UCL for FreeBSD

A universal config language for (almost) everything in FreeBSD

Allan Jude -- ScaleEngine Inc.
allanjude@freebsd.org twitter: @allanjude

Introduction

Allan Jude

- 13 Years as FreeBSD Server Admin
- FreeBSD docs committer (focus: ZFS, bhyve, ucl, xo)
- Co-Author of “*FreeBSD Mastery: ZFS*”
- Architect of the ScaleEngine CDN (HTTP and Video)
- Host of BSDNow.tv & TechSNAP.tv Podcasts
- Extensive work with Puppet to manage our 100+ servers in 35 data centers in 12 countries
- Lots of work with ZFS to manage large collections of videos, extremely large website caches, and the pkg mirror for PCBSD

Impetus

At EuroBSDCon 2014 in Sofia, Jordan Hubbard (jkh@freebsd.org) gave the opening keynote

Among the recommendations for how FreeBSD could be improved and made easier to manage was to move away from the ‘one config file format per utility’

Jordan, being from Apple, suggested the binary XML plists used by launchd. I really didn’t want the “one true format” to be XML.

Why is this a “Good Idea”™?

- Includes allow for more readable, structured, and upgradable config files
- Overlays separate the defaults from your site wide settings, and from your local settings (like defaults/rc.conf rc.conf rc.conf.local)
- Includes allow packages to install default settings (installing apache or nginx adds its log rotation to newsyslog)
-
-

What is UCL

- UCL -- universal configuration language
- Inspired by bind/nginx style configuration file that all sysadmins know well
- Fully compatible with JSON, but more liberal in what it accepts, so users do not have to write strict JSON
- Can output the configuration as ‘configuration’ (bind/nginx style), JSON (compact or pretty), or YAML
- Supports handy suffixes like k, kb, min, d

What does UCL look like?

```
key = “value”;
logical_name: {
    “somekey”: “somevalue”,
    port: 80;
    enabled: true
}
somethingelse {
    interval = 5min
}
```

Why UCL is better than JSON

- UCL can input and output JSON
- However, UCL is less strict
- Quoting is not required
- Commas can be added after every element (smaller diffs)
- Braces are not necessary
- Can use = instead of :
- Automatic array creation
- In general, less strict about syntax, writing JSON by hand is hard, so easier to handle

Why use UCL?

- Very flexible, can replace most config files
- Can be as simple as rc.conf or can express complex, structured data
- supports nesting
- rich includes feature (priorities, wildcards)
- Terse. contain little excess formatting or syntax (unlike XML)
- Human readable, and importantly, writable
- Allows comments (unlike JSON)
- Already in the base system, used by pkg(8)

crontab(5) before:

```
MAILTO=””
*/15 * * * * /usr/home/semirror/semirror.sh
```

After:

```
cron_name {
    command = "/usr/home/semirror/semirror.sh"
    user = "semirror"
    minute = "*/15"
    hour = "*"
    environment {
        MAILTO = ""
        PATH = "/sbin:/bin:/usr/sbin:/usr/bin:/usr/local/sbin:/usr/local/bin"
    }
}
```

newsyslog.conf(5) before:

```
/var/log/messages     644  5     100  @0101T JC
```

After:

```
mylogfile: {
     filename: “/var/log/messages”
     mode: 644
     count: 6
     size: 100kb
     when: “@0101T” /* this could be written in crontab(5) style? */
     create = yes
     compress = xz
}
include “/usr/local/etc/newsyslog.d/*.conf”
```

Extending the concept

- Moving all default entries to /etc/defaults means easier upgrades (mergemaster, etcupdate)
- UCL’s override feature allows you to change a specific settings from the default
- If a user wants to keep more versions of /var/log/messages, they add /etc/newsyslog.d/mylog.conf and override the ‘count’ key.
- Extend things like /etc/newsyslog.d/ for crontab, add a /usr/local/etc/ version as well
- Disable a default crontab? mycron {enabled: no}

Precedence Issues

If the configurations will allow ‘overlays’, such that the user can change the values from /etc/defaults/crontab in another file, it may be required to specify the priority of some of the config blocks, to solve conflicts. For example:

/etc/pkg/FreeBSD.conf: enabled=yes

/usr/local/etc/pkg/repos/FreeBSD.conf: enabled=no

/usr/local/etc/pkg/repos/something.conf: enabled=yes

What order are they considered in?

A priority key, sorted descendingly, so the config with ‘priority=1’ is considered last and gets ‘the final say’

Secondarily sort lexicographically to resolve conflicts

(Since I wrote this, this has been implemented in libUCL)

Ordering Issue

Some configs may be sensitive to the order of the declarations.

An ‘order’ key, when the configs from:

- /etc/defaults/newsyslog.conf
- /etc/newsyslog.conf
- /etc/newsyslog.d/*.conf
- /usr/local/etc/newsyslog.d/*.conf

are combined, they could be sorted by this ‘order’ key, which defaults to MAX, so any config block without an order key is sorted last.

Actually Doing It

After a lot of talk about how it would be really nice if we had all of this, while at the airport after MeetBSD, and on the flights home, I started to write some code.

I decided newsyslog was simple enough that my very novice understanding of C wouldn’t slow me down too much.

After the flight, and a few weekends of non-stop hacking, I had a prototype:

https://reviews.freebsd.org/D1548

Compatibility

In order to smooth the transition from legacy config files to the new UCL format, the new config files will be marked with a version sentinel. If the config file does not have the magic mark on the first line, it is parsed using the original code.

I decided it made sense to add versioning support from the start, because I am sure I'll make some mistakes, and that we'll want to change the schema again later.

New newsyslog startup routine

- newsyslog starts as it did before
- reads the first line of the config file, then rewinds the file pointer to the start of the file
- if the first characters are #fucl, check the version number
- else, parse with the legacy config code
- Run the parser that matches the version, error if the version is too new or no longer supported

New Parser

- The new parser runs libUCL on the config
- libUCL handles includes, layering, etc
- Results in a libUCL object containing the resultant configuration
- Iterate over each log file (root keys in the object)
- Process the sub-keys, and load the configuration into the existing unmodified structs
- Continue the rest of the code as normal

Remaining Items

- Design a consistent set of common keys for use across all configuration files
  - enabled, order, others?
  - should they be namespaced?
- Develop a universal C api for loading data from libUCL objects into existing data structures in various applications
  - libfigpar?
- More required features for uclcmd
- Was the topic of a Dev Summit working group here at BSDCan. <discuss results>

Other Victims

- bhyve
- crontab
- iscsi / ctld
- autofs
- freebsd-update
- portsnap
- jail.conf (need support for variables like ${host.hostname} in path, has keys with dots in them)
- devd.conf (syntax doesn’t match well, will need work)

New Toys

- While thinking about Jordan’s talk, I saw the need for a tool for interacting with UCL config files via the command line (and via shell scripts) -- github.com/allanjude/uclcmd
- sysrc type tool to be able to extract values in a shell script:

```
# uclcmd get -f /etc/pkg/FreeBSD.conf freebsd.url
    "pkg+http://pkg.FreeBSD.org/${ABI}/latest"
```

- Already in use: shell script to launch a bhyve VM from a UCL config file: https://github.com/allanjude/bhyveucl

More Uses

- Reading is all well and good, but I want to modify my configs from puppet/salt/etc, without a lot of hassle
- Lets write some UCL into a file
- Simple case: change the pkg repo

```
# uclcmd set -f /usr/local/etc/pkg/repos/myrepo.conf
myrepo.url https://pkg.scaleengine.net/10_64-default-edge
```

- More complex cases require a bit more code

More Features of uclcmd

- Supports some basic looping functionality
- Inspired by jq, a tool for parsing JSON at the command line
- **each** - perform an operation on each key
- **keys** - list the keys of the selected object
- **values** - show the value of each key
- **length** - how many items in an array
- **type** - the type of the object
- **iterate** - UCL’s iteration function (useful for implicit arrays etc)
- **recurse** - flatten tree into key/value pairs

More Power

- 4 basic operations:
- **Get** - read a key or object
- **Set** - overwrite (create) in place
- **Merge** - combine with existing data, create if required
- **Remove** - delete a key
- Example: Merge operation: Take a UCL fragment via STDIN and merge it with what is already in the file

Figure 4

```
root {
    rootval = “this is a key in the first object";
    subkey {
        size = 100kb;
        listofthings = [
            { name = thing1, value= value1 },
            { name = thing2, value= value2 },
            { name = thing3, value= value3 },
            { name = thing4, value= value4 },
        ];
    }
}
```

Get Operations

```
% uclcmd get -f fig4.conf .root.rootval
"this is a key in the first object"
% uclcmd get -f fig4.conf '.root.subkey.
listofthings|each|.value'
"value1"
"value2"
"value3"
"value4"
```

Figure 5

```
root {
    subkey {
        size = 150kb;
        newkey = “newvalue";
    }
}
```

Change the size from 100kb to 150kb
Add the new key ‘newkey’

Merge Operation

```
% uclcmd merge -f fig4.conf -i fig5.conf --ucl .
```

```
root {
        rootval = "this is a key in the first object";
        subkey {
                size = 153600;
                listofthings [
                        { name = "thing1"; value = "value1"; }
                        { name = "thing2"; value = "value2"; }
                        { name = "thing3"; value = "value3"; }
                        { name = "thing4"; value = "value4"; }
                ]
                newkey = "newvalue";
        }
}
```

Moving Forward

- Some more tools in base for things like:
  - Schema Validation - UCL has a system for this, but it is not exposed to the command line easily
    - maybe a tool like vipw, atomically update config and block if parse fails
    - libUCL uses mmap() for includes, may require special consideration if files shrink
  - Config compilation - show what the final config looks like, all of the defaults and the multiple layers of config applied on top of them to show the final config (ala samba testparm)
  - Recipes for using uclcmd in common orchestration frameworks like puppet, ansible, salt, etc

Future Ideas

- "show running config" - A single compiled config for the entire system, each service in its own subkey
    - Possibly implement parsers for some config files that are not converted to ucl (sshd_config) so they can be included in the running config
- libuclnv - ucl -> nvlist, and nvlist -> ucl, by borrowing the libucl parser and emitter, but using nvlists for the internal representation
- Kyua regression tests for libUCL
- More simple tests for uclcmd

BSDctl?

- Light dependencies (no python or ruby)
- sysrc (rcctl for openbsd)
- service [start|stop|status]
- similar for sysctl’s
- data providers:
    - sysrc/rcctl (is sshd enabled)
    - service (is sshd running)
    - bsnmpd/openbsd snmpd (counters)
    - sysctl (hw.ncpu, hw.physram, etc)
    - uclcmd (view/extract/change config)
    - libxo? (ifconfig etc, how is the network configured)

Podcasts

BSDNow.tv is a weekly video podcast featuring News, Interviews and Tutorials about the BSD family of Operating Systems. Hosted by Kris Moore (founder of PC-BSD) and Myself.

TechSNAP.tv is a weekly sysadmin video podcast covering an OS agnostic range of security and production issues of interest to those working, studying or interested in the field.

Twitter: @allanjude Email: allanjude@freebsd.org