clufter: CLUster Filter

what+why, how, highlights

jpokorny@redhat.com
May 2015
clufter: intro & motivation

$ clufter -h | sed -n '3,8p'

Tool/library for transforming/analyzing cluster configuration formats

While primarily aimed at (CMAN,rgmanager)->(Corosync/CMAN,Pacemaker) cluster stacks configuration conversion (as per RHEL trend), the command-filter-format framework (capable of XSLT) offers also other uses through its plugin library.

Primary use-case it aims to assist with is a cluster/HA configuration migration from a CMAN/rgmanager stack onto its (partial or full) successors featuring Pacemaker.

That's one of the possible convergence paths from previously fragmented free software clustering/HA landscape – here in particular the one essentially connected with Red Hat and its products (self-explanatory, isn't it?).
The framework & structure

Both stacks talk **XML** (Corosync configuration format can be, to some extent, also expressed in XML).

Hence the conversion naturally boils down to **XML transformations**.

For XML transformations, formalism + implementation already exist (**XSLT** and **libxml+libxslt** available in various forms/bindings).

**Problem #1**: Cannot rely just on XSLT, need some notion of generalized *pass* (XSLT or distilling [perhaps intermediate] output by other means).

**Problem #2**: One such pass not enough (+ XSLT 2.0 not readily available, yet), needs chaining/pipeline (even non-linear) of these passes.

**Solution #1**: **Framework** allowing to define chaining/pipeline dataflow as directed acyclic graph with **filters** (those passes/points of atomic transformation) as the nodes. Such graphs, enriched with the help-screen metadata, etc. actually define user-visible **commands**. Data passed across the graph in the chain are encapsulated as **formats'** instances (held in one of supported, convertible representations).
The components (plugins)

**formats**
- SimpleFormat vs. CompositeFormat (bases)
- XML (also built-in base)
- cib, ccs, coroxml, ...

**filters**
- `{,XML}Filter` (bases)
- ccs2ccsflat (non-XSLT, running embedded binary)
- ccsflat2cibprelude, ...

**commands** (note: identifiers in code vs. run-time separators: `_` vs. `-`)
- Command (base)
- ccs2pcs, ccs2pcscmd (old-stack config to new-stack config as raw files or pcs commands), pcs2pcscmd (just new-stack config as pcs commands) – actually *aliases* resolved per target system (**RHEL 6/7** [flatiron/needle]) specifics (current system or via options)
- ccs_artefacts, ccs_obfuscate, ccs_revitalize, ...
$ clufter -l  # <-- executed on RHEL 6.7

Commands (as available, but stable); those without star ('*') are built-in:

- **ccs-artefacts**  
  Output artefacts referenced in the config. in CVS format.

- **ccs-obfuscate**  
  Obfuscate credentials/IDs in CMAN-based cluster config.

- **ccs-revitalize**  
  Migrate deprecated config's props (agent params, etc.)

- **ccs2pcs-flatiron**  
  (CMAN,rgmanager)->(Corosync/CMAN,Pacemaker) cluster cfg.

- **ccs2pcs-needle**  
  (CMAN,rgmanager)->(Corosync v2,Pacemaker) cluster cfg.

- **ccs2pcscmd-flatiron**  
  (CMAN,rgmanager) cluster cfg. -> equivalent in pcs commands

- **ccs2pcscmd-needle**  
  [COMMAND CURRENTLY UNAVAILABLE]

- **pcs2pcscmd-flatiron**  
  (Corosync/CMAN,Pacemaker) cluster cfg. -> reinstating pcs commands

- **pcs2pcscmd-needle**  
  [COMMAND CURRENTLY UNAVAILABLE]

**Aliases thereof (environment-specific):**

- **ccs2pcs**  
  alias for ccs2pcs-flatiron

- **ccs2pcscmd**  
  alias for ccs2pcscmd-flatiron

- **pcs2pcscmd**  
  alias for pcs2pcscmd-flatiron
User-facing: commands (2)

$ clufter ccs2pcs -h  # <-- executed on RHEL 6.7

Usage: clufter [<global option> ...] ccs2pcs [<cmd option ...>]
        clufter [<global option> ...] ccs2pcs-flatiron [<cmd option ...>]

(CMAN,rgmanager)->(Corosync/CMAN,Pacemaker) cluster cfg.

More specifically, the output is suitable for Pacemaker integrated with
Corosync ver. 1 (Flatiron) as present, e.g., in RHEL 6.{5, ..}, and consists
of Corosync/CMAN configuration incl. fencing pass-through (~cluster.conf)
along with Pacemaker proper one (~cib.xml).

Options:
[...]  
  -i INPUT, --input=INPUT  
      input (CMAN,rgmanager) cluster config. file  
      [/etc/cluster/cluster.conf]
  -c CCS_PCMK, --ccs-pcmk=CCS_PCMK  
      output Corosync/CMAN (+fencing pass-through) config.  
      file [cluster-{ccs2ccsflat.in.hash}.conf]
  --cib=CIB  
      output proper Pacemaker cluster config. file  
      [cib-{ccs2ccsflat.in.hash}.xml]
Below commands

Graph prescribing command's dataflow expressed (per documented syntax) as an annotation of the command's main function that usually just maps overall arguments to the I/O specification of terminal (input/output) filters.

```python
@Command.deco('ccs-artefacts')
def ccs_artefacts(...):

@Command.deco(('ccs-obfuscate-credentials',
               ('ccs-obfuscate-identifiers')))
def ccs_obfuscate(...):
```
User’s look under the hood

$ clufter ccs2pcs -H  # <-- executed on RHEL 6.7; -H for “full help”.

[...]  

--noop=NOOP  debug only: NOOPize filter (2+: repeat) [none out of  
             ccs2ccsflat, ccs2ccs-pcmk, ccs-revitalize,  
             ccsflat2cibprelude, cibprelude2cibcompact,  
             cibcompact2cib, cib2cibfinal]

--dump=DUMP  debug only: dump (intermediate) output of the filter  
       (2+: repeat) [none out of ccs2ccsflat, ccs2ccs-pcmk,  
           ccs-revitalize, ccsflat2cibprelude,  
           cibprelude2cibcompact, cibcompact2cib, cib2cibfinal,  
           ANY]

The above options can be used for debugging purposes (“skip particular  
filter/s in the chain”, “dump the intermediate product to a dedicated  
file”, respectively) and also to actually see the filters employed  
by particular command.
Complex command sample (1)

ccs2pcs-needle

ccs2ccsflat

ccs-flat

ccs-propagate-cman

ccs

ccs2needlexml

corxml-needle

XML

xml2simpleconfig

simpleconfig

ccsflat2cibfinal_chain
Complex command sample (2)

ccsflat2cibfinal_chain

ccsflat2cibprelude

cib-prelude

cibprelude2cibcompact

cib-compact

cibcompact2cib

cib

cib2cibfinal

cib-final

ccs

ccs-revitalize

ccs

ccs-revitalize

ccs

excuse lack of better names
The framework: sanity first

For common (intermediate) product of the filters, XML, also ready-to-use validation systems exist – RelaxNG is both a popular and Pacemaker's choice.

Plus it is easy to spoil the conversion by making the products invalid because of inconsiderate modifications of XSLT snippets backing particular filter.

Problem #3: In the course of a single command, there can be lots of (intermediate) products as XML. It is vital that final ones are valid, which expects guarding the assumptions throughout the chain.

Problem #4: Moreover, in case of validation failure, it is desirable for user to help herself by modifying respective product (or even better, just a relevant part of that), i.e., assisted recovery instead of hard stop.

Solution #II: Append validation procedure (on opt-out basis) after any XSLT filter that produces a format with a defined validation schema, and run validation also within the particular steps of a single filter when possible. For Pacemaker use pacemaker-1.2.rng (compilation of the native schema), for Corosync use custom-built schema (even proposed upstream as a way to consolidate config. management/man page generation, etc., but hadn't been bought).
Guarded command sample (1)
Guarded command sample (2)

ccsflat2cibfinal_chain

ccs2
\text{ccs-revitalize} \quad \text{ccs6}

ccs6
\text{ccsflat2cibprelude} \quad \text{cib-prelude7}

cib-prelude7
\text{cibprelude2cibcompact} \quad \text{cib-compact8}

cib-compact8
\text{cibcompact2cib} \quad \text{cib9}

cib9
\text{cib2cibfinal} \quad \text{cib-final10}

\text{excuse lack of better names}
The CLI tool (1)

$ # executed on RHEL 6.7, i.e., ccs2pcscmd-flatiron is what's effectively used
$ clufter ccs2pcscmd --input=- --output=- <<-EOF
    <cluster name="clu" config_version="1">
        <clusternodes>
            <clusternode nodeid="1" name="a" />
            <clusternode nodeid="2" name="b" />
            <clusternode nodeid="3" name="c" />
        </clusternodes>
        <totem consensus="200" join="100" token="5000"/>
        <rm>
            <failoverdomains/>
            <resources/>
        </rm>
    </cluster>
EOF
    [cibcompact2cib     
    XSLT: NOTE: no fencing is configured hence stonith is disabled; please note, however, that this is suboptimal, especially in shared storage scenarios
pcs cluster setup --name clu a b c --consensus 200 --join 100 --token 5000
pcs property set 'stonith-enabled=false'
    [stringiter-combine2] output file: <stdout>
The CLI tool (2)

$ clufter ccs2pcscmd -h
[...]
Command options:
  -i INPUT, --input=INPUT
    input (CMAN,rgmanager) cluster config. file
    [/etc/cluster/cluster.conf]
  -o OUTPUT, --output=OUTPUT
    pcs commands to reinstate the cluster per the inputs
    [-]
[...]
Arguments to value-based 'command options' can go without labels when the order wrt. parsing logic respected; skipping those backed by default values otherwise requiring specification then allowed by syntactic sugar: all can be passed as a single, first, ::-delimited argument; magic files: '~', '@DIGIT+'. '{formula}' in output file spec: input-backed (e.g. hash) substitution recipe.
[...]
...which means following invocations are equivalent:
$ clufter ccs2pcscmd --input=- --output=-
$ clufter ccs2pcscmd -- # arguments passed positionally (per help-screen order)
$ clufter ccs2pcscmd -    # + rely on default output
Typical scenario with XSLT: one big mess upon a time due to uneasy modularization. But with cluster configuration formats, we have to deal with more or less regular structure (exc.: resource groups, see later).

Problem #5: Need to apply typical divide-et-impera approach to decompose complex transformation & concentrate on smaller bits, chunk-by-chunk, in an isolated manner (XSLT follows this principle, but...).

Problem #6: Need to find a scalable way to combine lots of (decomposed) transformations, ideally based on hierarchical grouping into likewise structured modules (as an well-established practice).

Solution #III: Split all the transformation on particular well-defined XML document type per the nesting hierarchy of the tags, putting transformations snippets that are limited to the particular XML subtree as the input into the same module, distinguished by the names of variables storing the snippets (discovery mechanism will pick up just those matching the selected filter's name). Python's native module nesting then reflects the document schema nicely, which brings better maintainability.
Dev: decomposition FTW (2)

**XSLT decomposition sketched so far is too loose...**

**Problem #7:** The relationship between snippet that is higher in the hierarchy to his descendant in the decomposition has to be well-defined.

**Problem #8:** In addition, one objective we pursue is ability to validate portion of the result as small and early as possible (see **Problem #4**) in our step-wise transformation engine, i.e., bottom-up validation.

**Solution #IV:** On top of standard XSLT instructions, define custom tags that will tell the pass how to glue snippets in the hierarchy together and/or how to proceed at particular points of transformation.

- **Clufter:** `descent` instructs the transformation engine to recurse into handling specified subtrees (denoted by `at` attribute, defaults to any: *) that are transformed first on their own, then appended at the tag's place.

- **Clufter:** `descent-mix` instructs the transformation engine to recurse into handling subtrees (denoted by `at` attribute, defaults to any: *) transformation of which is immediately appended at the tag's place, optionally along identity transform (`preserve-rest` attrib.).
Dev: decomposition FTW (3)

(filters/cluster/__init__.py (simplified):
ccs2needlexml = ''
<corosync>
  <clufter:descent at="clusternodes"/>
  [...]
</corosync>''

(filters/cluster/clusternodes/__init__.py:
ccs2needlexml = ''
<nodelist><clufter:descent at="clusternode"/></nodelist>
  ''

(filters/cluster/clusternodes/clusternode/__init__.py (sim.):
ccs2needlexml = ''
<nodelist>[...]</nodelist>'')
Dev: decomposition FTW (4)

```
filters/cluster/__init__.py:
ccspcmk2pcscmd = '''
    <xsl:output method="text"/>
    <xsl:strip-space elements="*"/>
    <xsl:value-of select="concat('pcs cluster setup',
        ' --name ', @name)"/>
    <clufter:descent-mix at="clusternode"/>
    <clufter:descent-mix at="cman"/>
    <clufter:descent-mix at="totem"' />
'''

filters/cluster/clusternodes/clusternode/__init__.py:
ccspcmk2pcscmd = '''
    <xsl:value-of select="concat(' ', @name)"/>
    <xsl:if test="altname/@name">
        <xsl:value-of select="concat(',', altname/@name)"/>
    </xsl:if>'''
```
Dev: macros FTW

**XSLT gets self-repeating and boring pretty fast...**

**Problem #9:** Even with decomposition (Solution #III and Solution #IV), XSLT snippets get cumbersome easily for lack of plaintext expansion.

**Solution #V:** We can apply the same generic approach that was used in C language – using out-of-band mechanism that is triggered prior to main processing and expands special kind of expressions: **macros.** Easily done with Python's **string substitution**/**concatenation** and **set of helper routines** producing text output per the parameters.

```
ResourceSpec('ocf:heartbeat:Ipaddr2').xsl_attrs maps to:
<xsl:attribute name="class">ocf</xsl:attribute>
<xsl:attribute name="provider">heartbeat</xsl:attribute>
<xsl:attribute name="type">IPaddr2</xsl:attribute>

xslt_is_member('name()', ('service', 'vm')) maps to:
contains(concat('|service','|vm','|'), concat('||', name(), '||'))
```
We need to somewhat **linearize the tree of resource groups** because regular/canonical structures is what we want to work with.

Embedded helper (binary) `ccs_flatten` does this for us in the same way `rgmanager` crunches the configuration for itself (**code reuse**).

To fit in default values for the parameters, validate permitted resource nesting, etc., **metadata** for `(rgmanager's)` resource agents **required**.

**Problem #10:** Metatada keep changing (extending) over time and the most authoritative source of these are those extracted directly from agents installed on the system (best if it is also a target for converted configuration).

**Problem #11:** There may be no resource agents installed on the system where `clufter` is invoked at all.

**Solution #VI:** **Maintain** the metadata at `clufter` side, but prefer raw system-installed resource agents to extract metadata from if present.
Other highlights

- very **limited dependencies** (~python-lxml)
- RPM packaging:
  - python-clufter: the core library (dep. of pcs)
  - clufter-cli: `clufter` command
  - clufter-lib-*: generics for reuse in plugins
- not presented in detail:
  - user-assisted recovery
  - maintainable 3rd party (or staging) plugins support
  - bash completion
  - unit tests (Travis CI integration), ...
Final note

The framework is already here and can be (conveniently?) reused for other purposes, especially when dealing with XML formats...