Introduction to the GConf library

by

Havoc Pennington

hp@redhat.com

This article introduces the concepts behind the GConf configuration library scheduled to ship with the next major revision of the GNOME development environment.

Introduction: What is GConf?

GConf is a configuration data storage mechanism scheduled to ship with GNOME 2.0. GConf does work without GNOME however; it can be used with plain GTK+, Xlib, KDE, or even text mode applications as well. As of March 2000, there is no stable release of GConf yet, but the library is feature-complete.

The GNOME desktop is currently in the Windows 3.1 era with respect to application configuration data; applications store their configuration in flat .INI-style files. Windows later introduced a more sophisticated solution, the Registry. However, the Registry still has a number of shortcomings:

- It's very difficult to manage a large number of computers; system administrators can't install defaults, or push changes out to multiple user computers. Proprietary add-on tools exist that try to resolve this problem in various (rather frightening) ways.
- The registry contains lots of undocumented, cryptically-formatted data, and regedit is therefore dangerous and difficult to use.
- The registry becomes corrupted, and this tends to destroy the whole operating system installation.

GConf attempts to leapfrog the registry concept. It's a *library* which provides a simple configuration data storage interface to applications, and also an *architecture* that tries to make things easy for system administrators.

Here are some of the features of GConf:

- Replaceable backend architecture. GConf currently has a single backend that stores configuration data in XML-format text files; however, the architecture allows a Registry-like binary database backend, an LDAP backend, or even a full-blown SQL database backend. The backend used is configurable by the system administrator. This is a valuable feature for IS deparatments managing large numbers of computers.
- Configuration key documentation. Applications can install documentation along with their configuration data, explaining the possible settings and the effect of each configuration key. A regedit-style tool can display this documentation, making it much easier to customize applications without breaking them. The GConf database

also stores useful meta-information such as the application that owns a key, when the key was last changed, etc.

- Data change notification service. If configuration data is changed, all interested applications are notified. The notification service works across the network, affecting all login sessions for a single user.

  This means that programs built from components (where each component may be in a separate process) are much easier to write, because if one component changes a setting the others can discover the change and update themselves. For example, GNOME's new Nautilus file manager is actually a collection of applications, including an embedded web browser, and various navigation components. The components communicate via CORBA. However, you want a single preferences dialog located in the top-level "shell" component. Without GConf, a custom protocol would have to be invented for each preferences setting, to notify embedded components of changes to user preferences.

  Notification is also useful if multiple application instances are running. GNOME 2.0 uses this feature to let user interface configuration take effect on-the-fly without restarting any applications; if you turn off toolbar icons, for example, toolbar icons will immediately disappear in all running apps.

- The client API is very abstract, which allows us to change its implementation at a later time without a big headache. Because a good implementation is a complex problem, this is important. I also like to think the API is simple to use.
- GConf does proper locking so that the configuration data doesn't get corrupted when multiple applications try to use it.

Implementation Overview

Keys and Values

GConf is not a full-blown database by any means. It stores simple key-value pairs. This keeps the implementation simple and efficient. In particular, the following database features are not currently implemented:

- Queries
- Efficient storage of large chunks of data (such as entire documents)
- Transactions

Applications that require these sophisticated features should use LDAP or a full-scale database. Transactions may be added eventually, and are currently in the API in the form of the GConfChangeSet object, but for now change sets are not stored atomically.

A GConf key looks like a UNIX filename. Keys are organized into "directories" just as UNIX files are. For example, the `/desktop` directory stores desktop settings, and the hypothetical key `/desktop/standard/background-color` might store the color of the desktop background. The GConf documentation establishes a number of conventional directory names; you should follow the guidelines in the documentation when naming your application keys.

Values have types; the primitive types are integers, booleans, strings, and floating point numbers. Also, lists and pairs of primitive types are possible. Recursive types (list of list) are not permitted. Types have no size restriction; strings can be as long as you like, though unreasonably large strings (such as an entire XML document) will cause performance problems.

Configuration Server

GConf is implemented as a per-user daemon called `gconfd`. `gconfd` actually accesses the user's configuration backends by dynamically loading the appropriate backend modules. `gconfd` is also in charge of sending out notifications to interested applications when configuration values are modified.

One `gconfd` should exist per user at any given time. If a user logs in to two machines simultaneously, `gconfd` will run on the first machine, and the second machine will access it over the network.

`gconfd` talks to applications using CORBA as a transport. This is purely an implementation detail; a custom protocol over sockets could have been used or could be used in the future. Directly accessing the CORBA interface is *not* supported. Application programmers do not see CORBA while using GConf.

In general, `gconfd` is not visible to the application programmer. It is automatically launched as required. (Internally, the OAF object activation framework is used to obtain an object reference for the `gconfd` CORBA server.)

Because all database access goes via `gconfd`, locking is not really an issue. `gconfd` also aggressively caches configuration values, because all applications can share the same cache.

Schemas

In addition to the primitive types mentioned earlier (integer, string, etc.) the GConf database can store a *schema*. A schema is a description of a key; it includes a short description of the key, a longer documentation paragraph, the type of the key, a default value for the key, and the name of the application owning the key. Schemas are primarily intended for use by a regedit-style tool; such a tool can display the documentation for a key, and validate the type of any values you enter for the key.

Because schemas are simply values in the database, they can be referenced by key name. Conventionally, all schemas are below the `/schemas` toplevel directory.

Applications come with a special *schema file* which encodes the schema information in a human-editable text format. When the application is installed, the `gconftool` program shipped with GConf is used to insert the schema information into the GConf database. Also, all applications should install their schema file to a system-wide directory (specified in the GConf documentation). This allows system administrators to re-install all schemas into a different GConf backend with a single command (something like `gconftool -install-schema-file /etc/gconf/schemas/*`).

Application Programming

GConf makes it as simple as possible for application programmers to store and retrieve their configuration data. However, the general structure of configuration data

storage and retrieval is somewhat different from the one you might use with configuration files.

Model-View-Controller Architecture

Applications using GConf effectively should structure their preferences and configuration code according to the famous *Model-View-Controller* (MVC) design pattern. In this pattern, your application contains three separate objects:

1. The *model* traditionally models a real-world object; but more generally, it is the data you are planning to present to the user and allow the user to edit. In the GConf case, the model is the configuration database (in the big picture) and a particular configuration key (from a smaller-scale point of view). For example, the configuration key `/desktop/gnome/menus/show-icons` can be thought of as a model; it stores a boolean value indicating whether to show icons next to menu items in GNOME menus.
2. The *view* somehow displays or represents your model. It receives some sort of notification when the model changes, and updates itself accordingly. For example, the GNOME menu item widgets listen for changes to the `/desktop/gnome/menus/show-icons` configuration key, and show or hide icons accordingly. An important point about views is that there can be more than one of them; you can have many menu items, without changing the model.
3. The *controller* modifies the model. In the menu icons example, a GNOME control panel might be the controller; users use it to set the `/desktop/gnome/menus/show-icons` key to true or false.

An MVC architecture has a number of advantages. Most notably, it encourages code reuse and modularity; you can have any number of views, and different types of views, without changing the model or the controller; you can have any number of controllers, and different types of controller, without changing the model or the view. The main controller may be the GNOME control panel, but if the key is changed by some other application, that will also work properly. The primary view may be the GNOME menu item code, but if some different menu item code wanted to monitor and honor this setting, it could certainly do so.

GConf implements an exciting enhancement to the basic MVC pattern: the model is process-transparent. That is, the view and controller have no knowledge of which process contains the model; if you set a configuration key, then all views in all interested processes will be notified of the change. In a world of component technology, this is extremely useful.

All buzz-acronyms aside, all MVC means for you as an application programmer is that every section of code only has to be concerned with *directly related* configuration issues. For example, the menu item code only monitors the keys that affect how menu items are displayed. It is not concerned with any other keys, or with setting or saving that setting.

In contrast, with a simple configuration file, typically you have a global data structure which stores all your preferences, and you load and save it as a single unit. To add a setting, you have to modify the global structure. Whenever you load settings, you have to add code to the global preferences code that knows about all the interested parts of the program and notifies them (via some ad-hoc mechanism). It's kind of unpleasant and leads to poor modularity in a single process; but once multiple processes are involved, coming up with a suitable ad-hoc mechanism for change notification becomes a really significant problem.

Major Data Types

GConfEngine

A GConfEngine represents a configuration database (normally a connection to `gconfd`). You can store values in a GConfEngine, and retrieve values from one. You can also ask a GConfEngine to monitor a key or directory for changes, and invoke an application-supplied callback if changes occur. Each time you ask a GConfEngine to notify you of changes, it registers a request for notification with `gconfd`. Thus, notification requests are relatively expensive. With GConfClient, they are relatively cheap.

GConfClient

GConfClient is located in a separate library from GConf itself, because to use it you must link with GTK+. GConfClient derives from GtkObject and is a wrapper for the GConfEngine type. It adds a few nice features:

- It can keep a client-side cache of values. Sometimes this is just memory overhead, so you'll want to turn it off. However, if you access the same keys frequently it can be a performance win.
- It can do client-side notification request dispatching. This means that if you want to monitor the keys `/foo/bar` and `/foo/baz`, GConfClient registers a single notification request with `gconfd`; when the notification comes from `gconfd`, GConfClient decides which key has changed and dispatches to the proper application callback. With GConfEngine, each callback involves registering a request with `gconfd`.
- It uses the GTK signal system. Instead of using GConf's custom callback API for notification, you can simply connect to a `"value_changed"` signal.
- Finally, GConfClient provides default error handlers. This allows you to ignore errors, and let GConfClient present them to the user via an error dialog.

GConfValue

When GConf needs to pass around a dynamically-typed value retrieved from or to be stored in the database, it uses a type-tagged union called GConfValue. For example, the `gconf_get()` function takes a GConfEngine and a key name as arguments and returns a GConfValue. There are also convenience wrappers such as `gconf_get_string()` and `gconf_get_int()` that allow you to avoid the pesky GConfValue union.

GConfChangeSet

Modifications to the GConf database can be grouped into *change sets*, or collections of changes. A change is a new value for a key, or a directive to unset a key's current value. For now, the GConfChangeSet is primarily a convenience mechanism for managing the set of changes a user has made via a preferences dialog. However, in the future committing a change set may be an atomic operation (that is, GConf could be extended to support transactions).

GConfError

If you explicitly handle errors, rather than allowing the GConfClient default error handler to report them to the user, you receive information about an error in an object called GConfError. This contains a verbose error description, and an error code number.

GNOME Integration

Version 2.0 of GNOME automatically integrates with GConf, making it simpler to use the library. It does the following:

- Keeps a global GConfClient that's easy to access with a `gnome_get_gconf_client()` function.
- Automatically uses GNOME dialogs for default error handling.
- Provides convenience functions to extract and insert a GConfValue from various kinds of widget. For example, there is a function to get the string inside a GtkEntry as a GConfValue.

Coding Example

This section contains two simple example programs; one displays the value of a key (and updates the displayed value when the key changes), and the other allows you to change the value of the same key. These examples should make the MVC architecture more concrete, and give you a general idea how the GConf API works.

Although the examples use GConfClient, remember that it's also possible to use GConfEngine directly, to avoid linking to GTK. Also, in the next version of GTK and glib the object system will be moved to glib, so GConfClient can move to the main GConf library.

A Simple View

This program creates a GtkLabel that displays the current value of the key `/extra/test/directory/key`. (By the way, directories do have conventional names, described in the GConf documentation; in this case, the directory `/extra` is similar to the `X-` prefix used in mail headers and MIME types, that is, it's for nonstandard extensions to the standard. Here our "nonstandard extension" is a trivial little test program.)

I won't explain all the details of this program, since this article is just a general overview of GConf rather than a full tutorial; the GConf documentation goes into detail about the functions and data types you see here.

```
/* A very simple program that monitors a single key for changes. */

#include <gconf/gconf-client.h>
#include <gtk/gtk.h>
```

```
void
key_changed_callback(GConfClient* client,
                     guint cnxn_id,
                     const gchar* key,
                     GConfValue* value,
                     gboolean is_default,
                     gpointer user_data)
{
  GtkWidget* label;

  label = GTK_WIDGET(user_data);

  if (value == NULL)
    {
      gtk_label_set(GTK_LABEL(label), "<unset>");
    }
  else
    {
      if (value->type == GCONF_VALUE_STRING)
        {
          gtk_label_set(GTK_LABEL(label), gconf_value_string(value));
        }
      else
        {
          gtk_label_set(GTK_LABEL(label), "<wrong type>");
        }
    }
}

int
main(int argc, char** argv)
{
  GtkWidget* window;
  GtkWidget* label;
  GConfClient* client;
  gchar* str;

  gtk_init(&argc, &argv);
  gconf_init(argc, argv, NULL);

  client = gconf_client_new();

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);

  str = gconf_client_get_string(client, "/extra/test/directory/key",
                                NULL);

  label = gtk_label_new(str ? str : "<unset>");

  if (str)
    g_free(str);

  gtk_container_add(GTK_CONTAINER(window), label);

  gconf_client_add_dir(client,
                       "/extra/test/directory",
                       GCONF_CLIENT_PRELOAD_NONE,
                       NULL);

  gconf_client_notify_add(client, "/extra/test/directory/key",
                          key_changed_callback,
                          label,
                          NULL, NULL);
```

```
  gtk_widget_show_all(window);

  gtk_main();

  return 0;
}
```

A Simple Controller

This controller pops up a window with an entry box; if you type in the entry box and press return, the box contents will become the new value of the key `/extra/test/directory/key`. If you have the view program running, then its label should update to reflect the new value.

Again, see the GConf documentation for full details.

```
/* A very simple program that sets a single key value when you type
   it in an entry and press return */

#include <gconf/gconf-client.h>
#include <gtk/gtk.h>

void
entry_activated_callback(GtkWidget* entry, gpointer user_data)
{
  GConfClient* client;
  gchar* str;

  client = GCONF_CLIENT(user_data);

  str = gtk_editable_get_chars(GTK_EDITABLE(entry), 0, -1);

  gconf_client_set_string(client, "/extra/test/directory/key",
                          str, NULL);

  g_free(str);
}

int
main(int argc, char** argv)
{
  GtkWidget* window;
  GtkWidget* entry;
  GConfClient* client;

  gtk_init(&argc, &argv);
  gconf_init(argc, argv, NULL);

  window = gtk_window_new(GTK_WINDOW_TOPLEVEL);
  entry = gtk_entry_new();

  gtk_container_add(GTK_CONTAINER(window), entry);

  client = gconf_client_new();

  gconf_client_add_dir(client,
                       "/extra/test/directory",
```

```
                        GCONF_CLIENT_PRELOAD_NONE,
                        NULL);

  gtk_signal_connect(GTK_OBJECT(entry), "activate",
                     GTK_SIGNAL_FUNC(entry_activated_callback),
                     client);

  gtk_widget_show_all(window);

  gtk_main();

  return 0;
}
```

Appendix A. Other Resources

- GNOME Developer Site[1]
- GConf Documentation[2]

Notes

1. http://developer.gnome.org
2. http://developer.gnome.org/doc/API/gconf/index.html