D-Bus Tutorial

Havoc Pennington

Red Hat, Inc.

<hp@pobox.com>

David Wheeler

John Palmieri

Red Hat, Inc.

<johnp@redhat.com>

Colin Walters

Red Hat, Inc.

<walters@redhat.com>

Version 0.5.0

Table of Contents

Tutorial Work In Progress

This tutorial is not complete; it probably contains some useful information, but also has plenty of gaps. Right now, you'll also need to refer to the D-Bus specification, Doxygen reference documentation, and look at some examples of how other apps use D-Bus.

Enhancing the tutorial is definitely encouraged - send your patches or suggestions to the mailing list. If you create a D-Bus binding, please add a section to the tutorial for your binding, if only a short section with a couple of examples.

What is D-Bus?

D-Bus is a system for *interprocess communication* (IPC). Architecturally, it has several layers:

- A library, *libdbus*, that allows two applications to connect to each other and exchange messages.
- A *message bus daemon* executable, built on libdbus, that multiple applications can connect to. The daemon can route messages from one application to zero or more other applications.
- *Wrapper libraries* or *bindings* based on particular application frameworks. For example, libdbus-glib and libdbus-qt. There are also bindings to languages such as Python. These wrapper libraries are the API most people should use, as they simplify the details of D-Bus programming. libdbus is intended to be a low-level backend for the higher level bindings. Much of the libdbus API is only useful for binding implementation.

libdbus only supports one-to-one connections, just like a raw network socket. However, rather than sending byte streams over the connection, you send *messages*. Messages have a header identifying the kind of message, and a body containing a data payload. libdbus also abstracts the exact transport used (sockets vs. whatever else), and handles details such as authentication.

The message bus daemon forms the hub of a wheel. Each spoke of the wheel is a one-to-one connection to

an application using libdbus. An application sends a message to the bus daemon over its spoke, and the bus daemon forwards the message to other connected applications as appropriate. Think of the daemon as a router.

The bus daemon has multiple instances on a typical computer. The first instance is a machine-global singleton, that is, a system daemon similar to sendmail or Apache. This instance has heavy security restrictions on what messages it will accept, and is used for systemwide communication. The other instances are created one per user login session. These instances allow applications in the user's session to communicate with one another.

The systemwide and per-user daemons are separate. Normal within-session IPC does not involve the systemwide message bus process and vice versa.

D-Bus applications

There are many, many technologies in the world that have "Inter-process communication" or "networking" in their stated purpose: CORBA, DCE, DCOM, DCOP, XML-RPC, SOAP, MBUS, Internet Communications Engine (ICE), and probably hundreds more. Each of these is tailored for particular kinds of application. D-Bus is designed for two specific cases:

- Communication between desktop applications in the same desktop session; to allow integration of the desktop session as a whole, and address issues of process lifecycle (when do desktop components start and stop running).
- Communication between the desktop session and the operating system, where the operating system would typically include the kernel and any system daemons or processes.

For the within-desktop-session use case, the GNOME and KDE desktops have significant previous experience with different IPC solutions such as CORBA and DCOP. D-Bus is built on that experience and carefully tailored to meet the needs of these desktop projects in particular. D-Bus may or may not be appropriate for other applications; the FAQ has some comparisons to other IPC systems.

The problem solved by the systemwide or communication-with-the-OS case is explained well by the following text from the Linux Hotplug project:

> A gap in current Linux support is that policies with any sort of dynamic "interact with user" component aren't currently supported. For example, that's often needed the first time a network adapter or printer is connected, and to determine appropriate places to mount disk drives. It would seem that such actions could be supported for any case where a responsible human can be identified: single user workstations, or any system which is remotely administered.
>
> This is a classic "remote sysadmin" problem, where in this case hotplugging needs to deliver an event from one security domain (operating system kernel, in this case) to another (desktop for logged-in user, or remote sysadmin). Any effective response must go the other way: the remote domain taking some action that lets the kernel expose the desired device capabilities. (The action can often be taken asynchronously, for example letting new hardware be idle until a meeting finishes.) At this writing, Linux doesn't have widely adopted solutions to such problems. However, the new D-Bus work may begin to solve that problem.

D-Bus may happen to be useful for purposes other than the one it was designed for. Its general properties that distinguish it from other forms of IPC are:

- Binary protocol designed to be used asynchronously (similar in spirit to the X Window System protocol).
- Stateful, reliable connections held open over time.
- The message bus is a daemon, not a "swarm" or distributed architecture.
- Many implementation and deployment issues are specified rather than left ambiguous/configurable/pluggable.
- Semantics are similar to the existing DCOP system, allowing KDE to adopt it more easily.
- Security features to support the systemwide mode of the message bus.

Concepts

Some basic concepts apply no matter what application framework you're using to write a D-Bus application. The exact code you write will be different for GLib vs. Qt vs. Python applications, however.

Here is a diagram (png svg) that may help you visualize the concepts that follow.

Native Objects and Object Paths

Your programming framework probably defines what an "object" is like; usually with a base class. For example: java.lang.Object, GObject, QObject, python's base Object, or whatever. Let's call this a *native object*.

The low-level D-Bus protocol, and corresponding libdbus API, does not care about native objects. However, it provides a concept called an *object path*. The idea of an object path is that higher-level bindings can name native object instances, and allow remote applications to refer to them.

The object path looks like a filesystem path, for example an object could be named `/org/kde/kspread/sheets/3/cells/4/5`. Human-readable paths are nice, but you are free to create an object named `/com/mycompany/c5yo817y0c1y1c5b` if it makes sense for your application.

Namespacing object paths is smart, by starting them with the components of a domain name you own (e.g. `/org/kde`). This keeps different code modules in the same process from stepping on one another's toes.

Methods and Signals

Each object has *members*; the two kinds of member are *methods* and *signals*. Methods are operations that can be invoked on an object, with optional input (aka arguments or "in parameters") and output (aka return values or "out parameters"). Signals are broadcasts from the object to any interested observers of the object; signals may contain a data payload.

Both methods and signals are referred to by name, such as "Frobate" or "OnClicked".

Interfaces

Each object supports one or more *interfaces*. Think of an interface as a named group of methods and signals, just as it is in GLib or Qt or Java. Interfaces define the *type* of an object instance.

DBus identifies interfaces with a simple namespaced string, something like `org.freedesktop.Introspectable`. Most bindings will map these interface names directly to the appropriate programming language construct, for example to Java interfaces or C++ pure virtual classes.

Proxies

A *proxy object* is a convenient native object created to represent a remote object in another process. The low-level DBus API involves manually creating a method call message, sending it, then manually receiving and processing the method reply message. Higher-level bindings provide proxies as an alternative. Proxies look like a normal native object; but when you invoke a method on the proxy object, the binding converts it into a DBus method call message, waits for the reply message, unpacks the return value, and returns it from the native method..

In pseudocode, programming without proxies might look like this:

```
Message message = new Message("/remote/object/path", "MethodName", arg1, arg2);
Connection connection = getBusConnection();
connection.send(message);
Message reply = connection.waitForReply(message);
if (reply.isError()) {

} else {
   Object returnValue = reply.getReturnValue();
}
```

Programming with proxies might look like this:

```
Proxy proxy = new Proxy(getBusConnection(), "/remote/object/path");
Object returnValue = proxy.MethodName(arg1, arg2);
```

Bus Names

When each application connects to the bus daemon, the daemon immediately assigns it a name, called the *unique connection name*. A unique name begins with a ':' (colon) character. These names are never reused during the lifetime of the bus daemon - that is, you know a given name will always refer to the same application. An example of a unique name might be `:34-907`. The numbers after the colon have no meaning other than their uniqueness.

When a name is mapped to a particular application's connection, that application is said to *own* that name.

Applications may ask to own additional *well-known names*. For example, you could write a specification to define a name called `com.mycompany.TextEditor`. Your definition could specify that to own this name, an

application should have an object at the path `/com/mycompany/TextFileManager` supporting the interface `org.freedesktop.FileHandler`.

Applications could then send messages to this bus name, object, and interface to execute method calls.

You could think of the unique names as IP addresses, and the well-known names as domain names. So `com.mycompany.TextEditor` might map to something like `:34-907` just as `mycompany.com` maps to something like `192.168.0.5`.

Names have a second important use, other than routing messages. They are used to track lifecycle. When an application exits (or crashes), its connection to the message bus will be closed by the operating system kernel. The message bus then sends out notification messages telling remaining applications that the application's names have lost their owner. By tracking these notifications, your application can reliably monitor the lifetime of other applications.

Bus names can also be used to coordinate single-instance applications. If you want to be sure only one `com.mycompany.TextEditor` application is running for example, have the text editor application exit if the bus name already has an owner.

Addresses

Applications using D-Bus are either servers or clients. A server listens for incoming connections; a client connects to a server. Once the connection is established, it is a symmetric flow of messages; the client-server distinction only matters when setting up the connection.

If you're using the bus daemon, as you probably are, your application will be a client of the bus daemon. That is, the bus daemon listens for connections and your application initiates a connection to the bus daemon.

A D-Bus *address* specifies where a server will listen, and where a client will connect. For example, the address `unix:path=/tmp/abcdef` specifies that the server will listen on a UNIX domain socket at the path `/tmp/abcdef` and the client will connect to that socket. An address can also specify TCP/IP sockets, or any other transport defined in future iterations of the D-Bus specification.

When using D-Bus with a message bus daemon, libdbus automatically discovers the address of the per-session bus daemon by reading an environment variable. It discovers the systemwide bus daemon by checking a well-known UNIX domain socket path (though you can override this address with an environment variable).

If you're using D-Bus without a bus daemon, it's up to you to define which application will be the server and which will be the client, and specify a mechanism for them to agree on the server's address. This is an unusual case.

Big Conceptual Picture

Pulling all these concepts together, to specify a particular method call on a particular object instance, a number of nested components have to be named:

```
Address -> [Bus Name] -> Path -> Interface -> Method
```

The bus name is in brackets to indicate that it's optional -- you only provide a name to route the method call to the right application when using the bus daemon. If you have a direct connection to another application, bus names aren't used; there's no bus daemon.

The interface is also optional, primarily for historical reasons; DCOP does not require specifying the interface, instead simply forbidding duplicate method names on the same object instance. D-Bus will thus let you omit the interface, but if your method name is ambiguous it is undefined which method will be invoked.

Messages - Behind the Scenes

D-Bus works by sending messages between processes. If you're using a sufficiently high-level binding, you may never work with messages directly.

There are 4 message types:

- Method call messages ask to invoke a method on an object.
- Method return messages return the results of invoking a method.
- Error messages return an exception caused by invoking a method.
- Signal messages are notifications that a given signal has been emitted (that an event has occurred). You could also think of these as "event" messages.

A method call maps very simply to messages: you send a method call message, and receive either a method return message or an error message in reply.

Each message has a *header*, including *fields*, and a *body*, including *arguments*. You can think of the header as the routing information for the message, and the body as the payload. Header fields might include the sender bus name, destination bus name, method or signal name, and so forth. One of the header fields is a *type signature* describing the values found in the body. For example, the letter "i" means "32-bit integer" so the signature "ii" means the payload has two 32-bit integers.

Calling a Method - Behind the Scenes

A method call in DBus consists of two messages; a method call message sent from process A to process B, and a matching method reply message sent from process B to process A. Both the call and the reply messages are routed through the bus daemon. The caller includes a different serial number in each call message, and the reply message includes this number to allow the caller to match replies to calls.

The call message will contain any arguments to the method. The reply message may indicate an error, or may contain data returned by the method.

A method invocation in DBus happens as follows:

- The language binding may provide a proxy, such that invoking a method on an in-process object invokes a method on a remote object in another process. If so, the application calls a method on the proxy, and the proxy constructs a method call message to send to the remote process.

- For more low-level APIs, the application may construct a method call message itself, without using a proxy.
- In either case, the method call message contains: a bus name belonging to the remote process; the name of the method; the arguments to the method; an object path inside the remote process; and optionally the name of the interface that specifies the method.
- The method call message is sent to the bus daemon.
- The bus daemon looks at the destination bus name. If a process owns that name, the bus daemon forwards the method call to that process. Otherwise, the bus daemon creates an error message and sends it back as the reply to the method call message.
- The receiving process unpacks the method call message. In a simple low-level API situation, it may immediately run the method and send a method reply message to the bus daemon. When using a high-level binding API, the binding might examine the object path, interface, and method name, and convert the method call message into an invocation of a method on a native object (GObject, java.lang.Object, QObject, etc.), then convert the return value from the native method into a method reply message.
- The bus daemon receives the method reply message and sends it to the process that made the method call.
- The process that made the method call looks at the method reply and makes use of any return values included in the reply. The reply may also indicate that an error occurred. When using a binding, the method reply message may be converted into the return value of of a proxy method, or into an exception.

The bus daemon never reorders messages. That is, if you send two method call messages to the same recipient, they will be received in the order they were sent. The recipient is not required to reply to the calls in order, however; for example, it may process each method call in a separate thread, and return reply messages in an undefined order depending on when the threads complete. Method calls have a unique serial number used by the method caller to match reply messages to call messages.

Emitting a Signal - Behind the Scenes

A signal in DBus consists of a single message, sent by one process to any number of other processes. That is, a signal is a unidirectional broadcast. The signal may contain arguments (a data payload), but because it is a broadcast, it never has a "return value." Contrast this with a method call (see the section called “Calling a Method - Behind the Scenes”) where the method call message has a matching method reply message.

The emitter (aka sender) of a signal has no knowledge of the signal recipients. Recipients register with the bus daemon to receive signals based on "match rules" - these rules would typically include the sender and the signal name. The bus daemon sends each signal only to recipients who have expressed interest in that signal.

A signal in DBus happens as follows:

- A signal message is created and sent to the bus daemon. When using the low-level API this may be done manually, with certain bindings it may be done for you by the binding when a native object emits a native signal or event.

- The signal message contains the name of the interface that specifies the signal; the name of the signal; the bus name of the process sending the signal; and any arguments
- Any process on the message bus can register "match rules" indicating which signals it is interested in. The bus has a list of registered match rules.
- The bus daemon examines the signal and determines which processes are interested in it. It sends the signal message to these processes.
- Each process receiving the signal decides what to do with it; if using a binding, the binding may choose to emit a native signal on a proxy object. If using the low-level API, the process may just look at the signal sender and name and decide what to do based on that.

Introspection

D-Bus objects may support the interface `org.freedesktop.DBus.Introspectable`. This interface has one method `Introspect` which takes no arguments and returns an XML string. The XML string describes the interfaces, methods, and signals of the object. See the D-Bus specification for more details on this introspection format.

GLib API: Using Remote Objects

The GLib binding is defined in the header file `<dbus/dbus-glib.h>`.

D-Bus - GLib type mappings

The heart of the GLib bindings for D-Bus is the mapping it provides between D-Bus "type signatures" and GLib types (`GType`). The D-Bus type system is composed of a number of "basic" types, along with several "container" types.

Basic type mappings

Below is a list of the basic types, along with their associated mapping to a `GType`.

D-Bus basic type	GType	Free function	Notes
BYTE	G_TYPE_UCHAR		
BOOLEAN	G_TYPE_BOOLEAN		
INT16	G_TYPE_INT		Will be changed to a G_TYPE_INT16 once GLib has it
UINT16	G_TYPE_UINT		Will be changed to a G_TYPE_UINT16 once GLib has it
INT32	G_TYPE_INT		Will be changed to a G_TYPE_INT32 once GLib has it
UINT32	G_TYPE_UINT		Will be changed to a G_TYPE_UINT32 once GLib has it
INT64	G_TYPE_GINT64		
UINT64	G_TYPE_GUINT64		

DOUBLE	G_TYPE_DOUBLE		
STRING	G_TYPE_STRING	g_free	
OBJECT_PATH	DBUS_TYPE_G_PROXY	g_object_unref	The returned proxy does not have an interface set; use dbus_g_proxy_set_interface to invoke methods

As you can see, the basic mapping is fairly straightforward.

Container type mappings

The D-Bus type system also has a number of "container" types, such as DBUS_TYPE_ARRAY and DBUS_TYPE_STRUCT. The D-Bus type system is fully recursive, so one can for example have an array of array of strings (i.e. type signature aas).

However, not all of these types are in common use; for example, at the time of this writing the author knows of no one using DBUS_TYPE_STRUCT, or a DBUS_TYPE_ARRAY containing any non-basic type. The approach the GLib bindings take is pragmatic; try to map the most common types in the most obvious way, and let using less common and more complex types be less "natural".

First, D-Bus type signatures which have an "obvious" corresponding built-in GLib type are mapped using that type:

D-Bus type signature	Description	GType	C typedef	Free function	Notes
as	Array of strings	G_TYPE_STRV	char **	g_strfreev	
v	Generic value container	G_TYPE_VALUE	GValue *	g_value_unset	The calling conventions for values expect that method callers have allocated return values; see below.

The next most common recursive type signatures are arrays of basic values. The most obvious mapping for arrays of basic types is a GArray. Now, GLib does not provide a builtin GType for GArray. However, we actually need more than that - we need a "parameterized" type which includes the contained type. Why we need this we will see below.

The approach taken is to create these types in the D-Bus GLib bindings; however, there is nothing D-Bus specific about them. In the future, we hope to include such "fundamental" types in GLib itself.

D-Bus type signature	Description	GType	C typedef	Free function	Notes
ay	Array of bytes	DBUS_TYPE_G_BYTE_ARRAY	GArray *	g_array_free	
au	Array of uint	DBUS_TYPE_G_UINT_ARRAY	GArray *	g_array_free	
ai	Array of int	DBUS_TYPE_G_INT_ARRAY	GArray *	g_array_free	
ax	Array of int64	DBUS_TYPE_G_INT64_ARRAY	GArray *	g_array_free	
at	Array of uint64	DBUS_TYPE_G_UINT64_ARRAY	GArray *	g_array_free	

ad	Array of double	DBUS_TYPE_G_DOUBLE_ARRAY	GArray *	g_array_free	
ab	Array of boolean	DBUS_TYPE_G_BOOLEAN_ARRAY	GArray *	g_array_free	

D-Bus also includes a special type DBUS_TYPE_DICT_ENTRY which is only valid in arrays. It's intended to be mapped to a "dictionary" type by bindings. The obvious GLib mapping here is GHashTable. Again, however, there is no builtin `GType` for a GHashTable. Moreover, just like for arrays, we need a parameterized type so that the bindings can communiate which types are contained in the hash table.

At present, only strings are supported. Work is in progress to include more types.

D-Bus type signature	Description	GType	C typedef	Free function	Notes
a{ss}	Dictionary mapping strings to strings	DBUS_TYPE_G_STRING_STRING_HASHTABLE	GHashTable *	g_hash_table_destroy	

Arbitrarily recursive type mappings

Finally, it is possible users will want to write or invoke D-Bus methods which have arbitrarily complex type signatures not directly supported by these bindings. For this case, we have a `DBusGValue` which acts as a kind of special variant value which may be iterated over manually. The `GType` associated is `DBUS_TYPE_G_VALUE`.

TODO insert usage of `DBUS_TYPE_G_VALUE` here.

A sample program

Here is a D-Bus program using the GLib bindings.

```
int
main (int argc, char **argv)
{
  DBusGConnection *connection;
  GError *error;
  DBusGProxy *proxy;
  char **name_list;
  char **name_list_ptr;

  g_type_init ();

  error = NULL;
  connection = dbus_g_bus_get (DBUS_BUS_SESSION,
                               &error);
  if (connection == NULL)
    {
      g_printerr ("Failed to open connection to bus: %s\n",
```

```
                  error->message);
      g_error_free (error);
      exit (1);
    }

  /* Create a proxy object for the "bus driver" (name "org.freedesktop.DBus") */

  proxy = dbus_g_proxy_new_for_name (connection,
                                     DBUS_SERVICE_DBUS,
                                     DBUS_PATH_DBUS,
                                     DBUS_INTERFACE_DBUS);

  /* Call ListNames method, wait for reply */
  error = NULL;
  if (!dbus_g_proxy_call (proxy, "ListNames", &error, G_TYPE_INVALID,
                          G_TYPE_STRV, &name_list, G_TYPE_INVALID))
    {
      /* Just do demonstrate remote exceptions versus regular GError */
      if (error->domain == DBUS_GERROR && error->code == DBUS_GERROR_REMOTE_EXCEPTION)
        g_printerr ("Caught remote method exception %s: %s",
                    dbus_g_error_get_name (error),
                    error->message);
      else
        g_printerr ("Error: %s\n", error->message);
      g_error_free (error);
      exit (1);
    }

  /* Print the results */

  g_print ("Names on the message bus:\n");

  for (name_list_ptr = name_list; *name_list_ptr; name_list_ptr++)
    {
      g_print ("  %s\n", *name_list_ptr);
    }
  g_strfreev (name_list);

  g_object_unref (proxy);

  return 0;
}
```

Program initalization

A connection to the bus is acquired using `dbus_g_bus_get`. Next, a proxy is created for the object "/org/freedesktop/DBus" with interface `org.freedesktop.DBus` on the service `org.freedesktop.DBus`. This is a proxy for the message bus itself.

Understanding method invocation

You have a number of choices for method invocation. First, as used above, `dbus_g_proxy_call` sends a method call to the remote object, and blocks until a reply is recieved. The outgoing arguments are specified in the varargs array, terminated with `G_TYPE_INVALID`. Next, pointers to return values are specified, followed again by `G_TYPE_INVALID`.

To invoke a method asynchronously, use `dbus_g_proxy_begin_call`. This returns a `DBusGPendingCall` object; you may then set a notification function using `dbus_g_pending_call_set_notify`.

Connecting to object signals

You may connect to signals using `dbus_g_proxy_add_signal` and `dbus_g_proxy_connect_signal`. You must invoke `dbus_g_proxy_add_signal` to specify the signature of your signal handlers; you may then invoke `dbus_g_proxy_connect_signal` multiple times.

Note that it will often be the case that there is no builtin marshaller for the type signature of a remote signal. In that case, you must generate a marshaller yourself by using glib-genmarshal, and then register it using `dbus_g_object_register_marshaller`.

Error handling and remote exceptions

All of the GLib binding methods such as `dbus_g_proxy_end_call` return a `GError`. This `GError` can represent two different things:

- An internal D-Bus error, such as an out-of-memory condition, an I/O error, or a network timeout. Errors generated by the D-Bus library itself have the domain `DBUS_GERROR`, and a corresponding code such as `DBUS_GERROR_NO_MEMORY`. It will not be typical for applications to handle these errors specifically.

- A remote D-Bus exception, thrown by the peer, bus, or service. D-Bus remote exceptions have both a textual "name" and a "message". The GLib bindings store this information in the `GError`, but some special rules apply.

  The set error will have the domain `DBUS_GERROR` as above, and will also have the code `DBUS_GERROR_REMOTE_EXCEPTION`. In order to access the remote exception name, you must use a special accessor, such as `dbus_g_error_has_name` or `dbus_g_error_get_name`. The remote exception detailed message is accessible via the regular GError `message` member.

More examples of method invocation

Sending an integer and string, receiving an array of bytes

```
GArray *arr;

error = NULL;
if (!dbus_g_proxy_call (proxy, "Foobar", &error,
                        G_TYPE_INT, 42, G_TYPE_STRING, "hello",
                        G_TYPE_INVALID,
                        DBUS_TYPE_G_UCHAR_ARRAY, &arr, G_TYPE_INVALID))
```

```
    {
      /* Handle error */
    }
   g_assert (arr != NULL);
   printf ("got back %u values", arr->len);
```

Sending a GHashTable

```
GHashTable *hash = g_hash_table_new (g_str_hash, g_str_equal);
guint32 ret;

g_hash_table_insert (hash, "foo", "bar");
g_hash_table_insert (hash, "baz", "whee");

error = NULL;
if (!dbus_g_proxy_call (proxy, "HashSize", &error,
                        DBUS_TYPE_G_STRING_STRING_HASH, hash, G_TYPE_INVALID,
                        G_TYPE_UINT, &ret, G_TYPE_INVALID))
  {
    /* Handle error */
  }
g_assert (ret == 2);
g_hash_table_destroy (hash);
```

Receiving a boolean and a string

```
gboolean boolret;
char *strret;

error = NULL;
if (!dbus_g_proxy_call (proxy, "GetStuff", &error,
                        G_TYPE_INVALID,
                        G_TYPE_BOOLEAN, &boolret,
                        G_TYPE_STRING, &strret,
                        G_TYPE_INVALID))
  {
    /* Handle error */
  }
printf ("%s %s", boolret ? "TRUE" : "FALSE", strret);
g_free (strret);
```

Sending two arrays of strings

```
/* NULL terminate */
char *strs_static[] = {"foo", "bar", "baz", NULL};
/* Take pointer to array; cannot pass array directly */
char **strs_static_p = strs_static;
char **strs_dynamic;

strs_dynamic = g_new (char *, 4);
strs_dynamic[0] = g_strdup ("hello");
```

```
  strs_dynamic[1] = g_strdup ("world");
  strs_dynamic[2] = g_strdup ("!");
  /* NULL terminate */
  strs_dynamic[3] = NULL;

  error = NULL;
  if (!dbus_g_proxy_call (proxy, "TwoStrArrays", &error,
                          G_TYPE_STRV, strs_static_p,
                          G_TYPE_STRV, strs_dynamic,
                          G_TYPE_INVALID,
                          G_TYPE_INVALID))
    {
      /* Handle error */
    }
   g_strfreev (strs_dynamic);
```

Sending a boolean, receiving an array of strings

```
  char **strs;
  char **strs_p;
  gboolean blah;

  error = NULL;
  blah = TRUE;
  if (!dbus_g_proxy_call (proxy, "GetStrs", &error,
                          G_TYPE_BOOLEAN, blah,
                          G_TYPE_INVALID,
                          G_TYPE_STRV, &strs,
                          G_TYPE_INVALID))
    {
      /* Handle error */
    }
   for (strs_p = strs; *strs_p; strs_p++)
     printf ("got string: \"%s\"", *strs_p);
   g_strfreev (strs);
```

Sending a variant

```
  GValue val = {0, };

  g_value_init (&val, G_TYPE_STRING);
  g_value_set_string (&val, "hello world");

  error = NULL;
  if (!dbus_g_proxy_call (proxy, "SendVariant", &error,
                          G_TYPE_VALUE, &val, G_TYPE_INVALID,
                          G_TYPE_INVALID))
    {
      /* Handle error */
    }
  g_assert (ret == 2);
```

```
  g_value_unset (&val);
```

Receiving a variant

```
  GValue val = {0, };

  error = NULL;
  if (!dbus_g_proxy_call (proxy, "GetVariant", &error, G_TYPE_INVALID,
                          G_TYPE_VALUE, &val, G_TYPE_INVALID))
    {
      /* Handle error */
    }
  if (G_VALUE_TYPE (&val) == G_TYPE_STRING)
    printf ("%s\n", g_value_get_string (&val));
  else if (G_VALUE_TYPE (&val) == G_TYPE_INT)
    printf ("%d\n", g_value_get_int (&val));
  else
    ...
  g_value_unset (&val);
```

Generated Bindings

By using the Introspection XML files, convenient client-side bindings can be automatically created to ease the use of a remote DBus object.

Here is a sample XML file which describes an object that exposes one method, named `ManyArgs`.

```
<?xml version="1.0" encoding="UTF-8" ?>
<node name="/com/example/MyObject">
  <interface name="com.example.MyObject">
    <method name="ManyArgs">
      <arg type="u" name="x" direction="in" />
      <arg type="s" name="str" direction="in" />
      <arg type="d" name="trouble" direction="in" />
      <arg type="d" name="d_ret" direction="out" />
      <arg type="s" name="str_ret" direction="out" />
    </method>
  </interface>
</node>
```

Run `dbus-binding-tool --mode=glib-client` *`FILENAME`* `>` *`HEADER_NAME`* to generate the header file. For example: **dbus-binding-tool --mode=glib-client my-object.xml > my-object-bindings.h**. This will generate inline functions with the following prototypes:

```
/* This is a blocking call */
gboolean
com_example_MyObject_many_args (DBusGProxy *proxy, const guint IN_x,
                                const char * IN_str, const gdouble IN_trouble,
                                gdouble* OUT_d_ret, char ** OUT_str_ret,
                                GError **error);
```

```
/* This is a non-blocking call */
DBusGProxyCall*
com_example_MyObject_many_args_async (DBusGProxy *proxy, const guint IN_x,
                                      const char * IN_str, const gdouble IN_trouble,
                                      com_example_MyObject_many_args_reply callback,
                                      gpointer userdata);

/* This is the typedef for the non-blocking callback */
typedef void
(*com_example_MyObject_many_args_reply)
(DBusGProxy *proxy, gdouble OUT_d_ret, char * OUT_str_ret,
 GError *error, gpointer userdata);
```

The first argument in all functions is a `DBusGProxy *`, which you should create with the usual `dbus_g_proxy_new_*` functions. Following that are the "in" arguments, and then either the "out" arguments and a `GError *` for the synchronous (blocking) function, or callback and user data arguments for the asynchronous (non-blocking) function. The callback in the asynchronous function passes the `DBusGProxy *`, the returned "out" arguments, an `GError *` which is set if there was an error otherwise `NULL`, and the user data.

As with the server-side bindings support (see the section called “GLib API: Implementing Objects”), the exact behaviour of the client-side bindings can be manipulated using "annotations". Currently the only annotation used by the client bindings is `org.freedesktop.DBus.GLib.NoReply`, which sets the flag indicating that the client isn't expecting a reply to the method call, so a reply shouldn't be sent. This is often used to speed up rapid method calls where there are no "out" arguments, and not knowing if the method succeeded is an acceptable compromise to half the traffic on the bus.

GLib API: Implementing Objects

At the moment, to expose a GObject via D-Bus, you must write XML by hand which describes the methods exported by the object. In the future, this manual step will be obviated by the upcoming GLib introspection support.

Here is a sample XML file which describes an object that exposes one method, named `ManyArgs`.

```
<?xml version="1.0" encoding="UTF-8" ?>

<node name="/com/example/MyObject">

  <interface name="com.example.MyObject">
    <annotation name="org.freedesktop.DBus.GLib.CSymbol" value="my_object"/>
    <method name="ManyArgs">
      <!-- This is optional, and in this case is redunundant -->
      <annotation name="org.freedesktop.DBus.GLib.CSymbol" value="my_object_many_args"/>
      <arg type="u" name="x" direction="in" />
      <arg type="s" name="str" direction="in" />
      <arg type="d" name="trouble" direction="in" />
      <arg type="d" name="d_ret" direction="out" />
      <arg type="s" name="str_ret" direction="out" />
    </method>
```

```
  </interface>
</node>
```

This XML is in the same format as the D-Bus introspection XML format. Except we must include an "annotation" which give the C symbols corresponding to the object implementation prefix (`my_object`). In addition, if particular methods symbol names deviate from C convention (i.e. `ManyArgs` -> `many_args`), you may specify an annotation giving the C symbol.

Once you have written this XML, run `dbus-binding-tool --mode=glib-server` *`FILENAME`* `>` *`HEADER_NAME`*. to generate a header file. For example: **dbus-binding-tool --mode=glib-server my-object.xml > my-object-glue.h**.

Next, include the generated header in your program, and invoke `dbus_g_object_class_install_info` in the class initializer, passing the object class and "object info" included in the header. For example:

```
dbus_g_object_type_install_info (COM_FOO_TYPE_MY_OBJECT, &com_foo_my_object_info);
```

This should be done exactly once per object class.

To actually implement the method, just define a C function named e.g. `my_object_many_args` in the same file as the info header is included. At the moment, it is required that this function conform to the following rules:

- The function must return a value of type `gboolean`; `TRUE` on success, and `FALSE` otherwise.
- The first parameter is a pointer to an instance of the object.
- Following the object instance pointer are the method input values.
- Following the input values are pointers to return values.
- The final parameter must be a `GError **`. If the function returns `FALSE` for an error, the error parameter must be initalized with `g_set_error`.

Finally, you can export an object using `dbus_g_connection_register_g_object`. For example:

```
dbus_g_connection_register_g_object (connection,
                                     "/com/foo/MyObject",
                                     obj);
```

Server-side Annotations

There are several annotations that are used when generating the server-side bindings. The most common annotation is `org.freedesktop.DBus.GLib.CSymbol` but there are other annotations which are often useful.

`org.freedesktop.DBus.GLib.CSymbol`

> This annotation is used to specify the C symbol names for the various types (interface, method, etc), if it differs from the name DBus generates.

`org.freedesktop.DBus.GLib.Async`

This annotation marks the method implementation as an asynchronous function, which doesn't return a response straight away but will send the response at some later point to complete the call. This is used to implement non-blocking services where method calls can take time.

When a method is asynchronous, the function prototype is different. It is required that the function conform to the following rules:

- The function must return a value of type `gboolean`; `TRUE` on success, and `FALSE` otherwise. TODO: the return value is currently ignored.
- The first parameter is a pointer to an instance of the object.
- Following the object instance pointer are the method input values.
- The final parameter must be a `DBusGMethodInvocation *`. This is used when sending the response message back to the client, by calling `dbus_g_method_return` or `dbus_g_method_return_error`.

`org.freedesktop.DBus.GLib.Const`

This attribute can only be applied to "out" `<arg>` nodes, and specifies that the parameter isn't being copied when returned. For example, this turns a 's' argument from a `char **` to a `const char **`, and results in the argument not being freed by DBus after the message is sent.

`org.freedesktop.DBus.GLib.ReturnVal`

This attribute can only be applied to "out" `<arg>` nodes, and alters the expected function signature. It currently can be set to two values: `""` or `"error"`. The argument marked with this attribute is not returned via a pointer argument, but by the function's return value. If the attribute's value is the empty string, the `GError *` argument is also omitted so there is no standard way to return an error value. This is very useful for interfacing with existing code, as it is possible to match existing APIs. If the attribute's value is `"error"`, then the final argument is a `GError *` as usual.

Some examples to demonstrate the usage. This introspection XML:

```
<method name="Increment">
  <arg type="u" name="x" />
  <arg type="u" direction="out" />
</method>
```

Expects the following function declaration:

```
gboolean
my_object_increment (MyObject *obj, gint32 x, gint32 *ret, GError **error);
```

This introspection XML:

```
<method name="IncrementRetval">
  <arg type="u" name="x" />
  <arg type="u" direction="out" >
    <annotation name="org.freedesktop.DBus.GLib.ReturnVal" value=""/>
  </arg>
</method>
```

Expects the following function declaration:

```
gint32
my_object_increment_retval (MyObject *obj, gint32 x)
```

This introspection XML:

```
<method name="IncrementRetvalError">
  <arg type="u" name="x" />
  <arg type="u" direction="out" >
    <annotation name="org.freedesktop.DBus.GLib.ReturnVal" value="error"/>
  </arg>
</method>
```

Expects the following function declaration:

```
gint32
my_object_increment_retval_error (MyObject *obj, gint32 x, GError **error)
```

Python API

The Python API, dbus-python, is now documented separately in the dbus-python tutorial (also available in doc/tutorial.txt, and doc/tutorial.html if built with python-docutils, in the dbus-python source distribution).

Qt API: Using Remote Objects

The Qt bindings are not yet documented.

Qt API: Implementing Objects

The Qt bindings are not yet documented.