but does it well.

We will intend to look for develop with DBus in C++.

The beginning of the Odyssey

The biggest problem for developers is its documentation or its lack of documentation. And especially its lack of documentation for C++.

The official DBus documentation - found on the FreeDesktop site - is really disparate. We can found:

- A tutorial (http://dbus.freedesktop.org/doc/dbus-tutorial.html) which present DBus, how it works, but introduces samples which use GLib binding. As well see the GNOME developer center (http://developer.gnome.org/) and their pages related to high level (http://developer.gnome.org/gio/stable/gdbus-convenience.html) and low level (http://developer.gnome.org/gio/stable/gdbus-lowlevel.html) DBus support. Moreover, GLib approach constrains us to link with GLib.
- One Python tutorial (http://dbus.freedesktop.org/doc/dbus-python/doc/tutorial.html) interesting but in Python. We can refer DBus Python overview (http://packages.python.org/txdbus/dbus_overview.html) which provides some useful details to understand the protocol. But it is useless to develop in C++.
- The binding page (http://www.freedesktop.org/wiki/Software/DBusBindings) which present some approaches of C+ + usages (dbus-cpp (http://www.freedesktop.org/wiki/Software/dbus-cpp) , dbus-c++ (http://www.freedesktop.org /wiki/Software/dbus-c%2B%2B) or dbus-cxx) but it looks like the rat race and the status of these projects are not really clear.

I would add some pages found by my favorite web searcher:

- A fast and clean overview (http://www.matthew.ath.cx/misc/dbus) of the low level DBus API. Sharp and clear. Can be useful.
- DBus for GLibmm (http://developer.gnome.org/glibmm/stable/group__DBus.html) so for C++. To keep in mind.

Event the good old Wikipedia (http://en.wikipedia.org/wiki/D-Bus) does not really help.

Finally, I have found 2 libraries hosted on SourceForge: dbus-cxx (http://dbus-cxx.sourceforge.net/) and libdbus-c++ (http://dbus-cplusplus.sourceforge.net/index.html) .

A difficult choice

dbus-cxx

dbus-cxx (http://dbus-cxx.sourceforge.net/) is distributed on GPL 3. Its documentation, generated with Doxygen, seems sufficiently complete and offers easy samples to start. It announces being distributed in Fedora 9+ and in Ubuntu with a dedicated PPA (from Launchpad).

libdbus-c++

libdbus-c++ (http://dbus-cplusplus.sourceforge.net/index.html) is distributed on LGPL 2.1+ Its documentation, generated with Doxygen, is restricted to the minimum. No sample, no tutorial. It announces no support in any favorite distributions but a little look at my synaptic (LinuxMint 13 - Maya) show me that it is directly available (lib, dev and doc). It provides two little tools with the package:

- *dbusxx-introspect* which introspects a service and serialize its description in XML.
- *dbusxx-xml2cpp* which read an introspected XML file and produces some C++ headers with proxy classes.

In view of the various arguments, we will try to investigate around the libdbus-c++.

Hi free world

By Emilien Kia on Monday, November 5 2012, 12:09 - site

Hi free world.

Just few words to present this blog. Following pages are generally english equivalent of the french blog.