# Maemo Diablo Asynchronous GLib D-Bus Training Material

February 9, 2009

# Contents

# Chapter 1

# Asynchronous GLib D-Bus

## 1.1 Asynchronicity in D-Bus clients

So far all the RPC method calls that we've implemented have been "fast" in that their execution does not depend on access to slow services or external resources. In real life however, it is quite likely that you won't be able to provide some service immediately, but will have to wait for some external service to complete before completing your own method call.

The GLib wrappers provide a version of doing method calls where the call will be launched (almost) immediately, and a callback will be executed when the method call will return (either with a return value, or an error).

Using the asynchronous wrappers is important when your program needs to update some kind of status, or be reactive to the user (via a GUI or other interface). Otherwise the program would block waiting for the RPC method to return, and won't be able to refresh the GUI or screen when required. An alternative solution would be to use separate threads which would run the synchronous methods, but synchronisation between threads will become an issue and debugging threaded programs is much harder than single threaded ones. Also, implementation of threads might be sub optimal in some environments. These are the reasons why we won't be covering the thread scenario here.

We will simulate slow running RPC methods by adding a delay into the server method implementations so that it will become clear why asynchronous RPC mechanisms are important. As signals by their nature are asynchronous as well, they don't add anything to our example this time. In order to simplify the code listings, we drop signal support from the asynchronous clients (the server still contains them and will emit them).

## 1.2 Slow test server

The only change on the server side is the addition of delays into each of the RPC methods (`setvalue1`, `setvalue2`, `getvalue1` and `getvalue2`). This delay is added to the start of each function as follows:

```
/* How many microseconds to delay between each client operation. */
#define SERVER_DELAY_USEC (5*1000000UL)
```

```
  /*... Listing cut for brevity ...*/

gboolean value_object_setvalue1(ValueObject* obj, gint valueIn,
                                                   GError** error) {

  dbg("Called (valueIn=%d)", valueIn);
  g_assert(obj != NULL);

  dbg("Delaying operation");
  g_usleep(SERVER_DELAY_USEC);

  /* Compare the current value against old one. If they're the same,
     we don't need to do anything (except return success). */
  if (obj->value1 != valueIn) {
```

Listing 1.1: Slowing down setvalue1 (glib-dbus-async/server.c)

Building the server is done as before, but we'll notice the delay when we call an RPC method:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > run-standalone.sh ./server &
server:main Connecting to the Session D-Bus.
server:main Registering the well-known name (org.maemo.Platdev_ex)
server:main RequestName returned 1.
server:main Creating one Value object.
server:value_object_class_init: Called
server:value_object_class_init: Creating signals
server:value_object_class_init: Binding to GLib/D-Bus
server:value_object_class_init: Done
server:value_object_init: Called
server:main Registering it on the D-Bus.
server:main Ready to serve requests (daemonizing).
server: Not daemonizing (built with NO_DAEMON-build define)
[sbox-DIABLO_X86: ~/glib-dbus-async] > time run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue1
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
method return sender=:1.54 -> dest=:1.56
 int32 0

real    0m5.066s
user    0m0.004s
sys     0m0.056s
```

Testing the delayed server version

Above we use the `time` shell built-in command. It will run the given command while measuring the wall clock time (aka real time) and time used while executing the code and system calls. In our case, we're only interested in the `real` time. The method call will delay for about 5 seconds as it should. The delay (even if given with microsecond resolution) is always approximate and longer than the requested amount. Exact delay will depend on many factors, most of which you cannot influence directly.

We'll next experiment with a likely scenario where another method call comes along while the first one is still being executed. This is best tested by just repeating the sending command twice, but running the first one on the background (so that the shell doesn't wait it to complete first). The server is still running on the background from the previous test:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > time run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue1 &
[2] 17010
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
[sbox-DIABLO_X86: ~/glib-dbus-async] > time run-standalone.sh dbus-send \
 --type=method_call --print-reply --dest=org.maemo.Platdev_ex \
 /GlobalValue org.maemo.Value.getvalue1
method return sender=:1.54 -> dest=:1.57
 int32 0

real    0m5.176s
user    0m0.008s
sys     0m0.092s
server:value_object_getvalue1: Called (internal value1 is 0)
server:value_object_getvalue1: Delaying operation
method return sender=:1.54 -> dest=:1.58
 int32 0

real    0m9.852s
user    0m0.004s
sys     0m0.052s
```

Delays accumulating

What we can see from the above output is that the first client is delayed for about 5 seconds, while the second client (which was launched shortly after the first) is already delayed by much longer period. This is to be expected as the server can only process one request at a time and will delay each request by 5 seconds.

We'll cover some server concurrency issues later, but for now, we want our clients to be able to continue their "normal work" while they wait for the response from the server. Since we're dealing with example code, "normal work" for our clients will be just waiting for the response, while blocking on incoming events (converted into callbacks). However, if the example programs would be graphical, the asynchronous approach would make it possible for them to react to user input. D-Bus by itself does not support cancellation of method calls once processing has started on the server side, so adding cancellation support would require a separate method call to the server. Since the server only handles one operation at a time, the current server cannot support method call cancellations at all.

## 1.3 Asynchronous method calls using stubs

When you run the `glib-bindings-tool`, it will already generate the necessary wrapping stubs to support launching asynchronous method calls. What is then left to do is implementing the callback functions correctly, processing the return errors and launching the method call.

```c
/* Pull in the client stubs that were generated with
   dbus-binding-tool */
#include "value-client-stub.h"
```

Listing 1.2: The client still pulls the generated stub code in like before (glib-dbus-async/client-stubs.c)

The client has been simplified so that it now only operates on `value1`. The callback that will be called from the stub code is presented next:

```c
/**
 * This function will be called when the async setvalue1 will either
 * complete, timeout or fail (our server however does not signal
 * errors, but the client D-Bus library might). When this example
 * program is left running for a while, you will see all three cases.
 *
 * The prototype must match the one generated by the dbus-binding-tool
 * (org_maemo_Value_setvalue1_reply).
 *
 * Since there is no return value from the RPC, the only useful
 * parameter that we get is the error object, which we'll check.
 * If error is NULL, that means no error. Otherwise the RPC call
 * failed and we should check what the cause was.
 */
static void setValue1Completed(DBusGProxy* proxy, GError *error,
                                                  gpointer userData) {

  g_print(PROGNAME ":%s:setValue1Completed\n", timestamp());
  if (error != NULL) {
    g_printerr(PROGNAME "       ERROR: %s\n", error->message);
    /* We need to release the error object since the stub code does
       not do it automatically. */
    g_error_free(error);
  } else {
    g_print(PROGNAME "       SUCCESS\n");
  }
}
```

Listing 1.3: Callback to use when the RPC method completes (glib-dbus-async/client-stubs.c)

Since the method call does not return any data, the parameters for the callback are at minimum (you'll always get those three). Handling errors must be done within the callback since errors could be delayed from the server and not visible immediately at launch time. Note that the callback will not terminate the program on errors. We do this on purpose in order to demonstrate common asynchronous problems below. The `timestamp` function is a small utility function to return a pointer to a string representing the number of seconds since the program started (useful to visualise the order of the different asynchronous events below).

```c
/**
 * This function will be called repeatedly from within the mainloop
 * timer launch code.
 *
 * It will launch asynchronous RPC method to set value1 with ever
 * increasing argument.
 */
static gboolean timerCallback(DBusGProxy* remoteobj) {

  /* Local value that we'll start updating to the remote object. */
  static gint localValue1 = -80;

  /* Start the RPC.
     This is done by calling the stub function that will take the new
     value and the callback function to call on reply getting back.

     The stub returns a DBusGProxyCall object, but we don't need it
     so we'll ignore the return value. The return value could be used
```

```
    to cancel a pending request (from client side) with
    dbus_g_proxy_cancel_call. We could also pass a pointer to
    user-data (last parameter), but we don't need one in this example.
    It would normally be used to "carry around" the application state.
    */
  g_print(PROGNAME ":%s:timerCallback launching setvalue1\n",
          timestamp());
  org_maemo_Value_setvalue1_async(remoteobj, localValue1,
                                  setValue1Completed, NULL);
  g_print(PROGNAME ":%s:timerCallback setvalue1 launched\n",
          timestamp());

  /* Step the local value forward. */
  localValue1 += 10;

  /* Repeat timer later. */
  return TRUE;
}
```

Listing 1.4: The timer callback that will now use the async launching stub code (glib-dbus-async/client-stubs.c)

Using the stub code is rather simple. For each generated synchronous version of a method wrapper, there will also be a `_async` version of the call. The main difference with the parameters is the removal of the GError pointer (since errors will be handled in the callback) and addition of the callback function to use when the method will complete, time out or encounter an error.

The `main` function remains the same from previous client examples (a once per second timer will be created and run from the mainloop until the program is terminated).

## 1.4 Problems with asynchronicity

When the simple test program is built and run, we'll see that everything starts off quite well. But at some point problems start to appear:

```
[sbox-DIABLO_X86: ~/glib-dbus-async] > make client-stubs
dbus-binding-tool --prefix=value_object --mode=glib-client \
  value-dbus-interface.xml > value-client-stub.h
cc -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include \
 -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -g -Wall \
 -DG_DISABLE_DEPRECATED -DNO_DAEMON -DPROGNAME=\"client-stubs\" \
 -c client-stubs.c -o client-stubs.o
cc client-stubs.o -o client-stubs -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/glib-dbus-async] > run-standalone.sh ./client-stubs
client-stubs:main Connecting to Session D-Bus.
client-stubs:main Creating a GLib proxy object for Value.
client-stubs: 0.00:main Starting main loop (first timer in 1s).
client-stubs: 1.00:timerCallback launching setvalue1
client-stubs: 1.00:timerCallback setvalue1 launched
server:value_object_setvalue1: Called (valueIn=-80)
server:value_object_setvalue1: Delaying operation
client-stubs: 2.00:timerCallback launching setvalue1
client-stubs: 2.00:timerCallback setvalue1 launched
client-stubs: 3.01:timerCallback launching setvalue1
client-stubs: 3.01:timerCallback setvalue1 launched
client-stubs: 4.01:timerCallback launching setvalue1
client-stubs: 4.01:timerCallback setvalue1 launched
client-stubs: 5.02:timerCallback launching setvalue1
client-stubs: 5.02:timerCallback setvalue1 launched
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=-70)
server:value_object_setvalue1: Delaying operation
client-stubs: 6.01:setValue1Completed
client-stubs       SUCCESS
client-stubs: 6.02:timerCallback launching setvalue1
client-stubs: 6.02:timerCallback setvalue1 launched
client-stubs: 7.02:timerCallback launching setvalue1
client-stubs: 7.02:timerCallback setvalue1 launched
...
client-stubs:25.04:timerCallback launching setvalue1
client-stubs:25.04:timerCallback setvalue1 launched
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=-30)
server:value_object_setvalue1: Delaying operation
client-stubs:26.03:setValue1Completed
client-stubs       SUCCESS
...
client-stubs:30.05:timerCallback launching setvalue1
client-stubs:30.05:timerCallback setvalue1 launched
...
client-stubs:36.04:setValue1Completed
client-stubs       ERROR: Did not receive a reply. Possible causes include:
 the remote application did not send a reply, the message bus security policy
 blocked the reply, the reply timeout expired, or the network connection was
 broken.
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=-10)
server:value_object_setvalue1: Delaying operation
client-stubs:36.06:timerCallback launching setvalue1
client-stubs:36.06:timerCallback setvalue1 launched
client-stubs:37.04:setValue1Completed
client-stubs       ERROR: Did not receive a reply. Possible causes include:
 the remote application did not send a reply, the message bus security policy
 blocked the reply, the reply timeout expired, or the network connection was
 broken.
[Ctrl+c]
[sbox-DIABLO_X86: ~/glib-dbus-async] >
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=30)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=40)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
server:value_object_setvalue1: Called (valueIn=50)
server:value_object_setvalue1: Delaying operation
server:value_object_emitSignal: Emitting signal id 0, with message 'value1'
...
```

What happens above is rather subtle. The timer callback in the client will launch once per second and do the RPC method launch. The server however still has the 5 second delay for each method call in it. We see the successive launches going on without any responses for a while. The first response comes back at about 6 seconds since client has started. At this point the server already has 4 other outstanding method calls that it hasn't handled. Slowly the method calls are accumulating at the server end and it doesn't deal with them quickly enough to satisfy the client.

After about 30 seconds, we start seeing the `setValue1Completed` callback invoked, but with the method call failing. We have managed to trigger the method call timeout mechanism. After this point, all the method calls that have accumulated into the server (into a message queue) will fail in the client, since they all will now return late, even if the server actually does handle them.

Once we terminate the client, we'll see that the server is still happily continuing serving the requests, oblivious to the fact that there is no client to process the responses.

The above test demonstrates quite brutally that you need to design your services properly so that there is a clearly defined protocol what to do in case a method call is delayed. You also might want to design a notification protocol to tell clients that something has completed, instead of forcing them to time out. Using D-Bus signals is one way, but you need to take care not to generate signals when no one is listening for them. This can be done by only sending signals when an long operation will finish (assuming you have documented this as part of your service description).

One partial fix would be for the client to track and make sure that only one method call to one service it outstanding at any given time. So instead of just blindly launching the RPC methods, it should defer launching if it hasn't yet got a response from the server (and the call hasn't timed out).

This fix is not complete however, since the same problem will manifest itself once there are multiple clients running in parallel and requesting the same methods. The proper fix is to make the server capable of serving multiple requests in parallel. Some hints on how to do this are presented later on.

## 1.5 Asynchronous method calls using GLib wrappers

Sometimes the interface XML will be missing, so you cannot run the `dbus-bindings-tool` to generate the stub code. The GLib wrappers are generic enough for you to be able to build your own method calls when necessary.

It is often easiest to start with some known generated stub code to see which parts you could possibly reuse (with modifications). This is what we'll do last, in order to make a version of the asynchronous client that will work without the stub generator.

We start by taking a peek at the stub generated code for the `setvalue1` call (when used asynchronously):

```
typedef void (*org_maemo_Value_setvalue1_reply) (DBusGProxy *proxy,
                                                 GError *error,
                                                 gpointer userdata);

static void
org_maemo_Value_setvalue1_async_callback (DBusGProxy *proxy,
                                          DBusGProxyCall *call,
                                          void *user_data)
{
  DBusGAsyncData *data = user_data;
  GError *error = NULL;
  dbus_g_proxy_end_call (proxy, call, &error, G_TYPE_INVALID);
  (*(org_maemo_Value_setvalue1_reply)data->cb) (proxy, error,
                                                data->userdata);
  return;
}

static
#ifdef G_HAVE_INLINE
inline
#endif
DBusGProxyCall*
org_maemo_Value_setvalue1_async (DBusGProxy *proxy,
                                 const gint IN_new_value,
                              org_maemo_Value_setvalue1_reply callback,
                                 gpointer userdata)

{
  DBusGAsyncData *stuff;
  stuff = g_new (DBusGAsyncData, 1);
  stuff->cb = G_CALLBACK (callback);
  stuff->userdata = userdata;
  return dbus_g_proxy_begin_call (
    proxy, "setvalue1", org_maemo_Value_setvalue1_async_callback,
    stuff, g_free, G_TYPE_INT, IN_new_value, G_TYPE_INVALID);
}
```

Listing 1.5: Generated stub code for the asynchronous setvalue1 (glib-dbus-async/value-client-stub.h)

What is notable in the code snippet above is that the `_async` method will create a temporary small structure that will hold the pointer to the callback function, and a copy of the userdata pointer. This small structure will then be passed to `dbus_g_proxy_begin_call` along with the address of the generated callback wrapper function (`org_maemo_Value_setvalue1_async_callback`). The GLib async launcher will also take a function pointer to a function to use when the supplied "user-data" (in this case the small structure) will need to be disposed (after the call). Since it uses `g_new` to allocate the small structure, it passes `g_free` as the freeing function. Next comes the argument specification for the method call, which obeys the same rules as the LibOSSO ones before.

On RPC completion, the generated callback will be invoked, and it will get the real callback function pointer and the userdata as its "user-data" parameter. It will first collect the exit code for the call with `dbus_g_proxy_end_call` and unpacks the data and invokes the real callback. After returning, the GLib wrappers (which called the generated callback) will call `g_free` to release the small structure and the whole RPC launch will end.

We next re-implement pretty much the same logic, but also dispose of the

small structure, since we are going to implement our callback directly, not as a wrapper-callback (it also saves us from doing one memory allocation and one free).

We'll start with the RPC asynchronous launch code:

```
/**
 * This function will be called repeatedly from within the mainloop
 * timer launch code.
 *
 * It will launch asynchronous RPC method to set value1 with ever
 * increasing argument.
 */
static gboolean timerCallback(DBusGProxy* remoteobj) {

  /* Local value that we'll start updating to the remote object. */
  static gint localValue1 = -80;

  /* Start the first RPC.
     The call using GLib/D-Bus is only slightly more complex than the
     stubs. The overall operation is the same. */
  g_print(PROGNAME ":timerCallback launching setvalue1\n");
  dbus_g_proxy_begin_call(remoteobj,
                          /* Method name. */
                          "setvalue1",
                          /* Callback to call on "completion". */
                          setValue1Completed,
                          /* User-data to pass to callback. */
                          NULL,
                          /* Function to call to free userData after
                             callback returns. */
                          NULL,
                          /* First argument GType. */
                          G_TYPE_INT,
                          /* First argument value (passed by value) */
                          localValue1,
                          /* Terminate argument list. */
                          G_TYPE_INVALID);
  g_print(PROGNAME ":timerCallback setvalue1 launched\n");

  /* Step the local value forward. */
  localValue1 += 10;

  /* Repeat timer later. */
  return TRUE;
}
```

Listing 1.6: Launching RPC methods using GLib functions without stubs (glib-dbus-async/client-glib.c)

And the callback that will be invoked on method call completion, timeouts or errors:

```
/**
 * This function will be called when the async setvalue1 will either
 * complete, timeout or fail (same as before). The main difference in
 * using GLib/D-Bus wrappers is that we need to "collect" the return
 * value (or error). This is done with the _end_call function.
 *
 * Note that all callbacks that are to be registered for RPC async
 * notifications using dbus_g_proxy_begin_call must follow the
 * following prototype: DBusGProxyCallNotify .
```

```
 */
static void setValue1Completed(DBusGProxy* proxy,
                               DBusGProxyCall* call,
                               gpointer userData) {

  /* This will hold the GError object (if any). */
  GError* error = NULL;

  g_print(PROGNAME ":setValue1Completed\n");

  /* We next need to collect the results from the RPC call.
     The function returns FALSE on errors (which we check), although
     we could also check whether error-ptr is still NULL. */
  if (!dbus_g_proxy_end_call(proxy,
                             /* The call that we're collecting. */
                             call,
                             /* Where to store the error (if any). */
                             &error,
                             /* Next we list the GType codes for all
                                the arguments we expect back. In our
                                case there are none, so set to
                                invalid. */
                             G_TYPE_INVALID)) {
    /* Some error occurred while collecting the result. */
    g_printerr(PROGNAME " ERROR: %s\n", error->message);
    g_error_free(error);
  } else {
    g_print(PROGNAME " SUCCESS\n");
  }
}
```

Listing 1.7: Handling the method response and ending the async RPC call (glib-dbus-async/client-glib.c)

We no longer need the generated stub code, so the dependency rules for the stubless GLib version will also be somewhat different:

```
client-glib: client-glib.o
  $(CC) $^ -o $@ $(LDFLAGS)
# Note that the GLib client doesn't need the stub code.
client-glib.o: client-glib.c common-defs.h
  $(CC) $(CFLAGS) -DPROGNAME=\"$(basename $@)\" -c $< -o $@
```

Listing 1.8: Simpler dependencies for the stub code (glib-dbus-async/Makefile)

Since the example program logic hasn't changed from the previous version, testing `client-glib` is not presented here (you are of course free to test it yourself since the source code contains the fully working program). This version of the client will also launch the method calls without waiting for previous method calls to complete.