D-Bus and Interoperability

How You Can Use D-Bus to Achieve Interoperability

Thiago Macieira

September 23rd 2006

Definitions

What do I mean by interoperability?

Interoperability is the ability for one software to correctly work with another software

- By definition, requires two or more softwares
- A method of exchange of information must be defined
- Preferrably, such protocol should be open and public

Definition according to Merriam-Webster

Main Entry:
in.ter.op.er.a.bil.i.ty
Function: noun
: ability of a system (as a weapons system) to work with or use the parts or equipment of another system

This presentation focuses on inter-process interoperability

Definitions

What is D-Bus?

- D-Bus is an Inter-Process Comunication / Remote Procedure Calling (IPC/RPC) system defined by *freedesktop.org*
- Definition according to its homepage:
  "D-Bus is a message bus system, a simple way for applications to talk to one another"
- D-Bus provides a central daemon that connects multiple applications together, in a star formation
- Applications can also connect to each other directly, in peer-to-peer mode.
- D-Bus' only **required** dependency is an XML parser and includes many language bindings

Get on D-Bus today!

Interoperability: benefits

Why should I bother with it?

If you have an application or technology, interoperability provides you:

- Another method of access to your technology
- A broader user-base, adding other developers as users
- Less resource usage on your system, by allowing other applications to access it, instead of duplicating it
- Good PR, because you'll be a nice guy

Interoperability: benefits

I'm just starting an application, I have nothing to offer

If you don't have distinctive features in your application to offer, interoperability provides you:

- Ability to access other applications' features
- Reduce the time required for your application to be released
- Reduce your workload
- Benefit from the improvements in those applications almost immediately

Interoperability: example

Can you give an example?

The Desktop API

- DAPI allows access to desktop functionality like sending email, turning screensavers on/off, opening urls, etc.
- DAPI consists of two components: A daemon, and a library that links to the application
- The library handles the IPC (socket) communication with the daemon and implements the API

The `xdg-utils`

`xdg-utils` is a software package that provides command-line tools to:

- (un)install desktop menu items
- (un)install icons
- query MIME type information
- open URLs
- launching the email composer
- etc.

Implementing interoperability

What do I have to do to interoperate?

Assuming the need for interoperability has been established, you still need to:

- define an API to regulate the exchange of information
- define format of the information exchange

The definition of the API is out of the scope of this presentation. But the selection of a format for the exchange of information can lead to many solutions.

Implementing interoperability

First solution: write a library

The first solution is to write a library that other applications can link to and access the API through normal function calls

Advantages

- Fast
- Little modification required to existing code

Disadvantages

- If you write in C++, only C++ can use it
- If you use KDE and Qt classes, the client code has to use it too
- Even if you write in pure C (which is hard for a C++ developer), only C and C++ can access it:
  other languages, like Java, Python or Perl, still need some binding or glue code

Implementing interoperability

Second solution: write an external application

Like `xdg-utils`, we remove the language choice problem by providing an external application.

Advantages

- No problem with linking
- Relatively easy access from any language
- Easy access from shell scripts

Disadvantages

- Requires re-parsing of data
- Incurs a performance penalty of starting a new process every time:
  - fork
  - exec
  - load libraries
  - process dynamic relocations

Implementing interoperability

Third solution: socket, pipe or another raw IPC system

Using a raw IPC system like DAPI drops the requirement to launch an application at every turn:

Advantages

- No problem with linking
- No penalty associated with process starting
- Relatively easy access from any language

Disadvantages

- Requires establishing and testing the new protocol
- Requires the protocol to be established in each participant
- Usually difficult to add improvements to
- Difficult access from shell scripts

Implementing interoperability

Fourth solution: other IPC/RPC systems

Instead of defining one's own IPC/RPC system, the use of an existing solution like DCOP allows the developer to focus on the API itself, not on the details of how data is actually transmitted and parsed.

Advantages

- No problem with linking
- No penalty associated with process starting
- Potentially easy access from any language
- With a suitable command-line accessor tool, easy access from shell scripts

Disadvantages

- Access from any language is limited by availability of implementations
- DCOP suffers from forward compatibility problems and is hard to extend

Interoperability with D-Bus

Final solution: D-Bus

D-Bus has been modelled after DCOP and so has all of its benefits. In addition:

- It has been designed with forward compatibility with future extensions in mind
- Already possesses bindings to many other frameworks: glib, Mono, Python, Perl, Java, ...
- It works in all platforms KDE supports (work on Windows is on-going)
- Best of all: it is natively supported by Qt and KDE
  - it's the IPC/RPC system that we chose to replace DCOP
  - it's been designed to replace DCOP, so its concepts are very similar
  - it integrates nicely with the Qt Meta Object and Meta Type Systems

D-Bus in KDE

I'm convinced, how can I use it in my KDE application?

The best of all is that KDE already has everything you need to use D-Bus and also does some of the initialisation for you:

- The KDE buildsystem can parse the XML definition files for you and output C++ code you can use
- KDE also connects your application to D-Bus
  if you're using KUniqueApplication, you even get a nice name
- You just need to define what you want to export or what you want to access

D-Bus in KDE

What do I need to do to call a remote service?

To call remote functions, you need to identify the interface or interfaces that provide the functionality you need. By telling CMake which interfaces those are, it generates C++ code for you.

Example

Snippet from `kcookiejar/main.cpp`

```
org::kde::KCookieServer *kcookiejar = new org::kde::KCookieServer("org.kde.kded"
      , "/modules/kcookiejar", QDBusConnection::sessionBus());
if (args->isSet("remove-all")) {
    kcookiejar->deleteAllCookies();
} else {
    QDBusInterface("org.kded.kded", "/kded", "org.kde.kded").call("loadModule",
          QByteArray("kcookiejar"));
}
```

D-Bus in KDE

What do I need to export a service?

If you want to export some functionality from your application, you have the following options:

- Export an existing `QObject`-derived class, allowing any slot to be called on it
- Create a class derived from `QDBusAbstractAdaptor` that will be the interface between your code and the rest of the world
- Process the XML definition of an interface into a C++ class derived from `QDBusAbstractAdaptor`

With the latter alternative, you can easily implement an interface defined by third-parties.

Questions ?

Thiago Macieira
thiago@kde.org