Maemo Diablo Asynchronous GConf

Training Material

February 9, 2009

Contents

Chapter 1

Asynchronous GConf

1.1 Listening to changes in GConf

GConf as a central storage of configuration was presented before in the "maemo Application Development" material. We'll now see how to extend GConf to be more suitable in asynchronous work and especially when implementing services.

When you have simple configuration needs for your service, and want to support reacting to configuration changes in "real-time", you will want to use GConf. Also, people tend to use GConf when they're too lazy to write their own configuration file parsers (although there is a simple one in GLib) or too lazy to write the GUI part to change the settings. Our example program will simulate the first case and react to changes in a subset of GConf configuration name space when the changes happen.

Our application will be interested in two string values, one to set the device to use for communication (`connection`), and the other to set the communication parameters for the device (`connectionparams`). Since we'll be concentrating on just the change notifications, the program logic is simplified by omitting the proper setup code in the program. This means that you'll have to setup some values to the GConf keys prior to running the program. We'll be using the `gconftool-2` to do this, and have prepared a target in the Makefile just for this:

```
# Define a variable for this so that the GConf root may be changed
gconf_root := /apps/Maemo/platdev_ex

# ... Listing cut for brevity ...

# This will setup the keys into default values.
# It will first do a clear to remove any existing keys.
primekeys: clearkeys
  gconftool-2 --set --type string \
              $(gconf_root)/connection btcomm0
  gconftool-2 --set --type string \
              $(gconf_root)/connectionparams 9600,8,N,1

# Remove all application keys
clearkeys:
  @gconftool-2 --recursive-unset $(gconf_root)
```

```
# Dump all application keys
dumpkeys:
  @echo Keys under $(gconf_root):
  @gconftool-2 --recursive-list $(gconf_root)
```

Listing 1.1: Utility targets for setting up the GConf data for the application (gconf-listener/Makefile)

We prepare the keyspace next by running the `primekeys` target and verify that it succeeds by running the `dumpkeys` target:

```
[sbox-DIABLO_X86: ~/gconf-listener] > make primekeys
gconftool-2 --set --type string \
            /apps/Maemo/platdev_ex/connection btcomm0
gconftool-2 --set --type string \
            /apps/Maemo/platdev_ex/connectionparams 9600,8,N,1
[sbox-DIABLO_X86: ~/gconf-listener] > make dumpkeys
Keys under /apps/Maemo/platdev_ex:
 connectionparams = 9600,8,N,1
 connection = btcomm0
```

1.2 Implementing notifications on changes in GConf

We'll start with the necessary header information. The GConf namespace settings have been all implemented using `cpp` macros so that one can easily change the prefix of the name-space if required later on.

```
#include <glib.h>
#include <gconf/gconf-client.h>
#include <string.h> /* strcmp */

/* As per maemo Coding Style and Guidelines document, we use the
   /apps/Maemo/ -prefix.
   NOTE: There is no central registry (as of this moment) that you
         could check that your application name doesn't collide with
         other application names, so caution is advised! */
#define SERVICE_GCONF_ROOT "/apps/Maemo/platdev_ex"

/* We define the names of the keys symbolically so that we may change
   them later if necessary, and so that the GConf "root directory" for
   our application will be automatically prefixed to the paths. */
#define SERVICE_KEY_CONNECTION \
        SERVICE_GCONF_ROOT "/connection"
#define SERVICE_KEY_CONNECTIONPARAMS \
        SERVICE_GCONF_ROOT "/connectionparams"
```

Listing 1.2: The necessary headers and the name-space defines (gconf-listener/gconf-key-watch.c)

Our `main` starts innocently enough, by creating a GConf client object (that encapsulates the connection to the GConf daemon) and then displays the two values on output:

```
int main (int argc, char** argv) {
  /* Will hold reference to the GConfClient object. */
  GConfClient* client = NULL;
  /* Initialize this to NULL so that we'll know whether an error
     occurred or not (and we don't have an existing GError object
     anyway at this point). */
  GError* error = NULL;
  /* This will hold a reference to the mainloop object. */
  GMainLoop* mainloop = NULL;

  g_print(PROGNAME ":main Starting.\n");

  /* Must be called to initialize GType system. The API reference for
     gconf_client_get_default() insists.
     NOTE: Using gconf_init() is deprecated! */
  g_type_init();

  /* Create a new mainloop structure that we'll use. Use default
     context (NULL) and set the 'running' flag to FALSE. */
  mainloop = g_main_loop_new(NULL, FALSE);
  if (mainloop == NULL) {
    g_error(PROGNAME ": Failed to create mainloop!\n");
  }

  /* Create a new GConfClient object using the default settings. */
  client = gconf_client_get_default();
  if (client == NULL) {
    g_error(PROGNAME ": Failed to create GConfClient!\n");
  }

  g_print(PROGNAME ":main GType and GConfClient initialized.\n");

  /* Display the starting values for the two keys. */
  dispStringKey(client, SERVICE_KEY_CONNECTION);
  dispStringKey(client, SERVICE_KEY_CONNECTIONPARAMS);
```

Listing 1.3: Setting up the GConf connection (gconf-listener/gconf-key-watch.c)

The `dispStringKey` utility is rather simple as well, building on the GConf material that was covered in the "maemo Application Development" material:

```
/**
 * Utility to retrieve a string key and display it.
 * (Just as a small refresher on the API.)
 */
static void dispStringKey(GConfClient* client,
                          const gchar* keyname) {

  /* This will hold the string value of the key. It will be
     dynamically allocated for us, so we need to release it ourselves
     when done (before returning). */
  gchar* valueStr = NULL;

  /* We're not interested in the errors themselves (the last
     parameter), but the function will return NULL if there is one,
     so we just end in that case. */
  valueStr = gconf_client_get_string(client, keyname, NULL);

  if (valueStr == NULL) {
    g_error(PROGNAME ": No string value for %s. Quitting\n", keyname);
    /* Application terminates. */
  }

  g_print(PROGNAME ": Value for key '%s' is set to '%s'\n",
          keyname, valueStr);

  /* Normally one would want to use the value for something beyond
     just displaying it, but since this code doesn't, we release the
     allocated value string. */
  g_free(valueStr);
}
```

Listing 1.4: Utility to retrieve and display a string key value (gconf-listener/gconf-key-watch.c)

We next tell the GConf client to attach itself to a specific name-space part that we'll want to operate with:

```
  /**
   * Register directory to watch for changes. This will then tell
   * GConf to watch for changes in this namespace, and cause the
   * "value-changed"-signal to be emitted. We won't be using that
   * mechanism, but will opt to a more modern (and potentially more
   * scalable solution). The directory needs to be added to the
   * watch list in either case.
   *
   * When adding directories, you can sometimes optimize your program
   * performance by asking GConfClient to preload some (or all) keys
   * under a specific directory. This is done via the preload_type
   * parameter (we use GCONF_CLIENT_PRELOAD_NONE below). Since our
   * program will only listen for changes, we don't want to use extra
   * memory to keep the keys cached.
   *
   * Parameters:
   * - client: GConf-client object
   * - SERVICEPATH: the name of the GConf namespace to follow
   * - GCONF_CLIENT_PRELOAD_NONE: do not preload any of contents
   * - error: where to store the pointer to allocated GError on
   *          errors.
   */
  gconf_client_add_dir(client,
                       SERVICE_GCONF_ROOT,
                       GCONF_CLIENT_PRELOAD_NONE,
                       &error);

  if (error != NULL) {
    g_error(PROGNAME ": Failed to add a watch to GCClient: %s\n",
            error->message);
    /* Normally we'd also release the allocated GError, but since
       this program will terminate on g_error, we won't do that.
       Hence the next line is commented. */
    /* g_error_free(error); */

    /* When you want to release the error if it has been allocated,
       or just continue if not, use g_clear_error(&error); */
  }

  g_print(PROGNAME ":main Added " SERVICE_GCONF_ROOT ".\n");
```

Listing 1.5: Registering the interest in a part of the GConf name space (gconf-listener/gconf-key-watch.c)

Proceeding with the callback function registration, we have:

```
  /* Register our interest (in the form of a callback function) for
     any changes under the namespace that we just added.

     Parameters:
     - client: GConfClient object.
     - SERVICEPATH: namespace under which we can get notified for
```

```
                    changes.
      - gconf_notify_func: callback that will be called on changes.
      - NULL: user-data pointer (not used here).
      - NULL: function to call on user-data when notify is removed or
              GConfClient destroyed. NULL for none (since we don't
              have user-data anyway).
      - error: return location for an allocated GError.

     Returns:
     guint: an ID for this notification so that we could remove it
            later with gconf_client_notify_remove(). We're not going
            to use it so we don't store it anywhere. */
  gconf_client_notify_add(client, SERVICE_GCONF_ROOT,
                          keyChangeCallback, NULL, NULL, &error);
  if (error != NULL) {
    g_error(PROGNAME ": Failed to add register the callback: %s\n",
            error->message);
    /* Program terminates. */
  }

  g_print(PROGNAME ":main CB registered & starting main loop\n");
```

Listing 1.6: Registering the interest in a part of the GConf name space (gconf-listener/gconf-key-watch.c)

When dealing with regular desktop software, you could use multiple callback functions; one for each key to track. However, this would require you to implement multiple callback functions and this runs a risk of enlarging the size of your code. For this reason, the example code uses one callback function, which will internally multiplex between the two keys (by using `strcmp`):

```
/**
 * Callback called when a key in watched directory changes.
 * Prototype for the callback must be compatible with
 * GConfClientNotifyFunc (for ref).
 *
 * It will find out which key changed (using strcmp, since the same
 * callback is used to track both keys) and the display the new value
 * of the key.
 *
 * The changed key/value pair will be communicated in the entry
 * parameter. userData will be NULL (can be set on notify_add [in
 * main]). Normally the application state would be carried within the
 * userData parameter, so that this callback could then modify the
 * view based on the change. Since this program does not have a state,
 * there is little that we can do within the function (it will abort
 * the program on errors though).
 */
static void keyChangeCallback(GConfClient* client,
                              guint        cnxn_id,
                              GConfEntry*  entry,
                              gpointer     userData) {

  /* This will hold the pointer to the value. */
  const GConfValue* value = NULL;
  /* This will hold a pointer to the name of the key that changed. */
  const gchar* keyname = NULL;
  /* This will hold a dynamically allocated human-readable
     representation of the changed value. */
  gchar* strValue = NULL;
```

```c
  g_print(PROGNAME ": keyChangeCallback invoked.\n");

  /* Get a pointer to the key (this is not a copy). */
  keyname = gconf_entry_get_key(entry);

  /* It will be quite fatal if after change we cannot retrieve even
     the name for the gconf entry, so we error out here. */
  if (keyname == NULL) {
    g_error(PROGNAME ": Couldn't get the key name!\n");
    /* Application terminates. */
  }

  /* Get a pointer to the value from changed entry. */
  value = gconf_entry_get_value(entry);

  /* If we get a NULL as the value, it means that the value either has
     not been set, or is at default. As a precaution we assume that
     this cannot ever happen, and will abort if it does.
     NOTE: A real program should be more resilient in this case, but
           the problem is: what is the correct action in this case?
           This is not always simple to decide.
     NOTE: You can trip this assert with 'make primekeys', since that
           will first remove all the keys (which causes the CB to
           be invoked, and abort here). */
  g_assert(value != NULL);

  /* Check that it looks like a valid type for the value. */
  if (!GCONF_VALUE_TYPE_VALID(value->type)) {
    g_error(PROGNAME ": Invalid type for gconfvalue!\n");
  }

  /* Create a human readable representation of the value. Since this
     will be a new string created just for us, we'll need to be
     careful and free it later. */
  strValue = gconf_value_to_string(value);

  /* Print out a message (depending on which of the tracked keys
     change. */
  if (strcmp(keyname, SERVICE_KEY_CONNECTION) == 0) {
    g_print(PROGNAME ": Connection type setting changed: [%s]\n",
            strValue);
  } else if (strcmp(keyname, SERVICE_KEY_CONNECTIONPARAMS) == 0) {
    g_print(PROGNAME ": Connection params setting changed: [%s]\n",
            strValue);
  } else {
    g_print(PROGNAME ":Unknown key: %s (value: [%s])\n", keyname,
            strValue);
  }

  /* Free the string representation of the value. */
  g_free(strValue);

  g_print(PROGNAME ": keyChangeCallback done.\n");
}
```

Listing 1.7: Registering the interest in a part of the GConf name space (gconf-listener/gconf-key-watch.c)

The complications in the above code rise from the fact that GConf communicates values using a GValue structure, which can carry values of any simple type. Since we don't completely trust GConf (or the user for that matter) to return the correct type for the value, we need to be extra careful and not assume that it will always be a string. GConf also supports "default" values, which are communicated to the application using NULLs, so we need to guard against that as well. Especially since our application doesn't provide a schema for the configuration space which would contain the default values.

We'll next build and test the program. We'll start the program on the background so that we can use `gconftool-2` to see how the program will react to changing parameters:

```
[sbox-DIABLO_X86: ~/gconf-listener] > make
cc -I/usr/include/glib-2.0 -I/usr/lib/glib-2.0/include -I/usr/include/gconf/2 \
   -I/usr/include/dbus-1.0 -I/usr/lib/dbus-1.0/include -Wall -g \
   -DPROGNAME=\"gconf-key-watch\" gconf-key-watch.c -o gconf-key-watch \
   -lgconf-2 -ldbus-glib-1 -ldbus-1 -lgobject-2.0 -lglib-2.0
[sbox-DIABLO_X86: ~/gconf-listener] > run-standalone.sh ./gconf-key-watch &
[2] 21385
gconf-key-watch:main Starting.
gconf-key-watch:main GType and GConfClient initialized.
gconf-key-watch: Value for key '/apps/Maemo/platdev_ex/connection'
 is set to 'btcomm0'
gconf-key-watch: Value for key '/apps/Maemo/platdev_ex/connectionparams'
 is set to '9600,8,N,1'
gconf-key-watch:main Added /apps/Maemo/platdev_ex.
gconf-key-watch:main CB registered & starting main loop
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type string \
  /apps/Maemo/platdev_ex/connection ttyS0
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection type setting changed: [ttyS0]
gconf-key-watch: keyChangeCallback done.
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type string \
  /apps/Maemo/platdev_ex/connectionparams ''
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: []
gconf-key-watch: keyChangeCallback done.
```

The latter change is somewhat problematic (which your code needs to deal with as well). You need to decide how to react to values whose type is correct, but their values are not sensible. GConf in itself does not provide syntax checking for the values, nor any semantic checking support. It's recommended that you will only react to configuration changes that will pass some internal (to the application) logic that will check their validity, both at syntax level and also at semantic level.

One option would also be resetting the value back to a valid value whenever your program will detect an invalid value set attempt. This will lead to a lot of problems if the value is set programmatically from another program that will obey the same rule, so do not do it. Quitting your program on invalid values is also not an option that you should use, since the restricted environment doesn't provide many ways to inform the user that your program has quit.

An additional possible problem is having multiple keys that are all "related" to a single setting or action. You'll need to decide how to deal with changes across multiple GConf keys that are related, yet changed separately. The two key example code demonstrates the inherent problem: should the server re-initialise the (theoretic) connection when the `connection`-key is changed, or when the `connectionparams`-key is changed? If the connection will be re-initialized when either of the keys will change, then the connection will be re-initialized twice when both are changed "simultaneously" (user presses "Apply" on a settings dialog, or `gconftool-2` is run and sets both keys). You can see how this might be an even larger problem if instead of two keys, you would have 5 per connection. GConf (and the `GConfClient` GObject wrapper that we've been using) does not support "configuration set transactions", that would allow setting and processing multiple related keys using an atomic model. The example program ignores this issue completely.

We'll next test how the program (which is still running) will react to other problematic situations:

```
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type int \
  /apps/Maemo/platdev_ex/connectionparams 5
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: [5]
gconf-key-watch: keyChangeCallback done.
[sbox-DIABLO_X86: ~/gconf-listener] > gconftool-2 --set --type boolean \
  /apps/Maemo/platdev_ex/connectionparams true
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch: Connection params setting changed: [true]
gconf-key-watch: keyChangeCallback done.
```

Our application survives the wrong type of value

We'll next remove the configuration keys while the program is still running:

```
[sbox-DIABLO_X86: ~/gconf-listener] > make clearkeys
gconf-key-watch: keyChangeCallback invoked.
gconf-key-watch[21403]: GLIB ERROR ** default -
  file gconf-key-watch.c: line 129 (keyChangeCallback):
  assertion failed: (value != NULL)
aborting...
/usr/bin/run-standalone.sh: line 11: 21403 Aborted (core dumped) $@"
[1]+  Exit 134 run-standalone.sh ./gconf-key-watch
```

But doesn't survive the removal of the key

Since the code (in the callback function) contains an assert that checks for non-NULL values, it will abort when we remove the key and that causes the value to go to NULL. So the abortion in the above case is expected.