# Stephen's Work Ramblings

Just another blog.inf.ed.ac.uk weblog

- Home
- About

## LCFG Core: resource types

November 21, 2017

The recent round of LCFG client testing using real LCFG profiles from both Informatics and the wider community has shown that the code is now in very good shape and we're close to being able to deploy to a larger group of machines. One issue that this testing has uncovered is related to how the *type* of a resource is specified in a schema. A *type* in the LCFG world really just controls what regular expression is used to validate the resource value. Various type annotations can be used (e.g. `%integer`, `%boolean` or `%string`) to limit the permitted values, if there is no annotation it is assumed to be a tag list and this has clearly caught out a few component authors. For example:

```
@foo %integer
foo

@bar %boolean
bar

@baz
baz

@quux sub1_$ sub2_$
quux
sub1_$
sub2_$
```

Both of the last two examples (`baz` and `quux`) are tag lists, the first just does not have any associated sub-resources.

The compiler should not allow anything but valid tag names (which match `/^[a-zA-Z0-9_]+$/`) in a tag list resource but due to some inadequacies it currently permits pretty much anything. The new core code is a lot stricter and thus the v4 client will refuse to accept a profile if it contains invalid tag lists. Bugs have been filed against a few components (bug#1016 and bug#1017). It's very satisfying to see the new code helping us improve the quality of our configurations.

Comments Off on LCFG Core: resource types | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

## LCFG Core: Resource import and export

November 7, 2017

As part of porting the LCFG client to the new core libraries the qxprof and sxprof utilities have been updated. This has led to the development of a new high-level `LCFG::Client::Resources` Perl library which can be used to import, merge and export resources in all the various required forms. The intention is that eventually all code which uses the `LCFG::Resources` Perl library (in particular the `LCFG::Component` framework) will be updated to use this new library. The new library provides a very similar set of functionality and will appear familiar but I've taken the opportunity to improve some of the more awkward parts. Here's a simple example taken from the perldoc:

```
# Load client resources from DB
my $res1 = LCFG::Client::Resources::LoadProfile("mynode","client");

# Import client resources from environment variables
my $res2 = LCFG::Client::Resources::Import("client");

# Merge two sets of resources
my $res3 = LCFG::Client::Resources::Merge( $res1, $res2 );

# Save the result as a status file
LCFG::Client::Resources::SaveState( "client", $res3 );
```

The library can import resources from: Berkeley DB, status files, override files, shell environment and explicit resource specification strings. It can export resources as status files, in a form that can be evaluated in the shell environment and also in various terse and verbose forms (e.g. the output styles for qxprof).

The `LCFG::Resources` library provides access to resources via a reference to a hash which is structured something like:

```
{
   'sysinfo' => {
                 'os_id_full' => {
                                  'DERIVE' => '/var/lcfg/conf/server/releases/develop/core/include/lcfg/defaults/sysinfo.h:42',
                                  'VALUE' => 'sl74',
                                  'TYPE' => undef,
                                  'CONTEXT' => undef
                                 },
                 'path_lcfgconf' => {
                                  'DERIVE' => '/var/lcfg/conf/server/releases/develop/core/include/lcfg/defaults/sysinfo.h:100',
                                  'VALUE' => '/var/lcfg/conf',
                                  'TYPE' => undef,
                                  'CONTEXT' => undef
                                 },
                }
}
```

The top level key is the component name, the second level is the resource name and the third level is the name of the resource attribute (e.g. `VALUE` or `TYPE` ).

The new `LCFG::Client::Resources` library takes a similar approach with the top level key being the component name but the value for that key is a reference to a `LCFG::Profile::Component` object. Resource objects can then be accessed by using the `find_resource` method which returns a reference to a `LCFG::Resource` object. For example:

```
my $res = LCFG::Client::Resources::LoadProfile("mynode","sysinfo");

my $sysinfo = $res->{sysinfo};

my $os_id_full = $sysinfo->find_resource('os_id_full');

say $os_id_full->value;
```

Users of the `qxprof` and `sxprof` utilities should not notice any differences but hopefully the changes will be appreciated by those developing new code.

Comments Off on LCFG Core: Resource import and export | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

---

## Testing the new LCFG core : Part 2

May 18, 2017

Following on from the basic tests for the new XML parser the next step is to check if the new core libs can be used to correctly store the profile state into a Berkeley DB file. This process is particularly interesting because it involves evaluating any context information and selecting the correct resource values based on the contexts. Effectively the XML profile represents all possible configuration states whereas only a single state is stored in the DB.

The aim was to compare the contents of the *old* and *new* DBs for each Informatics LCFG profile. Firstly I used rdxprof to generate DB files using the current libs:

```
cd /disk/scratch/profiles/inf.ed.ac.uk/
for i in $(find -maxdepth 1 -type d -printf '%f\n' | grep -v '^\.');\
do \
 echo $i; \
 /usr/sbin/rdxprof  -v -u file:///disk/scratch/profiles/ $i; \
done
```

This creates a DB file for each profile in the `/var/lcfg/conf/profile/dbm` directory. For 1500-ish profiles this takes a **long** time…

The next step is to do the same with the new libs:

```
find /disk/scratch/profiles/ -name '*.xml' | xargs \
perl -MLCFG::Profile -wE \
'for (@ARGV) { eval { $p = LCFG::Profile->new_from_xml($_); \
$n = $p->nodename; \
$p->to_bdb( "/disk/scratch/results/dbm/$n.DB2.db" ) }; \
print $@ if $@ }'
```

This creates a DB file for each profile in the `/disk/scratch/results/dbm` directory. This is much faster than using rdxprof.

The final step was to compare each DB. This was done simply using the perl `DB_File` module to tie each DB to a hash and then comparing the keys and values. Pleasingly this has shown that the new code is generating identical DBs for all the Informatics profiles.

Now I need to hack this together into a test script which other sites can use to similarly verify the code on their sets of profiles.

Comments Off on Testing the new LCFG core : Part 2 | Uncategorized | Tagged: lcfg-client | Permalink
Posted by squinney

---

## Testing the new LCFG core : Part 1

May 17, 2017

The project to rework the core LCFG code is rattling along and has reached the point where some full scale testing is needed. The first step is to check whether the new XML parser can actually just parse all of our LCFG profiles. At this stage I'm not interested in whether it can do anything useful with the data once loaded, I just want to see how it handles a large number of different profiles.

Firstly a source of XML profiles is needed, I grabbed a complete local copy from our lcfg server:

```
rsync -av -e ssh lcfg:/var/lcfg/conf/server/web/profiles/ /disk/scratch/profiles/
```

I then ran the XML parser on every profile I could find:

```
find /disk/scratch/profiles/ -name '*.xml' | xargs \
perl -MLCFG::Profile -wE \
'for (@ARGV) { eval { LCFG::Profile->new_from_xml($_) }; print $@ if $@ }'
```

Initially I hit upon bug#971 which is a genuine bug in the schema for the gridengine component. As noted previously, this was found because the new libraries are much stricter about what is considered to be valid data. With that bug resolved I can now parse all 1525 LCFG XML profiles for Informatics.

Comments Off on Testing the new LCFG core : Part 1 | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

---

## LCFG Core Project

May 2, 2017

Over the last few years I have been working on (and off) creating a new set of "core" libraries for LCFG. This is now finally edging towards the point of completion with most of the remaining work being related to polishing, testing and documentation.

This project originated from the need to remove dependencies on obsolete Perl XML libraries. The other main aims were to create a new OO API for resources/components and packages which would provide new opportunities for code reuse between client, ngeneric and server.

Over time several other aims have been added:

- Simplify platform upgrades.
- Platform independence / portability.
- Make it possible to support new languages.
- Ensure resource usage remains low.

Originally this was to be a rewrite just in Perl but the heavy resource usage of early prototypes showed it was necessary to move at least some of the functionality into C libraries. Since that point the chance to enhance portability was also identified and included in the aims for the project. As well as making it possible to target other platforms (other Linux or Unix, e.g. MacOSX), the enhanced portability should make it much simpler and quicker to port to new Redhat based platforms.

The intention is that the new core libraries will be totally platform-independent and portable, for example, no hardwired paths or assumptions that platform is Redhat/RPM (or even Linux) based. The new core is split two parts: C and Perl libraries with the aim that as much functionality as possible is in the C libraries to aid reuse from other languages (e.g. Python).

The aim is that these libraries should be able to co-exist alongside current libraries to ease the transition.

I have spent a **lot** of time on documenting the entire C API. The documentation is formatted into html and pdf using doxygen, I had not used this tool before but I am very pleased with the results and will definitely be using it more in the future. Although a slow task, documenting the functions has proved to be a very useful review process. It has helped me find many inconsistencies between functions with similar purposes and has led to numerous small improvements.

## LCFG Client

The client has been reworked to use new Core libraries. This is where the platform-specific knowledge of paths, package manager, etc, is held.

## Resource Support

| Format | Read | Write |
|---|---|---|
| **XML** | YES | NO |
| **DB** | YES | YES |
| **Status** | YES | YES |
| **Environment** | YES | YES |

There is currently no support for reading header files or source profiles but this could be added later.

There is new support for finding the "diffs" between resources, components and profiles.

## Package Support

| Format | Read | Write |
|---|---|---|
| **XML** | YES | YES |
| **rpmcfg** | YES | YES |
| **rpmlist** | YES | YES |

There is currently no support for reading package list files but this could be added later.

## Remaining Work

There is still work to be done on the top-level profile handling code and the code for finding the differences between resources, components and profiles needs reworking. Also the libraries for reading/writing XML files and Berkeley DB need documentation.

That is all the remaining work required on the "core" libraries. After that there will be some work to do on finishing the port of the client to the new libraries. I've had that working before but function APIs have changed, I don't expect it to require a huge amount of work.

Comments Off on LCFG Core Project | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

---

## LCFG Client: Hasn't died yet…

August 2, 2016

Coming back from holiday I was pleased to see that I have a v4 client instance which has now been running continuously for nearly 3 weeks without crashing. It hasn't done a massive amount in that time but it has correctly applied some updates to both resources and packages.

In the time I've not been on holiday I've been working hard on documenting the code. For the C code I've chosen to use doxygen, it does a nice job of summarizing all the functions in each library and it makes it very simple to write the documentation using simple markup right next to the code for each function. I've also been working through some of the Perl modules and adding POD where necessary. It might soon be at the stage where others can pick it up and use it without needing to consult me for the details…

Comments Off on LCFG Client: Hasn't died yet… | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

---

## LCFG Client: It lives!

July 15, 2016

Cue forked lightning and crashes of thunder…

After much effort I finally have the first functional installation of the v4 LCFG client. This sees all the XML parsing and profile handling moved over to the new `LCFG::Profile` Perl modules which are wrappers around the new lcfg-core suite of libraries. There is still a bit of work required to properly handle LCFG contexts but otherwise it can handle everything we need. There are probably lots of small bugs to be resolved, there is also an almost total lack of documentation and the tests needs lot of attention but hey, at least it runs!

Comments Off on LCFG Client: It lives! | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

## LCFG Profile – Secure mode

May 19, 2016

The LCFG client has a, slightly weird, feature called "*secure mode*". This makes the client hold off applying any resource changes until they have been manually reviewed. The manual checking is done by examining the contents of a "hold file" which shows the differences in values for each modified resource in a simple text form. The file also contains a "signature" which is the MD5 digest (in hex) of the changes. A change set is applied manually by passing that signature to the client which then regenerates the hold file and compares that signature with the one supplied. This is not a heavily used feature of the client but it is something we want to support in the new LCFG profile framework. The new framework has built-in support for diffing the data structures which represent LCFG profiles, components and resources. This makes it relatively straightforward to add a feature which generates the secure-mode hold file when required, the only awkward part was finding some code to do the MD5 digest in a nice way.

Here's an example using the C API, error checking and suchlike has been dropped to keep it simple.

```
#include <lcfg/profile.h>
#include <lcfg/bdb.h>
#include <lcfg/differences.h>

int main(void) {

char * msg = NULL;

LCFGProfile * p1 = NULL;
lcfgprofile_from_status_dir( "/run/lcfg/status",
&p1, NULL, &msg );

LCFGProfile * p2 = NULL;
lcfgprofile_from_bdb( "/var/lcfg/conf/profile/dbm/example.lcfg.org.DB2.db",
&p2, NULL, 0, &msg );

LCFGDiffProfile * diff = NULL;
lcfgprofile_diff( p1, p2, &diff, &msg );

char * signature = NULL;
lcfgdiffprofile_to_holdfile( diff, "/tmp/holdfile", &signature, &msg );

lcfgprofile_destroy(p1);
lcfgprofile_destroy(p2);
lcfgdiffprofile_destroy(diff);

free(msg);

return 0;
}
```

Comments Off on LCFG Profile – Secure mode | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

## LCFG profile querying

May 13, 2016

The new LCFG profile framework makes it simple to retrieve component and resource information from profiles stored in the various standard formats (XML, Berkeley DB and status files).

Loading a profile from XML, DB or status directory:

```
my $p = LCFG::Profile->new_from_xml("example.xml");
```

my $p = LCFG::Profile->new_from_bdb(“example.db”);

my $p = LCFG::Profile->new_from_status_dir(“/run/lcfg/status”);

Loading a component from a DB or status file:

my $c = LCFG::Profile::Component->new_from_bdb( “example.bdb”, “client” );

my $c = LCFG::Profile::Component->new_from_statusfile( “/run/lcfg/status/client” );

Retrieving a component (e.g. client) from the profile:

my $c = $p->find_component(“client”);

Retrieving a resource (e.g. client.components) from a component:

my $r = $c->find_resource(“components”);

Getting the resource value:

say $r->value;

For convenience, if the resource is a tag list then you can get the value as a perl list:

@comps = $r->value;
for my $comp (@comps) {
…
}

Comments Off on LCFG profile querying | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

---

## LCFG profile handling

May 13, 2016

Over the last few months the new libraries for handling LCFG profiles have been shaping up nicely. They are finally reaching a point where they match up with my original aims so I thought I’d give a taste of how it all works. Here’s an example of processing an LCFG XML profile into the Berkeley DB and rpmcfg files required by the client:

```
use LCFG::Profile;

my $xml    = '/var/lcfg/conf/profile/xml/example.lcfg.org.xml';
my $dbm    = '/tmp/example.lcfg.org.DB2.db';
my $dbm_ns = 'example';
my $rpmcfg = '/tmp/example.lcfg.org.rpmcfg';

my $new_profile = LCFG::Profile->new_from_xml($xml);

my $update_dbm = 0;
if ( -f $dbm ) {
    my $cur_profile = LCFG::Profile->new_from_bdb($dbm);

    my $diff = $cur_profile->diff($new_profile);

    if ( $diff->size > 0 ) {
        $update_dbm = 1;
    }
} else {
    $update_dbm = 1;
}

if ( $update_dbm ) {
    $new_profile->to_bdb( $dbm, $dbm_ns );
    say 'Updated DBM';
}

my $pkgs_changed = $new_profile->to_rpmcfg($rpmcfg);
if ( $pkgs_changed ) {
    say 'Updated packages';
}
```

This is basically what the LCFG client does whenever it processes a new profile but is a lot nicer than the current rdxprof code!

Comments Off on LCFG profile handling | Uncategorized | Tagged: lcfg, lcfg-client | Permalink
Posted by squinney

---

## LCFG XML Profile changes

August 20, 2015

As part of the LCFG v4 client project I am working on converting the XML profile parsing over to using the libxml2 library. Recent testing has revealed a number of shortcomings in the way the LCFG XML profiles are generated which break parsers which are stricter than the old W3C code upon which the current client is based. In particular the encoding of entities has always been done in a style which is more suitable for HTML than XML. There is really only a small set of characters that must be encoded for XML, those are: single-quote, double-quote, left-angle-bracket, right-angle-bracket and ampersand (in some contexts the set can be even smaller). The new XML parser was barfing on unknown named entities which would be supported by a typical web browser. It is possible to educate an XML parser about these entities but it’s not really necessary. A better solution is to emit XML which is utf-8 compliant which avoids the needs for additional encoding. Alongside this problem of encoding more than was necessary the server was not encoding significant whitespace, e.g. newlines, carriage returns and tabs. By default a standards compliant XML parser will ignore such whitespace. An LCFG resource might well contain such whitespace so it was necessary to add encoding support to the server. In the process of making these changes to the LCFG::Server::Profile::XML module I merged all the calls to the encoder into a call to a single new `EncodeData` subroutine so that it is now trivial to tweak the encoding as required. These changes will be going out in version 3.3.0 of the LCFG-Compiler package in the next stable release. As always, please let us know if these changes break anything.

Comments Off on LCFG XML Profile changes | Uncategorized | Tagged: lcfg-client | Permalink
Posted by squinney

---

## MooX::HandlesVia and roles

August 20, 2015

I’ve been using Moo for Perl object-oriented programming for a while now. It’s really quite nice, it certainly does everything I need and it’s much lighter than Moose.

Whilst working on the LCFG v4 client project I recently came across a problem with the MooX::HandlesVia module when used in conjunction with roles. I thought it worth blogging about if only to save some other pour soul from a lot of head scratching (probably me in 6 months time).

If a class is composed of more than one role and each role uses the MooX::HandlesVia module, for example:

```
{
    package SJQ::Role::Foo;
    use Moo::Role;
    use MooX::HandlesVia;
}

{
    package SJQ::Role::Bar;
    use Moo::Role;
    use MooX::HandlesVia;
}

{
    package SJQ::Baz;
    use Moo;

    with 'SJQ::Role::Foo','SJQ::Role::Bar';

    use namespace::clean;
}

my $test = SJQ::Baz->new();
```

It fails and the following error message is generated:

```
Due to a method name conflict between roles 'SJQ::Role::Bar and
SJQ::Role::Foo', the method 'has' must be implemented by 'SJQ::Baz'
at /usr/share/perl5/vendor_perl/Role/Tiny.pm line 215.
```

It appears that MooX::HandlesVia provides its own replacement `has` method and this causes a problem when namespace::clean is also used.

The solution is to apply the roles separately, it’s perfectly allowable to call the `with` method several times. For example:

```
{
    package SJQ::Baz;
    use Moo;

    with 'SJQ::Role::Foo';
    with 'SJQ::Role::Bar';

    use namespace::clean;
}
```

Comments Off on MooX::HandlesVia and roles | Uncategorized | Tagged: lcfg-client | Permalink
Posted by squinney

---

## LCFG Client Guide

March 31, 2014

As part of my work on updating the LCFG client I've written a guide to the inner workings of the LCFG client. This is intended to be fairly high-level so it doesn't go into the details of which subroutine calls which subroutine. The aim is that this should cover all the main functionality and provide the information necessary to get started with altering and extending the client code base.

Comments Off on LCFG Client Guide | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## LCFG Client Refactor: New profile parser

January 24, 2014

Recently I've been working on developing a new framework which encapsulates all aspects of handling the LCFG profiles on the client-side. This framework is written in Perl and is named, appropriately enough, `LCFG::Profile`, I plan to blog about the various details in due course. The coding phase is almost complete and I've moved onto adding documentation for all the module APIs. I've found the documentation phase to be a very useful review process. It has helped me spot various dark corners of the code and methods which were added earlier in the development process which are no longer required. Removing this dead code now is a good idea as we may otherwise end up being committed to supporting the code if it forms part of a public API. I've also found it to be a very good way to spot inconsistencies between similar APIs implemented in the various modules. It's definitely a good idea to follow the principle of least surprise whenever possible. If methods are named similarly and take a similar group of arguments they probably ought to return similar types of results.

Comments Off on LCFG Client Refactor: New profile parser | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## LCFG Client Refactor: Phase Two

December 5, 2013

As the results of Phase One of the LCFG Client refactoring project are now in the beta-testing stage and approaching a roll-out date we have commenced work on Phase Two. The primary aim of this new work is to remove all dependencies on the `W3C::SAX` Perl modules which have been unmaintained for a very long time. We're probably the last place in the world still using those modules so it's definitely time to be moving on to something more modern. The project plan for this new work is available for anyone interested.

As a first step I've been prototyping some new XML parsing code based on the popular and well-maintained `XML::LibXML` module. I've also been thinking about ideas for an API for storing/accessing the information regarding components and resources. I've put together some useful notes on the LCFG XML profile structure to help me get my head around it all.

Comments Off on LCFG Client Refactor: Phase Two | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## LCFG V3 Client – beta release

October 17, 2013

I am pleased to announce that the v3 update for the LCFG client has now reached the beta-release stage. As of stable release 2013101401a everything is in place to begin testing at your ownsite. Full details are available on the LCFG wiki.

If you come across any bugs or unexpected behaviour please file a bug at bugs.lcfg.org.

Comments Off on LCFG V3 Client – beta release | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: Further node name support

June 3, 2013

I remember once as a 12 year old playing rugby at school. I received the ball, saw the field ahead was clear and knew that this was the time to run like hell. For one joyous moment I was brushing aside the defending team, spotting my moment of glory, having never been a particularly sporty kid was this my chance to join the cool crowd? Sadly, someone burst my bubble and pointed out that the main reason I wasn't being flattened was because we were actually playing *touch* rugby…

Anyway, my general point is, it's always good to know when, having been passed the ball, you should just run like hell and see what happens. It might also be good to remember which game you are playing but, hey, ho…

Having been given the chance to split the LCFG node name from the host name, I spotted a chance to really make it count. In short order the following code has been altered to extend this support to the whole of the LCFG client framework:

- perl-LCFG-Utils 1.5.0
- lcfg-ngeneric 1.4.0
- lcfg-om 0.8.0
- lcfg-file 1.2.0
- lcfg-authorize 1.1.0
- lcfg-hackparts 0.103.0
- lcfg-logserver 1.4.0
- lcfg-sysinfo 1.3.0
- lcfg-installroot 0.103.0

None of this has (yet) been shipped to the stable tree since it needs more hacking of the current LCFG client (v2) code to fix a compatibility issue.

The big achievement here is that it makes it possible to specify the lcfg nodename on the PXE installer kernel command-line via the `lcfg.node` parameter and get the whole way through to an installed managed machine which is using a LCFG profile which is completely unrelated to the host name.

There are various big benefits to this change. It is now possible to have a fully roaming machine which is LCFG managed, there is no requirement for a static host name or static IP address. This means that no matter what host name or domain name settings are in place the LCFG client will continue to work as required. This also makes it possible to use a single "generic" profile to configure multiple machines. If you know you have a lab full of identical machines this could be very handy indeed.

The downside of this is that some things like spanning maps will not work the way you might expect. You also will not receive notifications from the server when a profile changes, you have to really solely on the poll time (probably worth making the timeout shorter). You probably also cannot send acknowledgements to the server and the LCFG status pages will consequently be mostly useless for those clients. It is also difficult to configure networking to do anything other than use DHCP. You're choosing to move some of the configuration information back out of LCFG (or at least out of a particular profile). You may end up saving effort one way and adding it in another.

At the moment although I have broken the conceptual link between node and host name for the client framework there are still lots of components which are confused by this change. Components have traditionally been able to rely on combining the `profile.node` and `profile.domain` resources to form the FQDN. This was probably always on slightly shaky ground but now there can be no guarantee whatsoever of a useful value in the `profile.node` resource. If a component really cares about the host name (rather than the node name) then it will have to ask the host directly (using `hostname` or `Sys::Hostname` from Perl).

Comments Off on LCFG Client Refactor: Further node name support | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: host name versus node name

May 23, 2013

A long-standing issue that we have had with the LCFG client is that it is not possible to use an LCFG profile with a name which does not match the host name. They have always been treated by rdxprof and the ngeneric framework as conceptually interchangeable. There is no particular reason for this limitation other than the traditional "*it's always been that way*", also we've never had a requirement important enough to get this implemented or the opportunity to quickly make the change. As the refactoring project is drawing to a close it seemed like a good time to break this conceptual connection and rework the code to always use the LCFG node name. For the moment the actual behaviour won't change, since the node name defaults to the host name as before, but we now have a mechanism to allow it to be altered. When the client enters daemon mode it now stashes the name of the LCFG node being used. Since you can only run one client daemon at a time this makes reasonable sense. The standalone one-shot behaviour remains unaltered, you can still access any profile you like.

Comments Off on LCFG Client Refactor: host name versus node name | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactoring: starting the daemon

May 21, 2013

Following on from my previous work on fixing the way in which the UDP socket is opened for receiving notification messages I have been looking at why the LCFG component just hangs when the rdxprof process fails to daemonise.

It turns out that the LCFG client component uses an obscure ngeneric feature of the `Start` function which is that the final step is to call a `StartWait` function if it has been defined. In the client component this `StartWait` function sits waiting forever for a client context change even when the rdxprof process failed to start...

I think the problem comes from an expectation that the call to the `Daemon` function, which starts and backgrounds the rdxprof process, will fail if rdxprof fails to run. It does not fail (`$?` is zero) and the PID of the rdxprof process is always accessible through the `$!` variable, even if it was very short-lived.

There is, thankfully, a very simple solution here. The client component already has a `IsProcessRunning` function which can be used to check if the process associated with a PID is still active. This has to be used carefully, I have put a short sleep after the daemonisation stage to ensure that the process is fully started before doing the check. The check is also fairly naive so there is the slight risk that if the system is under resource pressure the rdxprof process could fail and then the PID could be immediately reused. For now I think it's reasonable to just accept the risks attached and revisit the issue later if it causes us problems. Associated with this, clearly the `StartWait` function really ought to eventually give up.

Comments Off on LCFG Client Refactoring: starting the daemon | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: notification handling

May 21, 2013

One long-standing issue with running the LCFG client (`rdxprof`) in daemon mode has been that if another process has already acquired the UDP socket which it wants (port 732) then it does not fail at startup but just hangs. This is clearly rather undesirable behaviour as it leaves the machine in an unmanageable state but because the client process **appears** to be running it's difficult to notice that anything is actually wrong.

Yesterday I spent a while looking at this problem. I reduced it to the most simple case of a script with a while-loop listening for messages on a UDP socket and then printing the messages to the screen. Running multiple processes at the same time revealed that there was nothing preventing multiple binds on the socket. I eventually discovered that this is caused by the `SO_REUSEADDR` option being set on the socket.

When using TCP setting this option is often necessary. It allows a process to reacquire access to a socket when it restarts. Sometimes processes may restart too quickly and the socket would otherwise not be ready. For UDP this option is only necessary if you want to listen on a broadcast or multicast address and have multiple listeners on the same machine, that's a fairly unusual scenario.

Disabling the `SO_REUSEADDR` option does exactly what we want. Attempting to run two rdxprof processes now results in it exiting with status 1 and this message:

```
[FAIL] client: can't bind UDP socket
[FAIL] client: Address already in use
```

There is a further problem with the LCFG client component not returning control to the caller when it fails to start rdxprof and I will have to do some further investigations into that problem.

Comments Off on LCFG Client Refactor: notification handling | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: splitting the project

May 14, 2013

I've recently been working on splitting the LCFG client code base from the LCFG component which is used to configure and manage the daemon. This allows the client Perl modules to be built in the style of a standard Perl module. The immediate benefit of this is the enhanced portability, it makes it much easier to build the code on platforms other than DICE Linux if you can use standard Perl module building tools. We could also upload the code to CPAN which would make it even easier to download and install.

There are also benefits for maintainability, the standard Perl build tools make it easy to run unit tests and do other tasks such as checking test and API documentation coverage for your code. It is not impossible to do these things without some tool like `Module::Build` but it is a lot more awkward. Also, without the standard tools you have to know, or be able to discover, where certain files should be installed, we have some of this built into the LCFG build tools CMake framework but it only handles fairly simple scenarios.

The new project which contains all the Perl modules for the client is named `LCFG-Client-Perl` in subversion and the component continues to be named `lcfg-client` in the standard LCFG component naming style. This completes stage 9 of the project plan.

Comments Off on LCFG Client Refactor: splitting the project | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: Comparing files

May 14, 2013

One thing that we need to do very frequently in the LCFG client, and also in many LCFG components, is comparing files. Typically we want to see if the new file we have just generated is any different from the previous version, in which case we will need to replace old with new and possibly carry out some action afterwards.

There are clearly many ways to solve this problem. We could read in the two files and do a simple string comparison (conceptually simple but tends to be messy, particularly if you want to minimise the memory requirements). It is also possible to calculate checksums for a file (MD5, SHA1, etc). I quite like this approach and it is nice and fast for small files. Up until now I've been using a mix of methods based on `Text::Diff` (wastes time since I don't actually **what** the differences are) or calculating check sums, neither of which is an ideal approach in most cases.

What I really want though is a standard API which can simply answer the question of *"are these two files the same?"*. Some of the older LCFG code shows its shell heritage by using the `cmp` command. This command does exactly what we want and does it in a fairly efficient manner. The downside is that we have to execute another process every time we want to compare two files.

Step forwards, `File::Compare`. I'm not sure why I hadn't spotted this module in the past. It works in a very similar way to the good old `cmp` command. It is also part of the set of core Perl modules which means it is available everywhere and it has a nice simple interface. I think I shall be converting various modules over to this approach in the future.

Comments Off on LCFG Client Refactor: Comparing files | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: New om interface

May 13, 2013

The LCFG client uses the `om` command line tool to call the `configure` method for an LCFG component when the resources change. Up until now this has been done using backticks which is not the best approach, particularly given that this involves building a command string and launching a full shell. I've now added a new Perl module to help with running om commands from perl. It's in version 0.7.1 of lcfg-om, you can use it like this:

```
use LCFG::Om::Command;

my ( $status, $stdout, $stderr ) =
    LCFG::Om::Command::Run( "updaterpms", "run", "-Dv", "-t" );
```

The parameters are: component, method, ngeneric args, component args. You only need to specify the component and method names, the other two are optional. The argument options can either be simple strings or references to lists.

The status will be true/false to show the success of the command. You also get any output to stdout and stderr separately.

If you’re concerned that some method might not complete in a reasonable amount of time you can specify a timeout for the command:

```
my ( $status, $stdout, $stderr ) =
    LCFG::Om::Command::Run( "openafs", "restart", "-Dv", "", $timeout );
```

If the timeout is reached then the Run command dies, you need to use `eval` or a module like `Try::Tiny` to catch that exception.

Nicely this will also close file-descriptors 11 and 12 which are used internally by the LCFG ngeneric framework for logging. This will avoid daemons becoming asociated with those files when they are restarted (and consequently
tieing up the rdxprof process).

This is one of those nice situations where fixing a problem for one project has additional benefits for others. The trick here was in realising that the code should be added to the `lcfg-om` project rather than it just being in the LCFG client code base.

Comments Off on LCFG Client Refactor: New om interface | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: File and Directory paths

May 9, 2013

The way in which the LCFG client handles paths to files and directories has never been pleasant. The old code contains a lot of hardwired paths inserted at package build-time by CMake using autoconf-style macros (e.g. `@FOO@`). This makes the code very inflexible, in particular, there is no way for a user to run the `rdxprof` script as a non-root user unless they are given write access to all directories and files in the `/var/lcfg/conf/profile` tree. There is no good reason to prevent running of `rdxprof` as a normal user, if they are authorized to access the XML profile then they should be allowed to run the script which parses the file and generates the DBM and RPM configuration files. They may not be able to run in full daemon mode and control the various components but one-shot mode certainly should be functional.

There are a couple of other things added into the mix which complicate matters further. Especially, there is some support for altering the root prefix for the file-system. This is used during install time where we are running from a file-system based in `/` as normal but installing to a file-system based in `/root`. I say **some** support since it seems that only certain essential code paths were modified.

I needed to come up with a universal solution for these two problems which could provide a fairly straightforward interface for locating files and directories. It had to neatly encapsulate the handling of any root prefix and allow non-root users to be able to store files. To this end I’ve introduced a new module, named `LCFG::Client::FileLocator`, which provides a class from which a locator object can be instantiated. There are instance attributes for the root prefix and the configuration data directory path (confdir) which can be set using `rdxprof` command line options. This object can be used to look up the correct path for any file which the LCFG client requires. There are basic methods for finding various standard LCFG paths and also useful higher-level methods for finding files for specific hosts or particular components. It’s got comprehensive documentation too so hopefully it will be a lot easier to understand in 10 years time than the previous code.

I’ve now completed stage 8 but will have to go back and finish stage 7 "*Improve option handling*", I would still like to try to add in configuration file handling. It’s a lot easier now that I’ve worked out the best way to deal with the various file paths. Having a single option for altering the configuration data directory was particularly useful.

So far I reckon I’ve spent just under 13 days of effort on the project. The allocation up to this point was 11 days (I have done the bulk of stage 7 though which takes it up to 12 days allocated). So, it’s still drifting away from the target a bit but not substantially.

Comments Off on LCFG Client Refactor: File and Directory paths | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: The joy of tests

May 9, 2013

I’m currently working on stage 8 of my project plan – "*Eradicate hard wired paths*". I’ll blog about the gory details later but for now I just wanted to show how the small number of tests I already have in place have proved to be very useful. As part of this work I have introduced a new module – `LCFG::Client::FileLocator` – which is nearly all new code. Having created this module I started porting over the rest of the client code to using it for file and directory path lookups. As I already had some tests I was able to gauge my progress by regularly running the test suite. As well as showing up the chunks of old client code which still needed to be ported it revealed bugs in 8 separate lines of code in the new `FileLocator` code. Finding these bugs didn’t require me to write a whole new set of tests for the new code (although that is on my todo list to ensure better coverage). For me that really shows the true value of writing some tests at the beginning of a refactoring process. It definitely produced higher quality code and the porting took much less time than it would have otherwise done.

Comments Off on LCFG Client Refactor: The joy of tests | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: Logging

May 8, 2013

The next stage of untangling the LCFG client code was to improve the logging system. Up till now it has just been done using a set of subroutines which are all declared in the `LCFG::Client` module. Using the logging code in any other module then requires the loading of that module, this accounts for the bulk of all the inter-dependencies between the main `LCFG::Client` module and all the others. With a single purpose the logging code is an obvious target for separation into a distinct sub-system.

With the logging code I felt that the best approach was to convert it into an object-oriented style. The typical way that logging is done in various Perl logging modules (e.g. something like `Log::Log4perl`) is to have a singleton logging object which can be accessed anywhere in the code base. The advantage of this is that it is not necessary to pass around the logging object to every subroutine where it might be needed but we can still avoid creating a new object every time it is required. If the code base was fully object-oriented we might be better served having it as an instance attribute (this is what `MooseX::Log::Log4perl` provides) but we don’t have that option here. The logging object can be configured once and then used wherever necessary. For simplicity of porting, for now, I have made it a global variable in each Perl module, that’s not ideal but it’s a pragmatic decision to help with the speed of porting from the old procedural approach.

The new `LCFG::Client::Log` module does not have a `new` method. To make it clear that we are not creating a new object every time it instead has a `GetLogger` method. If no object has previously been instantiated then one is created, otherwise the previous object is returned. Again this can be done easily using the new `state` feature in Perl 5.10, like this:

```
sub GetLogger {
    my ($class) = @_;

    use feature 'state';

    state $self = bless {
        daemon_mode => 0,
        verbose     => 0,
        abort       => 0,
        debug_flags => {%debug_defaults},
        warn_flags  => {%warn_defaults},
    }, $class;

    return $self;
}
```

This new OO-style API neatly encapsulates all the logging behaviour we require. Previously a few variables in the `LCFG::Client` module had to be made universally accessible so that they could be queried. The new module provides accessor methods instead to completely hide the internals. This all helps to make it possible to simply extend or switch to a more standard framework at some point in the future if we so desire.

Comments Off on LCFG Client Refactor: Logging | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: context handling

May 3, 2013

Now that the basic tidying is complete the code is in a much better condition for safely making larger scale changes. One of the particular issues that need to be tackled is the coupling between modules. The current code is a tangled web with modules calling subroutines from each other. This makes the code harder to understand and much more fragile than we would like. There has been some attempt to separate functionality (e.g. Build, Daemon, Fetch) but it hasn't entirely worked. For instance, in some cases subroutines are declared in one module but only used in another. The two main areas I wanted to concentrate on improving are context handling and logging.

Various client Perl modules have an interest in handling LCFG contexts and there is also a standalone script (`setctx`) used for setting context values. (For a good description of what contexts are and how they can be used see section 5.2.5 of the LCFG Guide). The context code was spread across `rdxprof` and `LCFG::Client` but in a previous round of tidying it was all merged into `LCFG::Client`. Ideally it should be kept in a separate module, this allows the creation of a standard API which improves code isolation by hiding implementation details. This in turn provides much greater flexibility for the code maintainer who can more easily make changes when desired.

The easiest part of this work was the shuffling of the context code into a new module (named `LCFG::Client::Contexts`). Once this was done I then took the chance to split down the code into lots of smaller chunks and remove duplication of functionality wherever possible (the code has gone from 4 big subroutines to 24 much smaller ones). This has resulted in a rather big increase in the amount of code (884 insertions versus 514 deletions) which is normally seen as a bad thing when refactoring but I felt in this case it was genuinely justified. Each chunk is now easier to understand, test and document – we now have a low-level API as well as the previous high-level functions. Also most of the subroutines are now short enough to view in a single screen of your favourite editor, that hugely helps the maintainer.

An immediate benefit of this refactoring work was seen when I came to look at the `setctx` script. There had been a substantial amount of duplication of code between this and the `rdxprof` script. As the context code was previously embedded in another script it was effectively totally inaccessible – the mere act of moving it into a separate module made it reusable. Breaking down the high-level subroutines into smaller chunks also made it much easier to call the code from `setctx` and remove further duplication. Overall `setctx` has dropped from 213 lines of code to 140 (including whitespace in both cases). Functionality which is implemented in scripts is very hard to unit test compared with that stored in separate modules. So it's now much easier to work with the context code and know that setctx won't suddenly break.

Comments Off on LCFG Client Refactor: context handling | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## LCFG Client Refactor: Initial tidying done

April 25, 2013

A quick dash through the code in the `LCFG::Client::Fetch` module (which is relatively small and fairly straightforward) means that all the LCFG client code has been checked using perlcritic and improved where necessary/possible. This completes the work for stages 4 and 5 of the project plan.

Some of the work involved in this stage has been rather more complex than anticipated. Mostly that was not related directly to resolving issues highlighted by perlcritic. In the main it was because whilst investigating the issues raised I spotted other, big problems with sections of code that I felt needed to be resolved. Those could have been kept separate and done as an additional stage in the project plan but I thought it was better to just do them. In particular, I have made large improvements to the sending and receiving of UDP messages for notifications and acknowledgements. I've also improved the logic involved with handling the "secure mode" and split out lots of sections of code into smaller, more easily testable, chunks.

This takes the effort expended up to about 7 days. That's about 1 day over what I had expected but that was accounted for in stages 1 and 2, the gap between predicted and actual effort requirements has not worsened.

Comments Off on LCFG Client Refactor: Initial tidying done | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## LCFG Client Refactor: Storing state

April 24, 2013

Having spent a while looking at the LCFG client code now it is clear that much of it would benefit from being totally restructured as a set of Object-Oriented classes (probably Moose-based). Making such a big change is beyond the scope of this project but there is still a need to store state in a sensible fashion. Currently the code has a heavy dependence on global variables which are scoped at the module level. In many ways the modules are being used like singleton objects and most of the globals are not accessible from outside of the parent module so it's not as bad as it could be. The biggest issue with these globals is initialisation, where multiple subroutines need to use a global they all dependent on one of them having initialised the variable first. We are thus in a situation where the order in which the subroutines are called is important. This is bad news for anyone wanting to be able to fully understand the code, it also makes it impossible to test each subroutine in an isolated fashion (i.e. given this input, do I get the right output).

With the move to SL6 we got an upgrade to perl to 5.10, this is still utterly ancient but it does provide a few new handy features. The one I've begun using a fair bit is the `state` function which is used similarly to `my`. The difference is that these variables will never be reinitialized when a scope is re-entered (whereas `my` would reinitialize the value every time). This makes it possible to write subroutines which act a bit like Object-Oriented accessors with the values being set to a sensible default value where necessary. I've used this to nicely handle the acknowledgement and notification port global variables. Here's an example:

```
use feature 'state';

sub AckPort {
    my ($value) = @_;

    # Default: Either from /etc/services or hardwired backup value
    state $ack_port = getservbyname( 'lcfgack', 'udp' )
                         // $DEFAULT_PORT_ACK;

    # Allow user to override
    if ( defined $value ) {
        $ack_port = $value;
    }

    return $ack_port;
}
```

Note that the *state* feature needs to be specifically enabled to use this approach. On the first call to the `AckPort` function the `$ack_port` variable will be initialised. If the `getservbyname` function returns an undefined value (i.e. the named service was not found) then the default value will be used. If the caller specifies a value then that will override the port number. On subsequent calls the initialisation is not done and the current value will be returned. This provides a public API for getting and setting the port number with simple handling of the default value. There is no issue of needing to know in what sequence of subroutines this method will be called, all functionality is neatly encapsulated. The method is also easily testable. Overall an Object-Oriented approach would be better but this is a good halfway house.

Comments Off on LCFG Client Refactor: Storing state | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: Sending acks

April 24, 2013

Part of the functionality in the `LCFG::Client::Daemon` code is to send acknowledgement messages to the LCFG servers whenever a new profile has been applied. The ack is sent via UDP using the `SendAck` method. The original code to do this took the traditional C-style approach:

```
  return ::TransientError("can't open UDP socket",$!)
    unless (socket(ACKSOCK,PF_INET,SOCK_DGRAM,$proto));

  return ::TransientError("can't bind UDP socket",$!)
    unless (bind(ACKSOCK,sockaddr_in(0,INADDR_ANY)));

  my $addr = inet_aton($name);
  return ::DataError("can't determine host address: $name") unless ($addr);

  my $res = send(ACKSOCK,$msg,0,sockaddr_in($aport,$addr));
  return ::TransientError("can't send notification: $name",$!)
    unless ($res == length($msg));
```

with a smattering of weirdness and `unless` thrown in for good measure. Things have moved on a bit since the days when this was the recommended approach. There is now a very handy suite of modules in the `IO::Socket` namespace which can handle the dirty work for us. The replacement code looks like this:

```
   my $port = AckPort();

    my $socket = IO::Socket::INET->new (
        PeerAddr   => $server,
        PeerPort   => $port,
        Proto      => 'udp',
    ) or return LCFG::Client::TransientError(
             "can't connect to $server:$port", $! );

    my $res = $socket->send($message);

    $socket->close();

    if ( $res != length $message ) {
        return LCFG::Client::TransientError(
                 "can't send notification: $server", $! );
    }
```

That is, without a doubt, much easier to read and maintain. We are now relying on someone else to do the socket handling but that's fine as this is a core Perl module which should be totally reliable.

Comments Off on LCFG Client Refactor: Sending acks | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: Daemon state tables

April 17, 2013

Having finished the tidying of the `LCFG::Client::Build` module I have now moved onto `LCFG::Client::Daemon`. The first thing which caught my eye was the handling of the state tables. These state tables are used to control how the daemon handles notifications from the server, timeouts and signals from the LCFG client component. I pretty much totally rewrote the `MakeTable` function so that it processed the input text and built the data structures for the tables in a much cleaner and more comprehensible manner. As with previous changes, my first step was to write some tests which checked the old function then ran them again with the new code to ensure I had not altered the API. I also introduced a new `NextStateInTable` function which contained code which was previously duplicated inside `NextState`. Finally I introduced an `InitStateTables` function which is called from within `ServerLoop` which hides the initialisation of the global variables used to hold the state tables. This means we now have a much cleaner API for handling all the state transitions based around smaller, testable functions.

Comments Off on LCFG Client Refactor: Daemon state tables | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: tidying LCFG::Client::Build

April 17, 2013

The `LCFG::Client::Build` module is the largest part of the LCFG client code. It weighs in at 1800 lines which is nearly 50% of all the code in the project. It contains a lot of functionality related to processing the data from the XML profile into the format stored in the local DB file and triggering components to reconfigure as necessary. Improving this code was always going to be a big task but at least once this module is done the remainder will seem easy.

The main changes which stand out are, like with `LCFG::Client`, related to noticing repeated coding of the same functionality. The first larger change came from noticing that in many places the value of an attribute (for example, the LCFG resource value) are decoded using the `HTML::Entities` module but only for LCFG profile version 1.1 and newer. Now we probably haven't supported anything older than this for a **very** long time but it occurred to me that rather than just drop the version check it would be better to completely enhance the attribute decoding. So, rather than have calls to `HTML::Entities::decode` all over the place we now pass the value through a new `DecodeAttrValue` function which in turn calls a new `NeedsDecode` function to check if decoding is required. These are both small easily testable functions so I added a few tests along the way. The big benefit here is that if we now ever need to change the encoding/decoding of values and increment the profile version we are already prepared for the necessary code modifications.

The second big change was to improve the code of the `InstallDBM` function. This had two copies of a complex regular expression used to parse a fully-qualified resource name (e.g. host.component.resource_name) so I moved this code into a new function named `ParseResourceName`. Again this is now easily reusable and testable whereas before it was buried in the midst of other complex code. This led to some other improvements in how the debugging was done, I noticed there were many calls to `KeyType` which was just returning a prettified name for the underlying attribute type indicators which are all single characters (in the set `[#%=^]`). Each debug statement was very similar but handled a slightly different case, these were all merged into a `ResourceChangesDebug` function. This new function massively improves code readability and also improves efficiency since it only actually does something when the "*changes*" debug option is enabled. By reworking the debugging it is now possible to use the `KeyType` function in a totally generic manner. Anything which needs to know about the type of the attribute can work with the named versions rather than the almost-meaningless single character indicators.

There is still a lot more to do on this module to really improve the code standards but much of that might well be beyond the scope of this initial code cleanup project. The XML profile parsing and the DB handling are particularly in need of attention.

Comments Off on LCFG Client Refactor: tidying LCFG::Client::Build | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

## LCFG Client Refactor: tidying LCFG::Client

April 8, 2013

The first round of tidying code to satisfy perlcritic was focussed on the central `LCFG::Client` module which contains code used by all the other modules.

As well as the tidying there were a couple of slightly larger changes. I had spotted that several routines (RPMFile, DBMFile and ProfileFile) were each doing their own mangling of the host FQDN and then doing similar work based on the results. To reduce duplication I introduced a `SplitFQDN` function which contains an improved version of the hostname splitting functionality (and which can now be used in other places). I then also introduced another new function (named `HostFile`) which contains all the other duplicated functionality between the 3 functions. Each of the 3 functions are now pretty much reduced to a single line call to `HostFile` with the relevant parameters set. At the same time as adding these new functions I added tests for them and also the higher-level functions which use them. As each has now been reduced in complexity it is much easier to test them. This gives me a good guarantee that if I have to make changes in the future they will continue to work as expected.

Beyond tidying the code to resolve the worst of the perlcritic issues I also applied a number of changes which come from the lower levels. In particular I removed a lot of unnecessary brackets associated with calls to built-in functions and conditionals. This might seem like a tiny thing but it does reduce the “noise” quite considerably and vastly improves the code readability. I also removed all uses of the `unless` conditional, this is something which drives me crazy, anything more than an utterly simple condition is very hard to comprehend when used in conjunction with `unless`. That is one feature I really wish was not in Perl! I’ve seen unless-conditions which are so complicated that only a truth table can fathom out what is going on…

Another code smell which was eradicated was the heavy usage of the default scalar variable (`$_`). In my opinion there is no place for using this in large code bases outside of situations like code blocks for `map` and `grep`. Using it pretty much guarantees that there will be the potential for weird, inexplicable action-at-a-distance side-effects in your code.

One thing I would like to spend more time on at some point is improving the naming of variables. There is frequent use of single-letter variable names (`$c`, `$t`, etc) which is mostly meaningless. This might not be a problem in a very short (couple of lines) block where the context is clear but in chunks longer than a screen-full it’s really hard to track the purpose of all the variables. There is also quite regular reuse of variable names within a subroutine which again makes mentally tracking the purpose of each variable very difficult.

Comments Off on LCFG Client Refactor: tidying LCFG::Client | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## LCFG Client Refactor: perltidy and perlcritic

April 5, 2013

The next phase of the project to clean up the LCFG client (goals 3, 4 and 5) is to run everything through the perltidy tool and then fix the code to satisfy the perlcritic code checker down to level 4. Having all the code consistently indented with an automated tool may seem like a trivial thing to do but it makes such a difference to the maintainability of code. I realise that Python coders have been going on about this for years so it’s nothing new… We chose a coding style for the LCFG server refactoring project and I am using the same for the LCFG client. At the time we added a few notes on the LCFG wiki PerlCodingStyle page. I guess I probably ought to upload my `.perltidyrc` configuration file to that page so that it can be easily reused.

The use of perlcritic to check code is probably slightly more controversial for some people. It checks your code against a set of rules and recommendations originally laid out in Damian Conway’s book Perl Best Practices. If you don’t like some of those rules you are going to find it very annoying. We’ve found that aiming to satisfy levels 4 and 5 (the most critical issues) results in a vast improvement in code quality. Below that you very rapidly get into a lot of tedious transformations not all of which give you any great benefit. Knowing when to just ignore the moans of the tool is a very useful skill.

Comments Off on LCFG Client Refactor: perltidy and perlcritic | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## LCFG Client Refactor: rdxprof finished

April 5, 2013

The work to cleanup rdxprof is now pretty much finished. All the functionality has been moved out into the `LCFG::Client` module so that all that happens in the rdxprof code is 3 simple calls to subroutines in the core module:

1. **SetOptions** – Parse the command line parameters and sets LCFG::Client variables
2. **Init** – Initialises the environment (mostly just ensuring certain directories exist)
3. **Run** – This does the actual work (either OneShot or ServerLoop)

There is still a small number of dependencies on global variables that would be nice to remove in the future but nothing critical for now.

This concludes goals 1 and 2 on the project plan. The hope was that this would only take one day of work but it ended up needing 2 days. That is due to my not having initially spotted the real degree of peculiarity of the coding style. The rdxprof code was definitely much more complex in terms of how it approached the “structure” of the entire program than anything I had encountered in the LCFG server code refactoring project. Hopefully now that particular intricate unpicking job is complete the rest will be more straightforward.

Comments Off on LCFG Client Refactor: rdxprof finished | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## LCFG Client Refactor: rdxprof cleanup

April 2, 2013

The refactoring of the LCFG client has been continuing at good pace. I have now managed to move all the subroutines from the rdxprof script into the `LCFG::Client` module. This means that it is now possible to add unit tests for the functionality in these subs and I spent a bit of time yesterday adding the first simple tests. There are a **lot** more to go but it’s a good start. Adding the tests really helped me load more of the code into my brain so there are more benefits than just having testable code.

The big job for yesterday was really improving the sanity of the global variables. Some of the module code relied on the fact that it was being called from rdxprof to be able to access global variables declared in that script. Thus those modules wouldn’t work properly when loaded individually. In one case (the `$root` variable) it was declared as a global in two places and used as a local variable in many subroutines when passed in as an argument, that’s just a recipe for utter confusion. I’ve now removed one global but there is clearly a need to improve the situation further.

I also moved all the uses of values which are hardwired at build-time using cpp-style macros (e.g. `@FOO@`) into readonly variable declarations at the top of the modules. This makes it much more obvious which hardwired strings are required by each module. This is a first step towards replacing this approach with a configuration module (e.g. `LCFG::Client::Config`) which is how we handled the situation for the LCFG server.

Comments Off on LCFG Client Refactor: rdxprof cleanup | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

## Refactoring the LCFG client

March 29, 2013

The time has come to start work on refactoring the code base of the LCFG client. This has been overdue for a while now as the current state of the code is preventing us from doing much in the way of new developments for fear of breaking something important. The aim is to tidy and clean the code to bring it up to modern Perl standards and generally make it much more readable and maintainable. The aim is to avoid altering the functionality if at all possible although a number of small bug fixes will be tackled if time allows. The full project plan is available for reading on the devproj site. This project incorporates many of the lessons we learnt when we refactored the LCFG server code last year, again see the devproj site for details.

I made an initial start on the project today. As with all refactoring the best first move is to ensure you have some tests in place. In this case I just added simple compilation tests for each of the 5 Perl modules involved. Interestingly this immediately flagged up a genuine bug which existed in the code, this was related to the use of subroutine prototypes. Now anyone who has a reasonable amount of experience with Perl programming will tell you that subroutine protoypes are evil, full of gotchas and rarely do what you expect. One of the tasks in the plan is to rid the code of them entirely but that’s not for today. Thankfully this was a simple error where the prototype stated that 4 scalars were required when, in actual fact, only 3 were needed (and only 3 were provided when the subroutine was called). I’m surprised the code actually worked at all with that bug, this shows how useful even simple testing can be for improving code quality.

The whole code base is basically 5 Perl modules and a script which uses them all. An interesting strategy was taken with the module loading, all subroutines from the modules were imported into the “main” namespace of the script (which is effectively global) and then all calls to them anywhere in the code base were referred to the version in that namespace. So, all subroutine calls were done with unqualified, short names, I guess this makes it quick to hack out but coming at the code without a huge amount of prior knowledge it is almost impossible to quickly reckon the source location for each subroutine. So, my second step was to work through all the code and replace the calls with fully-qualified names. To make it doubly clear that the old way wasn't readable or maintainable I also ripped out (the now unnecessary) support for exporting subroutines into another namespace and ensured that when these modules are loaded there is no attempt to import anything.

This sort of change should be zero impact, right? Turns out, not entirely, nothing is ever simple… I had to shuffle a few subroutines out of the script into the helper modules, in turn that meant fixing a few references to global variables. This in turn required passing another parameter to a couple of subroutines which meant hacking out a few evil subroutine prototypes. I think that shows up a few code smells which will have to be tackled very soon.

Before I can really get stuck in though a few more tests are going to be necessary. At the very least there is going to have to be a test of the client's ability to download an XML profile and convert it into the locally stored file format. At this stage I don't know enough about the code to create tests for each subroutine so a large-scale test of functionality is the only option. Without that test it won't be safe to make any bigger code changes.

Comments Off on Refactoring the LCFG client | Uncategorized | Tagged: lcfg, lcfg-client, refactoring | Permalink
Posted by squinney

---

- March 2018

| M | T | W | T | F | S | S |
|---|---|---|---|---|---|---|
| | | | 1 | 2 | 3 | 4 |
| 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 12 | 13 | 14 | 15 | 16 | 17 | 18 |
| 19 | 20 | 21 | 22 | 23 | 24 | 25 |
| 26 | 27 | 28 | 29 | 30 | 31 | |

« Feb

- **Tags**

afs agile apache bugzilla buildtools catalyst cmake cosign cpan dice f12 fail2ban fc3 fedora fedora-12 fosdem gnuplot koji lcfg lcfg-auth lcfg-client lcfg-server ldap logging mock Module::Build moose pandas perl perl6 profsec python refactoring remctl rhel security spring2008 sql testing ukuug xrdp xs

- **Blogroll**
  - Informatics blogs
  - WordPress.com
  - WordPress.org

- **Meta**
  - Site Admin
  - Log out
  - Entries RSS
  - Comments RSS
  - WordPress.org

- **Archives**
  - March 2018 (1)
  - February 2018 (2)
  - November 2017 (5)
  - May 2017 (3)
  - September 2016 (1)
  - August 2016 (1)
  - July 2016 (1)
  - May 2016 (3)
  - January 2016 (2)
  - November 2015 (1)
  - August 2015 (2)
  - June 2015 (1)
  - January 2015 (1)
  - December 2014 (4)
  - November 2014 (1)