Maemo Diablo D-Bus server design issues

Training Material

February 9, 2009

Contents

Chapter 1

D-Bus server design issues

1.1 Definition of a server

When speaking about software, a server is commonly understood to mean some kind of software component that provides some service to its clients. In Linux, servers are usually implemented as _daemons_, which is a technical term for a process that has detached from the terminal session, and done other preparatory actions so that it will stay running on the background until it terminates (or is terminated).

Sometimes you might hear people refer to servers as _engines_, but it is a more generic term and normally isn't related directly to the way how a service is implemented (as a separate process, or as part of some library, directly used from within a client process). Broadly defined, an engine is the part of application that implements the functionality, but not the interface, of an application. In Model-View-Controller, it would be the Model.

Our servers so far have been running without daemonisation, in order to display debugging messages on the terminal/screen more easily. Often a server can be started with a `"--stay-on-foreground"` option (or `-f` or something similar), which means that they will not daemonise. This is a useful feature to have since it will allow you to use simpler outputting primitives when testing your software.

By default, when a server will daemonise, its output and input files will be closed, so reading user input (from the terminal session, not GUI) will fail, as will each output write (including `printf` and `g_print`).

1.2 Daemonisation

The objective of turning a process into a daemon is to detach it from its parent process and create a separate session for it. This is necessary so that parent termination doesn't automatically cause the termination of the server as well. There is a library call that will do most of the daemonisation work, called `daemon` but it is also instructive to see what is necessary (and common) to do when implementing the functionality yourself:

- `fork` the process so that the original process can be terminated and this

will cause the child process to move under the system `init` process.

- Create a new session for the child process with `setsid`.
- Possibly switch working directory to root (/) so that the daemon won't keep filesystems from being unmounted.
- Setup up a restricted `umask` so that directories and files that will be created by the daemon (or its child processes) will not create publicly accessible objects in the filesystem. In Internet Tablets this doesn't really apply since the devices only have one user.
- Close all standard I/O file descriptors (and preferable files too) so that if the terminal device will close (user logs out), that won't cause `SIGPIPE` signals to the daemon when it next accesses the file descriptors (by mistake or intentionally because of `g_print`/`printf`). It is also possible to reopen the file descriptors so that they'll be connected to a device which will just ignore all operations (like `/dev/null` which is used with `daemon`).

The `daemon` function allows you to select whether you want to change the directory and to close the open file descriptors and we'll utilise that in our servers in the following way:

```
#ifndef NO_DAEMON

  /* This will attempt to daemonize this process. It will switch this
     process' working directory to / (chdir) and then reopen stdin,
     stdout and stderr to /dev/null. Which means that all printouts
     that would occur after this, will be lost. Obviously the
     daemonization will also detach the process from the controlling
     terminal as well. */
  if (daemon(0, 0) != 0) {
    g_error(PROGNAME ": Failed to daemonize.\n");
  }
#else
  g_print(PROGNAME
          ": Not daemonizing (built with NO_DAEMON-build define)\n");
#endif
```

Listing 1.1: Adding daemonisation support to server.c based on a define (glib-dbus-sync/server.c)

This define is then available to the user inside the Makefile:

```
# -DNO_DAEMON : do not daemonize the server (on a separate line so can
#               be disabled just by commenting the line)
ADD_CFLAGS += -DNO_DAEMON

# Combine flags
CFLAGS  := $(PKG_CFLAGS) $(ADD_CFLAGS) $(CFLAGS)
```

Listing 1.2: Daemonisation as a compile time option driver by make (glib-dbus-sync/Makefile)

Combining the options so that `CFLAGS` is appended to the Makefile provided defaults allows the user to override the define as well:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > CFLAGS='-UNO_DAEMON' make server
dbus-binding-tool --prefix=value_object --mode=glib-server \
  value-dbus-interface.xml > value-server-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 -UNO_DAEMON
   -DPROGNAME=\"server\" -c server.c -o server.o
cc server.o -o server -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
```

Undoing a define using an environmental variable, without modifying the Makefile

Since all `-D` and `-U` options will be processed from left to right by `gcc`, this allows the `-UNO_DAEMON` to undefine the symbol that is preset in the Makefile. If the user doesn't know this technique, it's also possible to edit the Makefile directly. Grouping all additional flags that the user might be interested to the top of the Makefile will make this simpler (for the user).

Running the server with daemonisation support is done as before, but this time we leave out the `&` (don't wait for child exit) token for the shell:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > 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:main Registering it on the D-Bus.
server:main Ready to serve requests (daemonizing).
[sbox-DIABLO_X86: ~/glib-dbus-sync] >
```

Server running as daemon

Since server messages will not be visible any more, we need some other mechanism to find out that the server is still running:

```
[sbox-DIABLO_X86: ~/glib-dbus-sync] > ps aux | grep "./server" | grep -v pts
user 8982  0.0  0.1 2780 664 ? Ss 00:14 0:00 ./server
```

The slightly convoluted way of using `grep` was necessary to only list those lines of the `ps` report which have `./server` in them and remove the lines which do not have `pts` in them (so that we see processes which have no controlling terminals).

We could've used the client to test whether the server responds as well, but the above technique is slightly more general. If you have the `pstree` tool available, you could run it with `-pu` options to see how the processes relate to each other, and that our demonised server is running directly as a child of `init` (which was the objective of the `fork`).

1.3 Event loops and power consumption

Most modern CPUs (even for desktops and servers) allow multiple levels of power savings to be selected. Each of the levels will be progressively more power-conservative, but there is always a price involved. The deeper the power saving level required, the more time it normally takes to achieve it, and the more time it also takes to come out of it. In some CPUs it also requires special sequences of instructions to run, hence taking extra power itself.

All this means that changing the power state of the CPU should be avoided when possible. Obviously this is in contrast in the requirement to conserve

battery life, so in effect, what is needed is to require the attention of the CPU as rarely as possible.

One way at looking the problem field is contrasting event-based and polling-based programming. A code which continuously checks for some status and only occasionally does useful work, is clearly keeping the CPU from powering down properly. This model should be avoided at all costs, or at least restricting its use to bare minimum if no other solution is possible.

In contrast, event-based programming is usually based on execution of callback functions when something happens, without requiring a separate polling loop. This then leaves the question on how to trigger the callbacks so that they will be issued when something happens. Using timer callbacks might seem like a simple solution, so that we continuously (once per second or more often) will check for some status and then possibly react to the change in status. This model is undesirable as well, since the CPU won't be able to enter into the deep sleep modes, but fluctuate between full power and high-power states.

Most operating system kernels provide a mechanism (or multiple mechanisms) by which a process can be woken up when data is available, and kept off the running queue of the scheduler otherwise. The most common mechanism in Linux is based around the `select/poll` system calls which are useful when waiting for change in status for a set of file descriptors. Since most of the interesting things in Linux can be represented as a "file" (an object supporting `read` and `write` system calls), using `select` and `poll` is quite common. However, when you're writing software that uses GLib (implicitly like in GTK+ or explicitly like in our non-GUI examples), you will use the GMainLoop structure instead. Internally it will use the event mechanism available on your platform (`select`/`poll`/others), but your program will need to register callbacks, start the main loop execution and then just execute the callbacks as they come.

If you then have some file descriptors (network sockets, open files, etc), you may integrate them into the GMainLoop using GIOChannels (please see the GLib API reference on this).

This still leaves the question of using timers and callbacks that are triggered by timers. You should avoid them when:

- You plan to use the timer at high frequencies (> 1 Hz) for long periods of time (> 5 sec).
- There is a mechanism that will trigger a callback when something happens, instead of forcing you to poll for the status "manually" or re-execute a timer callback that does the checking.

As an example, the LibOSSO program (`FlashLight`) that was covered before, will have to use timers in order to keep the backlight active. However, the timer is very slow (only once per 45 seconds), so this is not a big issue. Also, in `flashlight`'s defense, the backlight is on all the time, so having a slow timer won't hurt battery life very much anyway.

As an another example, consider a long lasting download operation which proceeds slowly, but steadily. You might want to consider whether updating a progress bar after each small bit of data is received makes sense (it doesn't normally). Instead you will want to keep track of when was the last time when you updated a progress bar, and if enough time has passed since the last time, update the GUI. In some cases this will allow the CPU to be left in somewhat

lower power state than full-power and will allow it to fall back to sleep more quickly.

Having multiple separate programs running each having their own timers presents another interesting problem. Since the timer callback is not precise, at some time the system will be waking at a very high frequency, handling each timer separately (the frequency and the number of timers executing in the system is something which you cannot control from a single program, but instead is a system wide issue).

If you're planning a GUI program, it is fairly easy to avoid contributing to this problem, since you can get a callback from LibOSSO which will tell you when your program is "on top" and when not visible. When not visible, you won't need to update the GUI, especially with timer based progress indicators and similar.

Since servers do not have a GUI (and their visibility is not controlled by the window manager), such mechanism does not exist. One possible solution in this case would be avoiding using timers (or any resources for that matter) when the server does not have any active clients. Only use resources when you get a client connection, or need to actually do something. As soon as it becomes likely that your server won't be used again, you will want to release the resources (remove the timer callbacks, and so on).

If you can, try to utilise the D-Bus signals available on the system bus (or even the hardware state structure via LibOSSO) to throttle down activity based on the conditions in the environment. Even if you're doing a non-GUI server, you will want to listen to the system shutdown signals, as they will tell your process to shutdown gracefully.

All in all, designing for a dynamic low powered environment is not always simple. Four simple rules will hold for most cases (all of them being important):

- Avoid doing extra work when possible.
- Do it as fast as possible (while trying to minimise resource usage).
- Do it as rarely as possible.
- Keep only those resources allocated that you need to get the work done.

For GUI programs one will have to take into account the "graphical side" of things as well. Making a GUI that is very conservative in its power usage will most of the time be very simple, provide little excitement to users and might even look quite ugly. Your priorities might lie in a different direction.

1.4 Supporting parallel requests

The value object server with delays has one major deficiency: it can only handle one request at a time, while blocking the progress of all the other requests. This will be a problem if multiple clients will use the same server at the same time.

Normally one would add support for parallel requests by using some kind of multiplexing mechanism right on top of the message delivery mechanism (libdbus in this case).

One can group the possible solutions around three models:

- Launching a separate thread to handle each request. This might seem like an easy way out of the problem, but coordinating access to shared resources (object states in this case) between multiple threads is prone to cause synchronisation problems and makes debugging much harder. Also, performance of such an approach would depend on efficient synchronisation primitives in the platform (which might not always be available) as well as light weight thread creation and tear-down capabilities of the platform.

- Using an event-driven model that supports multiple event sources simultaneously and "wakes up" only when there is an event on any of the event sources. The `select` and `poll` (and `epoll` on Linux) are very often used in these cases. Using them will normally require an application design that is driven by the requirements of the system calls (i.e., it is very difficult to retrofit them into existing "linear" designs). However, the event based approach normally outperforms the thread approach, since there is no need for synchronisation (when implemented correctly) and there will only be one context to switch from the kernel and back (there will be extra contexts with threads). GLib provides an high-level abstraction on top of the low level event programming model, in the form of GMainLoop. One would use GIOChannel objects to represent each event source and register callbacks that will be triggered on the events.

- Using `fork` to create a copy of the server process so that the new copy will just handle one request and then terminate (or return to the pool of "servers"). The problem here is the process creation overhead and lack of implicit sharing of resources between the processes. One would have to arrange a separate mechanism for synchronisation and data sharing between the processes (using shared memory and proper synchronisation primitives). In some cases resource sharing is not actually required, or happens at some lower level (accessing files), so this model shouldn't be automatically ruled out even if it seems quite heavy at first. Many static content web-servers use this model because of its simplicity (and they don't need to share data between themselves).

The problem for the slow server however lies elsewhere: the GLib/D-Bus wrappers do not support parallel requests directly. Even using the `fork`-model would be problematic as there would be multiple processes accessing the same D-Bus connection. Also, this problem is not specific to the slow server only. You will bump into the same issues when using other high-level frameworks (like GTK+) whenever cannot complete something immediately because not all data is present in your application. In the latter case it is normally sufficient to use the GMainLoop/GIOChannel approach in parallel with GTK+ (since it uses GMainLoop internally anyway), but with GLib/D-Bus there's no mechanism which you could use to integrate your own multiplexing code (no suitable API exists).

In this case, the solution would be picking one of the above models, and then using libdbus functions directly. In effect, this would require a complete rewrite of the server, forgetting about the GType implementation and possibly creating a light-weight wrapper for integrating libdbus functions into GLib GMainLoop mechanism (but dropping support for GType).

Dropping support for the GType and stub code will mean that you would have to implement the introspection support manually and be dependent on possible API changes in libdbus in the future.

Another possible solution would be to "fake" the completion of client method calls, so that the RPC method will complete immediately, but the server will continue (using GIOChannel integration) processing the request until it will really complete. The problem in this solution is that it is very difficult to know which client actually issued the original method call and how to communicate the final result (or errors) of the method call to the client once it does complete. One possible model here would be using signals to broadcast the end result of the method call, so that the client will get the result at some point (assuming the client is still attached to the message bus). Needless to say, this is quite inelegant and difficult to implement correctly, especially since sending signals will cause unnecessary load by waking up all the clients on the bus (even if they're not interested in that particular signal).

In short, there is no simple solution that works properly when GLib/D-Bus wrappers are used.

1.5 Debugging

The simplest way to debug your servers will be intelligent usage of print out of events in the code sections that are relevant. Tracing everything that goes on rarely makes sense, but having a reliable and working infrastructure (in code level) will help. One such mechanism is utilising various built in "magic" that gcc and cpp provide. In the server example, a macro called `dbg` is used, which will expand to `g_print` when the server is built as non-daemonizing version. If the server will become a daemon, the macro expands to "nothing", meaning that no code will be generated to format the parameters or to even access them. You will want to extend this idea to support multiple levels of debugging, and possibly use different "subsystem" identifiers so that you can switch one subsystem on or off depending on what you're debugging.

The `dbg` macro utilises the `__func__` symbol, which expands to the function name where the macro will be expanded, which is quite useful so that you don't have to add the function name explicitly:

```
/* A small macro that will wrap g_print and expand to empty when
   server will daemonize. We use this to add debugging info on
   the server side, but if server will be daemonized, it doesn't
   make sense to even compile the code in.

   The macro is quite "hairy", but very convenient. */
#ifdef NO_DAEMON
#define dbg(fmtstr, args...) \
  (g_print(PROGNAME ":%s: " fmtstr "\n", __func__, ##args))
#else
#define dbg(dummy...)
#endif
```

Listing 1.3: Adding debug message support to a server based on a define (glib-dbus-sync/server.c)

Using the macro is then quite simple, as it will look and act like a regular `printf`-formatting function (`g_print` included):

```
  dbg("Called (internal value2 is %.3f)", obj->value2);
```

Listing 1.4: Utilising the dbg macro (glib-dbus-sync/server.c)

The only small difference being that you don't have to explicitly add the trailing newline (`\n`) into each call, since it will be automatically added.

Assuming `NO_DAEMON` is defined, the macro would expand to the following output when the server would be run:

```
server:value_object_getvalue2: Called (internal value2 is 42.000)
```

For larger projects, you will also want to combine `__file__` so that tracing multi file programs will become easier.

Coupled with proper test cases (which would be using the client code, and possibly also `dbus-send` in D-Bus related programs), this is a very powerful technique and often much easier than single stepping through your code with a debugger (`gdb`) or setting and evaluation breakpoints. You will also be interested in using Valgrind to help you detect memory leaks (and some other errors). More information on these topics and examples are available in maemo.org documentation.