UniConf, GConf, KConfig, D-BUS, Elektra, oh my!
or DConf — a configuration framework for everyone

Simon Law
Net Integration Technologies
sfllaw@nit.ca

Patrick Patterson
Net Integration Technologies
ppatters@nit.ca

Abstract

We've been watching the discussions surrounding DConf with interest and amusement. Coincidentally, we've been talking amongst ourselves and have been saying, "wouldn't it be nice if there were just some way to glue all these different applications together?" And then we realize that we've been asking a rhetorical question. In this paper, we're going to recap what people want in a universal configuration system. Then we're going to show you the one we've built; because we're kinda sassy that way.

N.I.H.

We're going to start from the assertion that nearly every user application needs to store preferences. The traditional way of doing this in a POSIX environment is by storing them in dot-files. These are simple text-based configuration files, which look sort of like Figure 1.

```
xterm*eightBitInput: false
xterm*eightBitOutput: true
xterm*scrollTtyOutput: false
xterm*saveLines: 500
```

Figure 1: An .Xresources file

Throughout the software world, there are many different types of configuration systems. Some of them are based on simple line-delimited text files, others are complex hierarchical XML structures, and there are even those that are Turing-complete programming languages. Programmers seem to have a lot of fun inventing new types of configuration formats, as evidenced by the wide variety that we see. We must admit, this hobby is so much fun that we had to restrain ourselves from writing one of our own.

For the purposes of our discussion here, we're going to limit ourselves to the most common type of configuration system: one that stores unordered key-value pairs. As an example, we can look at GConf[1] which keeps its configuration in XML-files similar to the one shown in Figure 2.

It is all very nice when each application has its own configuration files. It feels like it's the master of its own domain, and that's a mighty splendid feeling. What happens, however, when

[1]http://www.gnome.org/projects/gconf/

```
<?xml version="1.0"?>
<gconf>
 <entry name="focus_mode"
        mtime="1089315055"
        type="string">
  <stringvalue>sloppy</stringvalue>
 </entry>
 <entry name="num_workspaces"
        mtime="1092539650"
        type="int" value="1">
 </entry>
</gconf>
```

Figure 2: A GConf file

you want certain applications to talk to each other? Or if sane global defaults ought to be provided? Then things start getting messy. You could write an application that knows how to divine information from another application's configuration files, much like Mozilla Firefox[2] does to Seamonkey[3]. Or if you've ever been a systems administrator, you can probably remember writing a script that migrates configuration data from one system to another. Or if you maintain a desktop environment, maybe you've helped to write a configuration system like KConfig[4].

All three of these options don't seem to be very appealing. Which is why people have converged on freedesktop.org in the hopes that a unified configuration system can be developed. This effort, currently called DConf, has been a difficult one. Not only is implementation a hurdle, but getting people to agree on a standard configuration format has been tough. After all, nobody wants to change software that already works — an understandable position to take.

[2]http://www.mozilla.org/products/firefox/
[3]http://www.mozilla.org/products/mozilla1.x/
[4]http://www.kde.org/

Wherefore art thou?

Before we start to consider a unified configuration system, we should probably ask ourselves, "why do we want one anyway?" After all, it's going to be a lot of work to implement one. Not only do you have to design one that's technically superior, you also have to convince the rest of the world to like it. This, my friends, is no easy task.

For starters, it would be very nice if there were one central interface for manipulating a user's configuration files. You, as a user, really ought to be perfectly oblivious as to how the configuration is stored, while still able to modify it to your heart's content.

Say you're a system or network administrator. Right now, changing the global configuration requires digging into the `/etc` directory to find the correct files to twiddle. And that's only for one machine. Woe to the admin who has a cluster of machines. Wouldn't it be great if all the settings were stored in one central place, accessible through one interface; rather than a dozen different places in a hundred different ways?

Perhaps you're a programmer who just wants to solve a very interesting problem, and doesn't want to write yet another configuration system. It sure would be nice if all the correct semantics were worked out in advance, readily available to use in a standard library. And it would be really nice if you could slowly adopt this library, instead of having to restructure existing programs.

Maybe you're a software distributor. Right now, updating configuration files politely requires writing a parser to understand the files. If only there were a standard interface to make the minimal change without having to write configuration parsers in shell script.

Shopping list

What kinds of properties would make a good configuration framework? Well, here's a simple list of desirable properties that we've come up with:[5]

- Stackable storage back-ends
- Standalone and dæmonized modes
- Low-latency network-transparent protocol
- Simple programming interface
- Permissions and access control
- Notifications of changes
- Transactions and rollbacks
- Global policy inheritance
- Free beer

All very desirable properties, and nicely enough, most of them are quite tractable.

Columns and beams

It just so happened that we were solving a problem that needed a configuration database. So we wrote one — and realized that we had done it wrong. We brushed ourselves off, picked ourselves back up, and wrote the UniConf system.

UniConf was designed to be very simple, so that anyone can understand it. It was also designed to be very powerful, so that anyone can extend it. Simplicity, elegance, and power are not easy to get right, so we had to think hard about our previous mistakes.

We should begin by describing the architecture of the UniConf system. Figure 3 shows how the various pieces fit together. The library provides many stackable back-end modules which we call *generators*.

Some of these generators store data directly, for instance the `temp:` generator stores data in core. Some generators are transports, the `unix:` generator talks over Unix domain sockets. These generators are stackable, which means that you can layer them on top of each other, like `cache:` which maintains a fast in-core cache of its underlying generator.

[5]With plenty of help from xdg@lists.freedesktop.org

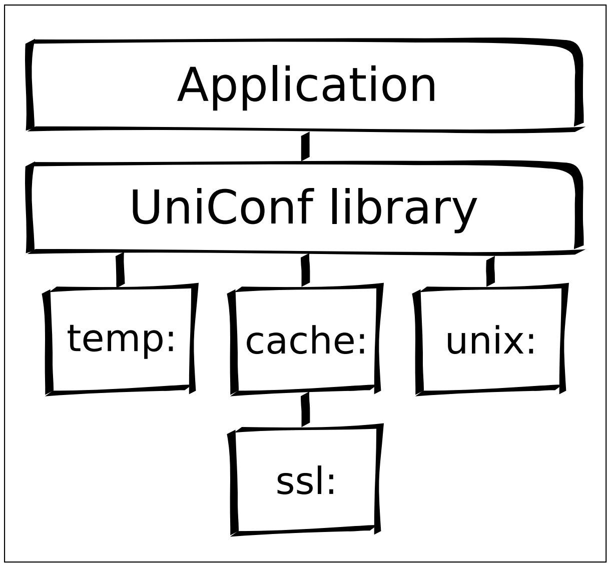

Figure 3: UniConf architecture

You'll notice that we mentioned Unix domain sockets. Well, there must be something listening on the other side, so we have written a dæmon that listens for incoming connections. You can see how a clients are connected to UniConf servers in Figure 4. Notice that although some of them are connected locally through D-BUS and Unix sockets, others can be connected remotely via SSL-over-TCP.

By having remote UniConf servers, one can easily create an architecture that distributes configuration data from a central master to end-user slave machines. This can be useful for a multinational enterprise that needs to propagate settings to each of its desktops. Or it can be useful to the end-user who wants her browser bookmarks to remain synchronised on her desktop and her laptop.

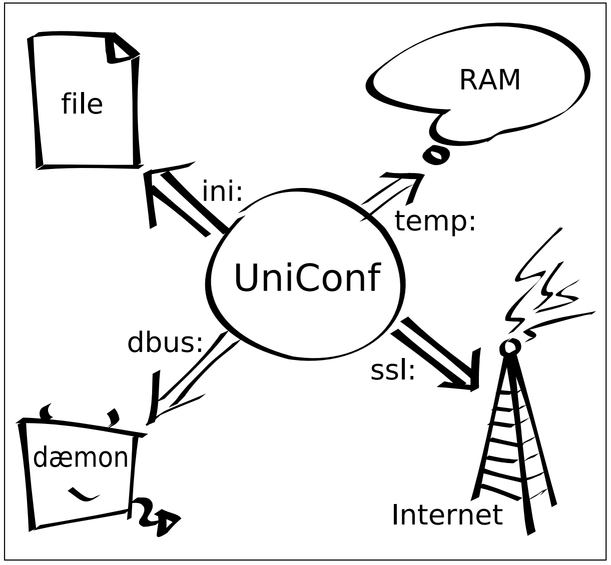

Figure 4: UniConf network

The choice to work standalone, locally with a server, or remotely with a server should remain with the user. To facilitate this, the generator stacks are not hard-coded in a UniConfiscated program. Instead, the location of its configuration is specified using a *moniker string*. To connect to a remote configuration, the moniker string to construct looks like

```
ssl:juin.nit.ca:4113
```

which tells UniConf to connect to juin.nit.ca on port 4113 using SSL-over-TCP.

Because the UniConf system is built upon stackable generators, the `ssl:` generator knows nothing about caching or reliability. So we have written other generators that do just that. The canonical way to connect to a remote UniConf source is to write

```
cache:retry:ssl:juin.nit.ca:4113
```

which means to cache reads and writes to the underlying generator; as well as reconnect to that generator, if it ever disappears.

Mapping the space

Now that we've gone over how to configure a UniConf system, what about getting at the data inside of one? Since almost all configuration systems store unordered key-value pairs, that's what we used as the domain for UniConf. Each key stored in the system must have one value associated with it. As well, each key can have zero or more child keys. We call this structure a *UniConf tree.* A delightful property that emerges is that any child of a key are also UniConf trees.

In order to address each key, we decided to adopt the same naming scheme as the Unix filesystem. Keys are forward-slash delimited and are relative to a particular UniConf tree. A partial UniConf tree for a POSIX name service would look somewhat like Figure 5.

```
users = {}
users/sfllaw = {}
users/sfllaw/uid = 1000
users/sfllaw/gid = 1000
users/sfllaw/homedir = /home/sfllaw
users/sfllaw/shell = /bin/bash
groups = {}
groups/sfllaw = {}
groups/sfllaw/gid = 1000
groups/sfllaw/users = {}
groups/sfllaw/users/sfllaw = 1
groups/sfllaw/users/ppatters = 1
```

Figure 5: UniConfiscated name service

Unlike the Unix filesystem, there are no absolute paths in UniConf. This makes UniConf closed under composition: any sub-tree is a first-class tree of its own. We don't provide hard links, to inhibit cycles from being created.

There are three operations that can be performed: `setting`, `geting`, and `iteration` over

UniConf keys. A `set` operation replaces the value of the key, creating it and its parents if necessary. A `get` operation returns the value of a UniConf key, returning a null value if it doesn't exist. An `iteration` operation provides a list of sub-keys that exist. With these operations, it is possible to programmaticly build and traverse a UniConf tree.

To control access to parts of the UniConf tree, we support Unix-style permissions on individual keys. The standard read, write, and execute bits apply as one would expect. These permissions are enforced by the `perm:` generator, which controls access to its underlying generator. Since we've implemented permission control in the form of a generator, if you need another schema for access control, a generator could be written to provide those semantics.

Ring ring!

For applications that only source their configuration files on start, this is sufficient functionality. But UniConf was built to glue together local and remote configuration systems, so we needed a low-latency method to propagate changes to the configuration tree, thereby maintaining consistency across the entire UniConf network. We decided to implement *notifications,* both in the generators and exposed to the application programming interface.

Your application can ask to be notified when a particular UniConf sub-tree changes by providing a callback function. When it does, your application will be notified by the callback, which will be provided with the key that was modified. The callback function should effect the change immediately or queue up the key if it needs more time. In this way, your application doesn't need to poll its configuration files for changes. Nor will the user need to press keys to refresh the configuration manually. And it certainly means that the user won't have to restart your application.

Keep in mind that your program can be completely ignorant of the notification system and it will work just as well. But using notifications will make it seem more responsive. For graphical applications, instant reflection of configuration changes is very intuitive. And for dæmons, you no longer need to implement `SIGHUP` handlers in order to get reasonable behaviour.

Atomic generation

UniConf is designed to work asynchronously. By that, I mean that after `set`ting a value, you may not see the same value when `get`ting it immediately afterwards. To guarantee synchronicity, UniConf would be very slow, since the data would have to propogate throughout the entire network before a `set` could be committed.

Instead, you can rely on generators to provide a sane view of the underlying UniConf tree. For instance, using the `transaction:` generator makes UniConf appear synchronous. That is, after performing a `set`, a `get` will return the previously `set` value.

It's called the `transaction:` generator for a good reason, though. It provides database-like atomic transactions for the UniConf tree that it wraps. Your application can `set` several values, and then decide to commit them or rollback them. We find rollbacks very useful for error-handling. For back-ends that don't support atomic commits, the generator tries its best to `set` values all at once, so you should treat atomic commits as an optimization, as opposed to a guarantee.

Shaking hands

The UniConf network protocol is defined as the set of operations that are network-based client generators talk when they connect to a UniConf server. It's designed as an asynchronous pipelined protocol, so there's very little latency. There are five basic operations:

`set` sends a key-value pair to the remote server.

`get` retrieves the value associated with a key, from the remote server.

`subt` retrieves a list of sub-keys that are children of a key. This is used to implement `iteration`.

`commit` commits changes to storage.

`refresh` refresh contents from storage.

With these five, you can create a general configuration system that stores key-value pairs. We also have three operations that are performance optimizations, and do not need to be supported by clients:

`del` deletes the key-value pair, and all their children.

`hchild` queries the remote server for whether a key has any children.

`setv` sets multiple key-value pairs.

Now the server can respond to these operations with one of the following:

`OK` The operation succeeded.

`FAIL` The operation failed.

`UNKNOWN` The operation is unknown. This must be returned by servers when they receive a command they don't understand.

`ONEVAL` A reply to a `get`.

`PART` A partial reply, used with `subt`.

`CHILD` A boolean reply to an `hchild` query.

Finally, the server has two events it can send to a client. The client is not expected to respond to them:

`HELLO` On connection, the server sends the protocol version.

`NOTICE` Notification that a key-value pair has changed.

We provide these operators in our `unix:`, `udp:`, `tcp:`, `ssl:`, and `dbus:` transports. As such, you don't actually need to link in the UniConf library at all, as long as your application is able to understand the client side of the protocol.

Sales pitch

If you're writing a new application, we exhort you to consider UniConf as your native configuration system. Just look at the advantages: you don't have to implement it from scratch, you get a tree of key-value configuration items which you can access from nearly everywhere. It provides advanced features like notifications and transactions. Not only that, you're not tied in to any particular format because UniConf is extensible.

For maintainers of existing software, migrating to UniConf is not particularly painless. It's easy if you're already using a configuration library like GConf, there's a UniConf generator that communicates with GConf already. If you're already using KConfig, there's a generator for that as well.

If you've got your own configuration format, however, migrating is still rather simple. You could decide to use the UniConf library natively, and pick up various features as you need them. Or you could write a UniConf generator that understands your configuration system, and interact with a UniConf network that way. Even a hybrid approach is attractive, to transition your users from one system to the other.

Finally, if you have a lightweight application that can't afford to link in large libraries, it can choose to participate in a UniConf network by means of Elektra, or by speaking the D-BUS protocol.

In short, there are few technical reasons that would prevent application developers from adopting UniConf as the unified configuration system. We hope that this simple, well-designed, extensible library will be widely accepted. Mostly because we want people to stop worrying about simple things like configuration files, so that they can worry about difficult things like writing usable desktop. Or arguing about a universal configuration schema.

Acknowledgments

We'd like to thank all the people who have hacked on UniConf to make it the useful piece of software it is today.

To be specific, thanks go to Jeff Brown, James deBoer, William Lachance, Andrew MacPherson, Joe Mason, Avery Pennarun, Pierre Phaneuf, Tristan Schmelcher, Dan Taylor, and Peter Zion.