CSC 195.01 2013S *Technologies for Mediascripting*

Class 06: Inter-Application Communication with D-Bus (2): Writing Servers

Held: Thursday, 28 February 2013

Summary: We continue our exploration of D-Bus by considering the typical form of a D-Bus server written using the GDBus library.

Related Pages:

- EBoard.

Notes:

- No class next week. I'll be off at SIGCSE.
- I may use the whiteboard board more today than normal. Sorry.
- We won't go over your homework.
- Today's code can be found in `Examples/DBus/Server`.
- We're working a bit with `GVariants` today. You may find it useful to read the documentation at http://developer.gnome.org/glib/2.28/glib-GVariant.html.
- Homework, part 1: Take one of the sample servers and add a few useful methods (E.g., to multiply two numbers).
- Homework, part 2: (Will be sent next week. Will involve looking at sample client code.)

Overview:

- Basic issues
- Preparation
- The `main` function
- When the bus is acquired
- Detour: GVariants
- Handling events
- Adding selected capabilities

Basic Issues in Writing a DBus Server

- So, you want to write a D-Bus server, what are things you need to know?
- More precisely, you want to write a D-Bus server using the glib and gio libraries. What are the things you have to do.
- As you may recall, D-Bus has an object-oriented model
    - Each service must register a name on the bus
    - Each service must have one or more objects that provide the services.
- So, the first thing is *register your name*

- And the second thing is *publish objects*
- Of course, publishing objects needs a lot of substeps. So, what do we need to do
  - We need to *describe the interfaces the object supports*
    - Internally
    - Externally
  - We need to publish those interface on the D-Bus
- Once we've published an object, we need to be able to *respond to requests*. That's typically done using an event loop. (Conveniently, glib/gio provide the event loop, along with a callback system that ensures that certain functions get called when requests arrive).
- We'll have handlers for
  - Method calls (`handle_method_call`)
  - Getting a property (`handle_get_property`)
- Setting a property (`handle_set_property`)
- Because glib often does things asynchronously, we'll also have handlers for
  - Successful acquistion of the bus (`on_bus_acquired`)
  - Successful acquisition of the name on the bus (`on_name_acquired`)
  - Loss of a name (for whatever reason) (`on_name_lost`)

Preparation

- You need to make sure that you can load the glib and gio libraries. That may depend on your installation.
  - If you're working on your own *nix installation, make sure that you have a recent version of glib and gio installed.
  - If you're working on MathLAN, I've set up some scripts that help
    - `source /home/rebelsky/Glimmer/scripts/slashglimmer.sh`
    - `source /home/rebelsky/Glimmer/scripts/env.sh`
  - If you're working in MathLAN, you may have to be on church
    - `ssh -X church.cs.grinnell.edu`
  - Warning! Only one service can own a name at a time. If you make a copy of my code, choose a different service name (and maybe a different object/interface name).
- Set up your Makefile to use gio

```
CFLAGS=-g -Wall $(shell pkg-config --cflags gio-2.0)
LDLIBS=$(shell pkg-config --libs gio-2.0)
```

- You need to make some decisions about names
  - Pick a name for the service (typically uses something like a reverse domain name)
    - edu.grinnell.glimmer.guide.SampleServer0
  - Pick a name for the object(s)
    - edu.grinnell.glimmer.guide.SampleObject0
  - Pick a name for the interface(s)
    - edu.grinnell.glimmer.guide.Interface0
- You need to design the interface (preferably as XML, at least at first).

Your `main`

- Set up glib
- Build an internal representation of the interfaces
- Request the name on the bus (`g_bus_own_name`)
    - Callbacks indicate when the bus and name have been acquired
- Start the loop
- Clean up

When the bus is acquired

- Register objects.
- Perhaps do other things (e.g., set up events to happen later).

A Detour: GVariants

- We're about to write the various handlers.
- How can we write handlers that will deal with different types of values? (After all, different properties may have different types.)
- The gio solution: have one generic type (`GVariant`) and provide methods for packing data into that type and extracting data from that type.
- To create a new `GVariant` use `g_variant_new(`*`type-abbrev`*`, `*`values`*`)` or `g_variant_new_`*`xxx`*`(`*`value`*`)`
- To extract a value from `GVariant` use `g_variant_get(`*`type-abbrev`*`, `*`locations`*`)` or `g_variant_get_`*`xxx`*`(`*`value`*`)`

Handling Stuff

- Details in the code.

Adding Features

- A method that returns a value. [v1]
- A property (readable and writable), but not implemented. [v2]
- A property (readable and writable). [v3]
- User data.
- ...
