Configuration Management in the PDP group at Nikhef

Andrew Pickford

HEPiX Oct 2023

Background

grid batch
system
180 machines

grid
dcache
18 machines

local batch
system
50 machines

local
dcache
20 machines

infrastructure
servers

ldap
elasticsearch
icinga
prometheus
salt
...

250+ machines total

- in 2017:
  - used quattor
  - upgrading systems to centos 7
  - update quattor or switch?
  - looked at ansible, puppet, quattor, salt
- chose salt
  - python based
  - easily extendable
  - can test to see what effect changes will have on a machine
  - reclass for storing configuration data

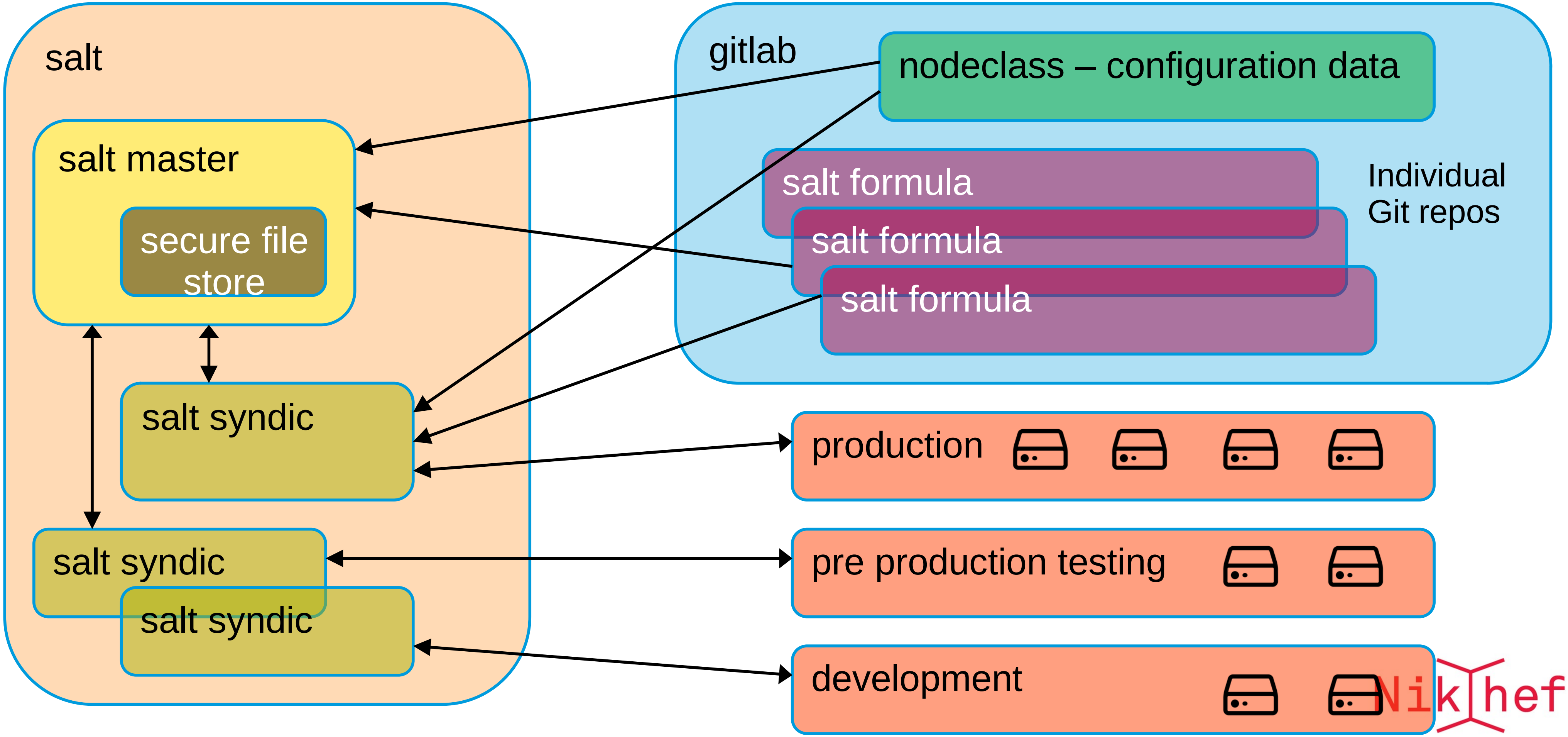
Guiding Ideas

- config system
  - well supported/documented
  - straight forward to adapt to
  - would last 10+ years
- usage
  - reuse the same configuration files between production/testing/development as much as possible
  - reduce data replication / improve consistancy
  - separate development environment with easy testing
  - test without deploying changes
  - all changes to production:
    - to be tested first
    - committed to version control before deployment

not possible with our old quattor system

lesson from the old setup, changes were deployed and the commit sometimes forgotten

Overview


salt
salt master
secure file store
salt syndic
salt syndic
salt syndic
gitlab
nodeclass – configuration data
salt formula
salt formula
salt formula
Individual Git repos
production
pre production testing
development

Git Repos I

- one git repo per salt formula
  - stored on a gitlab server
  - each formula typically manages one service
  - encodes the process of managing/configuring a service
  - handling software installation, configuration and service management
  - easy to add new formulas (easy development requirement)
  - easy to add external formulas and replace them if required
  - separates out service configuration into smaller, more managable/understandable chunks
  - production and development machines all use the same formulas (reuse requirement)

Git Repos II

- multiple branches for each repo
  - each branch maps to a salt environment
  - production + development branch for each admin (separate prod/devel requirement)
  - each machine is in one (and only one) environment
  - prod env
    - all salt masters access files via gitlab
    - all changes must be commited before being visible to production machines (commit requirement)
  - pre-prod env: checkout of prod env
    - final test of changes before moving to production (prod test requirement)
  - dev envs: access files via a git repo checkout on salt master
    - changes can tested/developed before being committed (easy development requirement)

Salt – The Good

- very flexible, highly configurable
- deploy configuration across multiple machines
- running commands across multiple machines
- test changes before deploying
- already written formulas for numerous services
  - easy to write new formulas
- easily extendable
  - written in python
  - straight forward to write new python modules
    - for services
    - for handling config data
  - override core modules while waiting for fixes in releases

In common with pretty much all modern configuration managers

```
diff:
  @@ -13,7 +13,7 @@
  pnfsmanager.enable.acl = true
  pnfsmanager.limits.list-chunk-size = 1000
  pnfsmanager.limits.list-threads = 24
  -    pnfsmanager.limits.threads = 8
  +    pnfsmanager.limits.threads = 32
```

https://github.com/saltstack-formulas
https://github.com/salt-formulas

Early on we added modules for torque and maui

Added a secure file store module for x509 key files and later an encrypted passwords module

Salt – The Bad

- lots of moving parts
  - how configuration data translates to changes on a machine it not always clear
- multiple ways of doing to same thing
  - different formulas solve similar problems in different ways
  - increases the knowledge required to use the system
- machines with 100s of states takes 10 minutes to run a deploy
- very slow with 100s of client machines
  - deploys to all production used to take several hours
  - started to plan days around deploys to production
- some error messages miss usefull information →

```
2023-02-27 17:37:03,061
[salt.utils.decorators:717 ]
[WARNING ][58006] The function
"module.run" is using its
deprecated version and will
expire in version "Phosphorus".
```

Salt - Mitigations

- python 3.11
  - significant speed improvements over python 3.6 (standard centos 7/rocky 8 python 3 version)
- syndics – move the majority of the cpu load onto dedicated machines
  - ex worker nodes, 24 cores
  - only cached data on syndics – easy to replace
  - one syndic for production machines
- full deploy to all production now take 20 minutes
- export data shared to syndic via stunnel encrypted nfs
  - more parts, more complexity - but reliable

Nodeclass I

- Used to store data describing a machine
  - organised into yaml files, each file called a class
  - classes can:
    - include other classes
    - define parameter values
    - reference other values
    - add to list of salt formulas to run
  - values definied multiply are merged together
    - scalars simply replace the previous value
    - lists are appended
    - dictionaries are merged together recursively
  - parameters become the salt pillar data for a machine

```
# role/server/bdii/init.yaml
classes:
  - service.bdii

parameters:
  bdii:
    ram_disk:
      enabled: true
    site_name: ${_cluster_:bdii:site_name}
    rootpw: ${_cluster_:bdii:rootpw}
    zone: ${_cluster_:bdii_zone}
```

```
# service/bdii/init.yml
applications:
  - bdii

parameters:
  bdii:
    enabled: true
```

nodeclass is our in house much extended and rewritten version of reclass

```
original reclass: https://reclass.pantsfullofunix.net/index.html
nodeclass: https://github.com/AndrewPickford/nodeclass
```

Nodeclass II

- node file
  - where nodeclass starts when evaluating the data for a machine
  - can include/define/reference values as a class
  - also defines the environment a machine is in
- our conventions:
  - node files include four classes: a hardware, os, role and cluster class describing the machine
    - these classes are hierarchies, each including the class above it until the top of hierarchy is reached
  - allows some mixing and matching of machine types
  - allows prod,dev and testing machines to use the same role, os and hardware classes (reuse requirement)

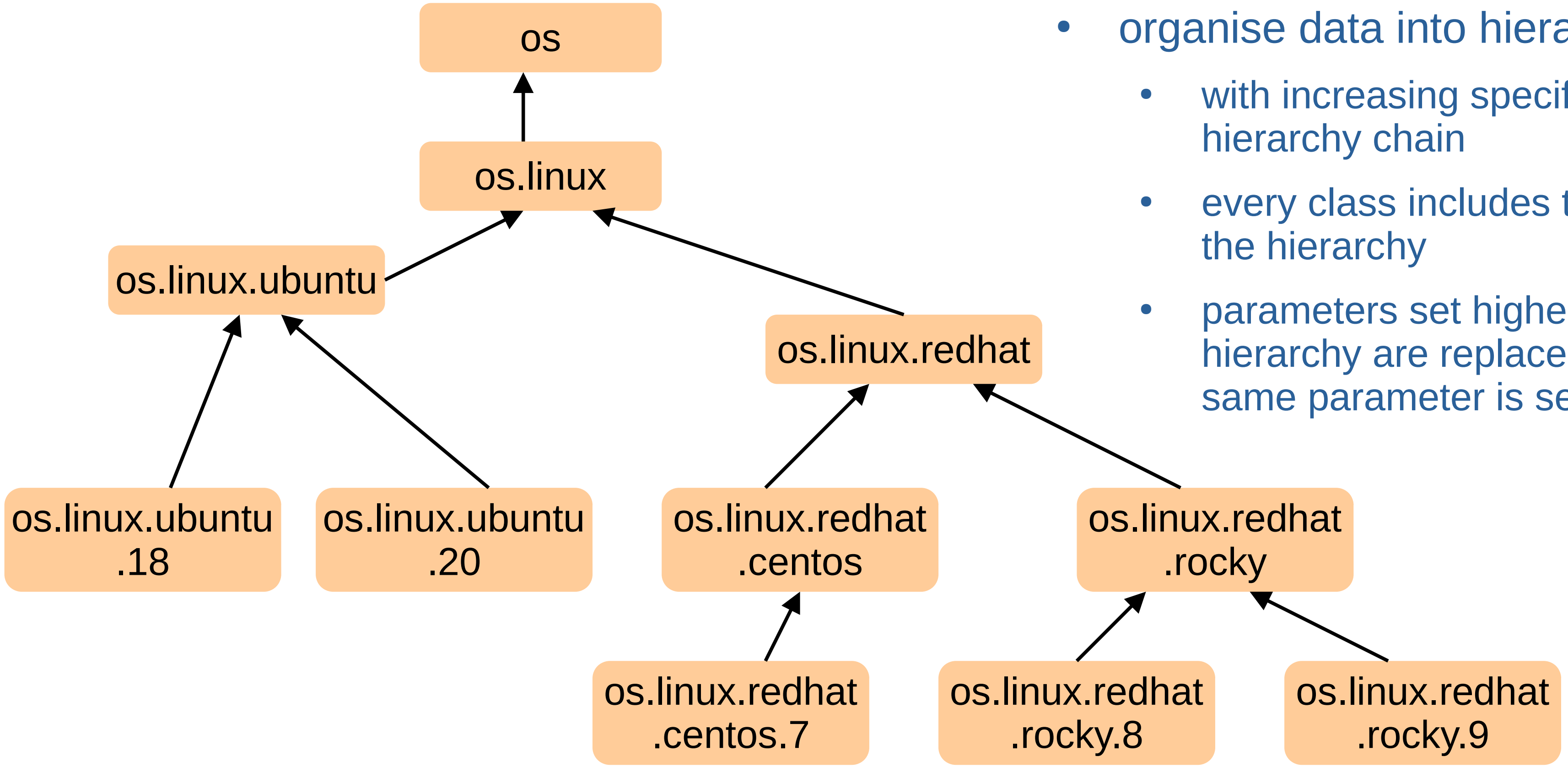
```
# dev elasticsearch machine node file
classes:
 - cluster.ndpf.andrewp.elasticsearch
 - hardware.vm.xen.storage
 - os.linux.redhat.centos.7
 - role.server.elasticsearch.cluster.universal

environment: andrewp

parameters:
 _hardware_:
  network_interfaces:
   eth0:
    mac_address: "aa:bb:cc:dd:ee:ff"
    address: "AAA.BBB.CCC.DDD"
```

```
# service/torque/server/init.yaml
parameters:
 torque:
  server:
   settings: ${_cluster_:torque:settings}
   queues: ${_cluster_:torque:queues}
```

Nodeclass Hierarchies I



- organise data into hierarchies
  - with increasing specificity down the hierarchy chain
  - every class includes the class above in the hierarchy
  - parameters set higher up in the hierarchy are replaced/added to if the same parameter is set lower down

Nodeclass Hierarchies II

- The Good
  - prod/pre-prod/dev machines as similar as possible
  - minimal unexpected changes moving from testing to production
  - one change effects all required machines
  - minimal data repetition
- The Bad
  - increases interconnectedness in the data structure
  - finding the correct place for a value can be difficult
    - there may not be one correct place
    - finding values later can take some searching
  - changes to one system can cause unexpected changes on other system
    - not necessarily a bad, as changes may be required for consistancy

```
# local dcache production machine
classes:
- cluster.ndpf.opn.dcache-stoomboot
- hardware.vm.xen.standard
- os.linux.redhat.centos.7
- role.server.dcache.ha.local.admin
```

only need to change
cluster class from one
dcache to another

Nodeclass Inventory I

- allows for data to be shared (exported) between nodes
- improves consistancy
  - when adding a new machine, very easy to forget to add it to a list of machines to monitor
  - inventory scheme generates these types of lists automatically
- exported data is visible/usable for all other nodes
- works by defining values to export then querying which nodes export a given piece of data
- increases complexity and required knowledge to use the system
- scaling issues
  - need to generate the exports for all nodes to answer any inventory query
  - solve by caching high cpu use operations and only computing the minimum possible to get an answer

Nodeclass Inventory II

- how to handle nodes with broken config data (errors in yaml, missing references,…)?
  - in principle one broken node makes any inventory query broken as the answer for that node cannot be calculated
  - handle by making inv queries for prod nodes only query prod nodes, any errors here are errors in prod nodes and should be flagged
  - for dev nodes, just ignore inv query errors

```
# inv query example

# flag a machine running bind
exports:
  bind: true

# generate a list of machines running bind
parameter:
  bind_nodes: $[ exports:bind == true ]
```

Process Issues

- final check before deploying to production goes through a git checkout on disc
  - only one person at a time can make changes
  - accidents could, and minor ones have, happened
  - blocks other changes
- overly flexible
  - a change on a machine can be done in multiple ways
  - not always clear what the best way is – if there is a ‘best’ way
  - have to look in multiple places to see how a particular configuration is done
  - we write short flight rules describing how to do tasks
    - it’s not a bad thing to write documentation
    - but it is bad that things are not more obvious

Bad Habbits I

```
iptables:
  ipv4:
    chains:
      DCACHE-CLUSTER-INPUT:
        rules:
        - rule: '-p tcp --dport 11111 -j ACCEPT'
  ipv6:
    chains:
      DCACHE-CLUSTER-INPUT:
        rules:
        - rule: '-p tcp --dport 11111 -j ACCEPT'


nftables:
  open:
    dcache_clust:
      tcp:
      - 11111
```

- overly tight coupling between config data and a (potentially) generic service
- iptables firewall configuration is the literal text to insert into the iptables config file
  - ties the configuration to iptables
  - makes it to use other firewalls
  - have to repeat rules for both ipv4 and ipv6
- reworked this for nftables
  - firewall description is technology agnnostic now
  - reduces the amount of config data
  - adds an assumption:
    - ipv4 and ipv6 have the same access rules
  - adds a layer of complexity in how the formula translates the config data into firewall rules

Bad Habbits II

- overly complex data flow
  - mostly doing things in the config data that are best done elsewhere
  - config data is not a good place to implement logic
- activating an rpm repo and using a specific repo mirror version happend over 5 classes
  - set the repo version to use
  - set the os version
  - setup an os independent parameter with the repo version
  - copy the snapshot version into the dictionary of active repos
  - copy the active repos dictionary to be visible to the salt formula

```
_cluster_:
  repo_snapshots:
    redhat_7:
      zookeeper: 20220201
```

```
_os_:
  major_version: 7
```

```
_os_:
  repo_snapshot: ${_cluster_:repo_snapshots:redhat_${_os_:major_version}}
```

```
_system_:
  repos:
    zookeeper: ${_os_:repo_snapshot:zookeeper}
```

```
repos:
  active: ${_system_:repos}
```

Summary

- the system meets our needs
  - all changes to production go through a set of test servers
  - need to be checked into git before deployment
  - high confidence changes do what is expected
  - and test for unexpected changes
  - separate development environment for each person
- but
  - it’s more complex than we’d like
  - that’s a trade off, it could be simpler – but we would then have aother problems - more repetition or less built in consistancy or ...