FIRST IMPRESSIONS OF SALTSTACK AND RECLASS

DENNIS VAN DOK

HEPIX SPRING 2018 WORKSHOP – MADISON, WI, THURSDAY 2018-05-17

A NEW CONFIGURATION MANAGEMENT SYSTEM?

We've been using Quattor since the early DataGrid days.

Changing landscape; grid services see less innovation, new CM systems emerged along with growing cloud deployments.

If there ever was a moment to do it, this was it!

Speaker notes

Disclaimer: these “speaker notes” reflect the intent of the talk I meant to give, and in no way can be guaranteed to match the delivery. I haven't got them memorised. But I provide them gladly to the reader who wishes a little more context than what is usually found on the slides.

I took over as system administrator for the Grid systems at Nikhef in 2013. By then, we had been using Quattor for many years, which had grown up alongside the Grid since DataGrid days.

Back then, long before I got my start at Nikhef in 2005, the local grid cluster consisted of a stack of upright beige boxes. The number of services quickly grew and rack mounted hardware was introduced. Around 2008, virtualisation made its debut and most non-computational services went virtual.

The need for a configuration management system that could manage all these services was clear. Systems like cfengine had been around for a while, but few sites were out there operating at the scale of a typical grid site.

This changed with the arrival of *the cloud*, which allowed companies to scale out aggressively based on demand. Their engineers needed control over rapid deployments, and a slew of systems entered the stage.

The Grid stopped growing and innovating. We've consolidated the services that we are running and we will keep them running a while longer, but we're deploying other services to match user's needs.

If we ever wanted to try out a new configuration management system, this was the right moment.

ABOUT THIS TALK

- not a technical talk
- the journey is more interesting than the destination
- we're got plenty of the road ahead of us

Speaker notes

This is a really large topic, so the material presented here will have to be limited to what we've found along the way, leaving most technicalities aside. But I'm a technician and I like to talk about technology, so I will have to really try to keep it to the bare minimum.

With systems like these, the end goal is pretty easy to understand. It's the getting there that counts. And we're not there yet.

A NEW SYSTEM!

Credits to Andrew Pickford!

Looked at quattor upgrade:

- a lot of work
- smallness of quattor community
  - they certainly wanted to help
  - not easy to get going based on available documentation

Speaker notes

Much of the work I present here is really done by Andrew Pickford, who joined the team in 2015. He's become a familiar face with the dCache community in the mean time, as we've switched from using Glusterfs to dCache for the shared storage filesystem on our local torque cluster, a relatively new use case for dCache that required some interaction with the developers.

We discussed the CM conundrum a lot and came to the conclusion that we should consider a Quattor upgrade first, since the version we were on was antiquated and we could certainly benefit from the input of the Quattor community.

It turned out that we were facing a tall order, whether we would pick the upgrade path or switching wholesale to a new system. Andrew gave it an honest attempt, but ran up against a shortage of documentation and things just not working as advertised.

It dawned on us that Quattor's reduced community would be of little benefit and we made the strategic choice to go for an all new system.

CONSIDERING SEVERAL ALTERNATIVES

(But some were rejected outright based on personal prejudice.)

An honest comparison would have been too much work.

Two candidates came very close: **Saltstack** and **Ansible** with no obvious winner.

Saltstack came out ahead by a nose on technicalities.

(Ansible would have served us just fine.)

Speaker notes

Making that choice was perhaps the hardest part of the endeavour; taking that first step out the door. The second hard choice was: which system would we adopt? There were many candidates, which made the choice a lot harder.

We discussed the idea of testing multiple systems, but rejected this because a fair comparison would have meant running each system in earnest. We simply did not have the time or resources for that.

Some personal misgivings about systems (more based on *smell* than technical merit) helped us to whittle down the shortlist to a mere two candidates: Ansible and Saltstack.

Apologies if I'm mixing sports metaphors here, but we really needed a jury decision to decide on the winner.

A good point to make here is that all or most of the systems we considered would have been able to do the job, given we would spend enough effort to tune it.

WHAT WE LIKED

(Based on previous experiences)

- we really liked the state concept of Saltstack (similar to Quattor).
- Everything is YAML and Python. (And, ok, Jinja2.)
- Nice integration with Reclass (more later).
- Test mode shows what *would* change.

Speaker notes

Among the advantages that helped Saltstack edge out the competition were its sound state based approach (this struck home with us at it shared that philosophy with Quattor); the fact that it's all written in Python, a language we were immediately comfortable with, and uses YAML as a simple specification language.

We really liked the test mode which helpfully indicates what the differences are between the current state of a node compared to the defined state.

Finally an interesting piece of software called Reclass provided a really nice approach to laying out the system data in a hierarchical manner that feels really natural—to a system administrator that is.

A FIRST LOOK AT SALTSTACK

Discussed (a bit) at HEPiX before.

- 2016, Sandy Philpott, Site report,
  https://indico.cern.ch/event/531810/contributions/2314173/
- 2017, Owen Synge, Technical talk,
  https://indico.cern.ch/event/595396/contributions/2544138/

Widely used in various open source communities.

Speaker notes

Although Saltstack has been growing a sizeable community, I've heard of few adoptions by the HEPiX community. Perhaps we will be the first to try this at a serious scale (although we are not a very large operation by today's standards). As far as I know there is a modest deployment at JLab, and Owen gave a talk as an independent.

I'm not going to discuss the ins and outs of Saltstack itself, but it is a large and interesting topic in its own right. I'm happy to discuss this during the breaks.

THIS IS NOT A TECHNICAL TALK

(But anyway…)

- master/minion system
- minions controlled by defined **states**
- static data provided by **pillars**
- states are logically bundled by **formulas**
- states are implicitly **ordered** by dependencies

Speaker notes

In a nutshell, this master/minion system compiles *state* data for the minions to process. A ‘State’ describes a single piece of the configuration of a machine, such as the presence of a particular line in `/etc/lvm/lvm.conf`, or a software package, or a user account.

States are *idempotent* which means running multiple times has the same effect as running once.

A bundle of states to configure a single service is called a formula. Much of what a formula will do is based on information provided about the node in the form of pillar data. We'll see some examples later on. A simple way to think about it is that the pillar is a collection of configuration variables and that the formula is a configuration script.

Saltstack knows various types of dependencies which cause the running of states to be ordered, and some states to be skipped entirely. For instance, a configuration file change may cause a service restart, but if the file is untouched the service is left running.

WHAT GOES WHERE

data source	kind of data	typical examples
pillar	static per-node	server name, ip address
formula	states related to a single aspect	mysql, iptables
state	elementary settings	installed packages, running services

Speaker notes

This table provides just a very simple view of the distinction between these central concepts. Keep this in mind in the discussion further on because it is easy to get these things muddled.

EXAMPLE OF STATE RUN IN TEST MODE

```
      ID: /etc/nova/nova.conf
Function: file.managed
  Result: None
 Comment: The file /etc/nova/nova.conf is set to be changed
 Started: 15:37:01.083553
Duration: 380.062 ms
 Changes:
          ----------
          diff:
              ---
              +++
              @@ -158,6 +158,7 @@
               # * ``hyperv.HyperVDriver``
               #  (string value)
               #compute_driver=<None>
              +compute_driver=libvirt.LibvirtDriver

               #
               # Allow destination machine to match source for resize. Useful when
```

Speaker notes

The output (nicely highlighted on a color terminal) shows part of a test run of bringing a node to its intended state. I intentionally removed a line from the config te show that the state would put that line right back in. Instead of showing the entire file it only outputs the diff.

```
----------
          ID: /etc/nova/nova.conf
    Function: file.managed
      Result: None
     Comment: The file /etc/nova/nova.conf is set to be changed
     Started: 15:37:01.083553
    Duration: 380.062 ms
     Changes:
              ----------
              diff:
                  ---
                  +++
                  @@ -158,6 +158,7 @@
                   # * ``hyperv.HyperVDriver``
                   #  (string value)
                   #compute_driver=<None>
                  +compute_driver=libvirt.LibvirtDriver

                   #
                   # Allow destination machine to match source for resize. …
----------
```

ORGANISING OUR DATA WITH RECLASS

We separated the

- moving parts (*states*) that are the same for all our nodes from the
- static data specific to each node (pillar).

The pillar is provided by Reclass.

Speaker notes

We can distinguish two main parts of the machinery: analogue to the configuration *scripts* and the configuration *data*. In the Saltstack vernacular called states and pillar, respectively.

RECLASS

A recursive classifier, collecting static hierarchical information about nodes providing *pillar* data.

Originally http://reclass.pantsfullofunix.net/, but the most active fork at the moment is https://github.com/salt-formulas/reclass/. Our version currently is https://github.com/AndrewPickford/reclass/.

Speaker notes

When you start to write configuration data for many nodes, you quickly realise that there is a lot of overlap between the nodes and that you'd rather avoid the duplication of data. The second realisation is that when you try to collect commonalities, there are subtle differences to be dealt with that cause a real headache. Soon, your data will be in tangles, and you are unhappily pondering whether it is too late to switch careers to herding cats.

The Reclass software offers some relief because it allows the specification of data in a tree of inherited classes. This means that each piece of information finds its natural place in the hierarchy, at the right level between generic and specific. It means that default values can be set at the top of the hierarchy and be overridden further down towards the leaves.

Reclass also allows referencing elements form other branches of the inheritance tree, and with late evaluation, the final values end up being used.

RECLASS IN A NUTSHELL

(Remember, not a technical talk!)

- Each node specifies which *classes* it belongs to;
- each class is a file in a hierarchy (i.e. directory structure);
- each class file lists more classes and/or parameters;
- later classes override (simple values) or merge (lists) values from earlier classes.

Speaker notes

The classes are written as YAML files in a directory structure. Each class file contains further classes to inherit and configuration variables called parameters. These are themselves structured, e.g.

```
_hardware_:network_interfaces:eth0:use_gateway: true
```

The final values can be simple scalars or lists or references.

RECLASS EXAMPLE

Example, slightly simplified. This is a dCache master node in our testbed.

```
classes:
  - cluster.ndpf.testbed.dcache
  - hardware.vm.xen.standard
  - os.linux.redhat.centos.7
  - role.server.dcache.plain.master
environment: pre-prod
parameters:
  _hardware_: (here be the VM provisioning parameters)
```

Speaker notes

This is as near as can be the full file. The hardware data lists stuff like amount of memory, VM pool name, MAC address, and ip addresses. There are four classes from different hierarchies; we've made this separation intentionally to separate the hardware descriptions from the collection of nodes which form a cluster and have a common environment, the operating system and the role of the system. As a rule, we try to stick to these hierarchies when specifying data, but there are some items that escape the labelling and end up in the catch-all system hierarchy.

We use the environment concept to group systems by areas of responsibility. There is production, pre-production and personal environments for testing and development purposes. These can still share much of the same Reclass data.

Nikhef

here is cluster/ndpf/testbed/dcache/init.yml:

```
classes:
  - cluster.ndpf.testbed
parameters:
  _cluster_:
    name: dcache testbed
    dcache_version: 3.1
    dcache_carbon_server: ${_cluster_:monitoring_satellite}
    dcache_nfs_allowed_ipv4:
      - ${_site_:networks:ipv4:stbcnet}
      - ${_site_:networks:ipv4:wnnet}
```

Speaker notes

Note the `_cluster_` namespace. By our own rule, classes in the cluster hierarchy should only set parameters in this namespace. The dollar-curly notation references a parameter set elsewhere, and this should *eventually* be set somewhere (evaluation happens late).

Another self-imposed rule is that classes always inherit up the chain, so that is why you see cluster.ndpf.testbed and this will include cluster.ndpf and so on.

cluster/ndpf/testbed/init.yml:

```
classes:
  - cluster.ndpf
parameters:
  _cluster_:
    name: testbed
    monitoring_satellite: vaars-03.nikhef.nl
```

Note that `_cluster_:name` is given here, but the class cluster.ndpf.testbed.dcache overrides it.

Speaker notes

This is just a part of the entire file, but this portion serves the example.

Here we see how `monitoring_satellite` obtains its value. And the `_cluster_:name` value is overridden in the more specific class.

WHAT DATA GOES WHERE

- Reclass allows more freedom in layout of data
- Following a logical structure rather than what is imposed by a system
- Only simple constructs allowed; complicated programming relegated to states

Speaker notes

Reclass allows us to order our data in way that is following a logical structure of our own design, free and independent from anything the system would impose on us.

Since reclass is not a Turing complete programming language, certain constructs are not possible. That is by design; it forces an overall simplicity. It is just an inventory.

The more powerful programmatic constructs are relegated to the states, i.e. the formulas.

SHORTCOMINGS

Reclass is not without its shortcomings. It needed work to make it do what we wanted, and was (therefore) almost rejected.

We still went ahead and fixed it.

Speaker notes

Reclass was not exactly polished and finished, but it was a good start. The first rule of programming is: *don't.* So we broke that rule and now we maintain a portion of Reclass.

REDEEMING QUALITIES

Written in python which is nice and forgiving to programmers.

Our patches are available on Github, and we're looking to integrate with versions maintained by the salt-formulas people.

Speaker notes

Because it's written in Python (like Saltstack!) we had little trouble hacking away at Reclass. Luckily others seem interested in the work that was done.

ADDED FEATURES

Exports

allow extraction of info from other nodes. This is conceptually related to the *salt mine* but comes in at an earlier stage of the processing chain.

References

were enhanced to allow nesting; overriding values will do merge instead of replace when values are lists or dicts.

Git backend

works just like the git backend for Salt, so data is taken straight from a repository/branch.

Speaker notes

Here are a couple of features we added to the original implementation.

There is a new section that can go in each class file called *exports*. The properties listed here are made available for inspection across nodes by means of an *inventory query*. The query language is rather primitive, but rich enough to populate our installation (cobbler) node and the monitoring systems.

References were already part of reclass, but the enhancement improved the merging behaviour and it is now possible to use nested references. (Whether that is wise is another matter.)

Originally, reclass looks for classes in a file system tree; Since Saltstack can use a git backend we gave reclass the same capability.

IMPROVED ERROR HANDLING AND REPORTING.

```
    - Failed to load ext_pillar reclass: ext_pillar.reclass: →
…-> cc2.cloud.ipmi.nikhef.nl
         Cannot resolve ${_cluster_:some:value}, at →
…_cluster_:monitoring_satellite, →
…in yaml_fs:///srv/salt/env/dennisvd/classes/cluster/ndpf/cloud/init.yml
```

Speaker notes

The lines in this example are wrapped to fit the width of the screen, but with some effort one can make out the hint. The error is completely intentional of course, but this illustrates that reclass will point to the origin of the problem, which (believe it or not) originally it could not do.

FORMULAS

All the moving parts are grouped by formulas.

apache, authconfig, autofs, backupninja, bind, certificates, cinder, cobbler, contrailctl, cups, cvmfs, dcache, dell_mdsm, docker, elasticsearch, eos, galera, git, glance, grafana, graphite, grid, haproxy, hardware, horizon, icinga, iptables, keepalived, kerberos, keystone, kibana, linux, logrotate, logstash, maui, memcached, munge, mysql, neutron, nfs, nikhef, nova, ntp, pacemaker, pakiti, php, postfix, postgresql, prometheus, python, rabbitmq, reclass, repo-mirrors, rsync, rsyslog, salt, sanity-check, secure, tftpd_hpa, torque, zookeeper

Speaker notes

This is a list of formulas we are currently using. Roughly half of them are of our own making, or adaptations of existing formulas that needed too much work to keep calling it by the same name. The other half are community formulas that we forked.

PROS AND CONS

Pros:

- encapsulate a functional element
- forms a clear conceptual boundary
- places complexity where we want to handle it

Cons:

- many repositories (requires scripting)
- mixed quality (often only tested on Debian)

Speaker notes

The choice to go with formulas was one of the easiest to make, It has been picked up by the community very readily as just a Very Good Idea™.

SINGLE OR SEPARATE REPOSITORIES?

Choice:

- put all formulas in a single repository, or
- keep all formulas in their own repository

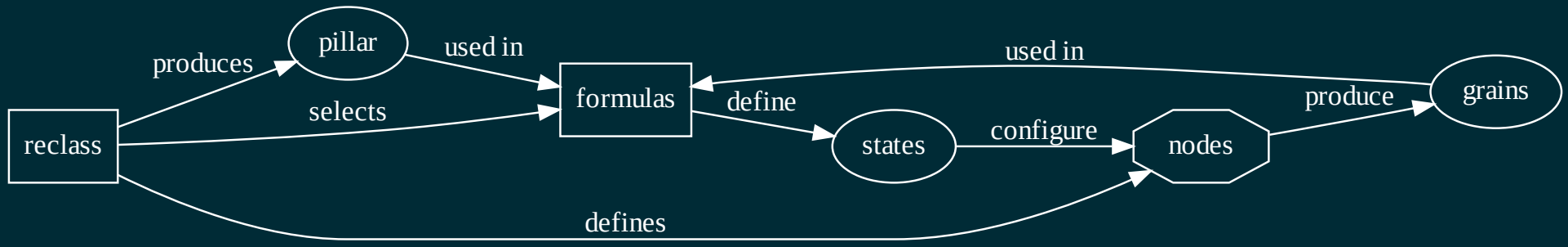
Speaker notes

We struggled with the integration of formulas in our system. Incorporating all the formulas in a single git repository would have simplified version control, but it would have made it harder to exchange enhancements with other users and developers.

FORMULAS AND RECLASS

- Formulas are driven by pillar data
- This makes them integrate well with reclass.

INFORMATION FLOW AND RELATIONSHIPS



Speaker notes

This scheme shows the relationships between the elements discussed so far. What is clear from this picture is how reclass serves as a source for most of the data.

We did not discuss grains in great detail, but they are not a very important part of the equation. Grains offer live data from the node, but the formulas only use the operating system to determine names of packages and such.

(What is left out of the picture are deployment and orchestration.)

Nikhef

VERSION CONTROL

- keep everything in private Gitlab
- master branch in Gitlab **defines** what is in **production**
- other branches correspond to environments

Speaker notes

Naturally all our work is kept under version control, and naturally we chose to use git. Since the data contains some sensitive elements we cannot use a public service like Github, so we set up our own private Gitlab server.

Thinking through the version control/configuration management link, the natural question to ask is: what is the correspondence between the version control system and the current production configuration?

We settled on the rule that production should look at the master branch in Gitlab.

GIT AS A WORKFLOW DRIVER

- git push to master determines what is in production
- manual deploy initiated thereafter still necessary
- we needed a pre-production testbed to test changes before the push
- we needed a way to sync up the many formula repositories

Speaker notes

Once the above rule was settled, many workflow concepts followed as corollaries. Updates to production are traced to a commit that is pushed to the master branch. This should be followed by a deploy to all production nodes.

Because there is no light shining through the cracks anymore, we hare no opportunity here to do any late-stage testing before a true deployment. This called for a pre-production system based on the same data and as near as can be identical to production (albeit of a lesser scale).

Because of the multitude of formula repositories, we also required some scripting to synchronise commits across all of these.

PRE-PRODUCTION

- Each type of system has its counterpart in pre-production.
- Pre-production looks at a local checked out version of the master branch.
- Variants for treating updates:
  - minor changes can be applied and tested before committing
  - major updates are tested in other environments and handled via git merging of branches

Speaker notes

The pre-production system is based on a local check-out, which opens some opportunity for running tests before committing and pushing. At least for production there must be an associated commit.

PEPPER WRAPPER

High level pepper scripts to replace low level salt.

- dealing with multiple repositories
- test
- deploy
- commit
- other git commands

```
Pepper-deploy
```

will stagger updates to prevent overload on the master.

Speaker notes

With our workflow and multiplicity of repositories, the low-level commands were inconvenient. The pepper scripts take care of all more usual cases. It is a wrapper around git, as well as wrapper for salt commands to test and deploy.

Because the master takes a performance hit when all nodes updated simultaneously, the scripts will stagger the updates, doing a handful at a time.

ENVIRONMENTS

Environments correspond to branches in git.

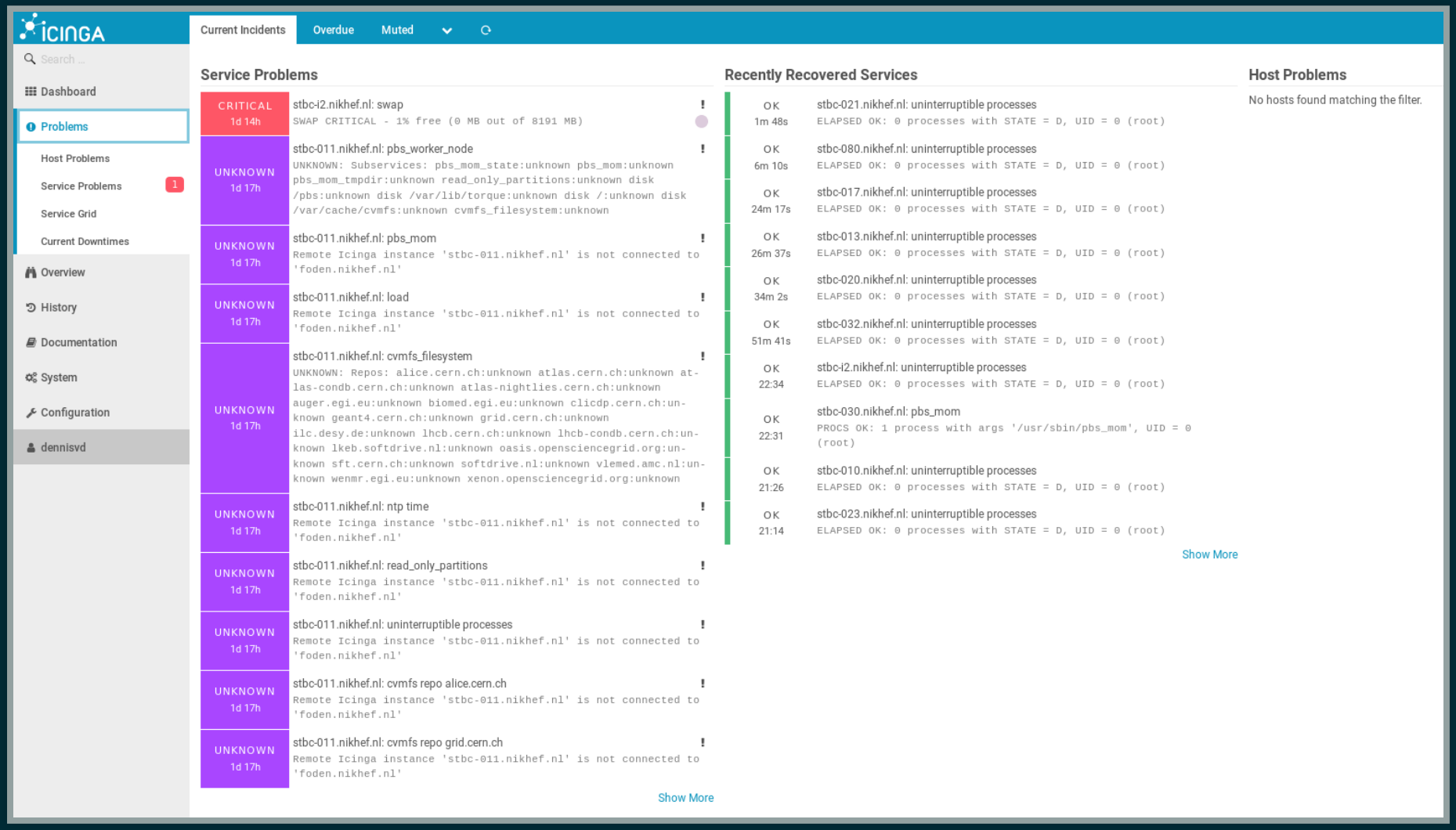
- Each newly introduced formula must have branches for every environment.
- Pre-production is the exception, because it looks at the master branch (but actually a local checkout).
- People have their 'own' environment for testing and development purposes.
- possibility to ‘move’ a machine between environments

Speaker notes

The concept of environments is native to Reclass. In Saltstack an environment is defined by a series of file roots (or git urls).

The environment of a node is defined in the node reclass file.

MONITORING


icinga
Current Incidents
Overdue
Muted
Search ...
Dashboard
Problems
Host Problems
Service Problems 1
Service Grid
Current Downtimes
Overview
History
Documentation
System
Configuration
dennisvd
Service Problems
CRITICAL 1d 14h
stbc-i2.nikhef.nl: swap
SWAP CRITICAL - 1% free (0 MB out of 8191 MB)
UNKNOWN 1d 17h
stbc-011.nikhef.nl: pbs_worker_node
UNKNOWN: Subservices: pbs_mom_state:unknown pbs_mom:unknown pbs_mom_tmpdir:unknown read_only_partitions:unknown disk /pbs:unknown disk /var/lib/torque:unknown disk /:unknown disk /var/cache/cvmfs:unknown cvmfs_filesystem:unknown
stbc-011.nikhef.nl: pbs_mom
Remote Icinga instance 'stbc-011.nikhef.nl' is not connected to 'foden.nikhef.nl'
stbc-011.nikhef.nl: load
Remote Icinga instance 'stbc-011.nikhef.nl' is not connected to 'foden.nikhef.nl'
stbc-011.nikhef.nl: cvmfs_filesystem
UNKNOWN: Repos: alice.cern.ch:unknown atlas.cern.ch:unknown atlas-condb.cern.ch:unknown atlas-nightlies.cern.ch:unknown auger.egi.eu:unknown biomed.egi.eu:unknown clicdp.cern.ch:unknown geant4.cern.ch:unknown grid.cern.ch:unknown ilc.desy.de:unknown lhcb.cern.ch:unknown lhcb-condb.cern.ch:unknown lkeb.softdrive.nl:unknown oasis.opensciencegrid.org:unknown sft.cern.ch:unknown softdrive.nl:unknown vlemed.amc.nl:unknown wenmr.egi.eu:unknown xenon.opensciencegrid.org:unknown
stbc-011.nikhef.nl: ntp time
Remote Icinga instance 'stbc-011.nikhef.nl' is not connected to 'foden.nikhef.nl'
stbc-011.nikhef.nl: read_only_partitions
Remote Icinga instance 'stbc-011.nikhef.nl' is not connected to 'foden.nikhef.nl'
stbc-011.nikhef.nl: uninterruptible processes
Remote Icinga instance 'stbc-011.nikhef.nl' is not connected to 'foden.nikhef.nl'
stbc-011.nikhef.nl: cvmfs repo alice.cern.ch
Remote Icinga instance 'stbc-011.nikhef.nl' is not connected to 'foden.nikhef.nl'
stbc-011.nikhef.nl: cvmfs repo grid.cern.ch
Remote Icinga instance 'stbc-011.nikhef.nl' is not connected to 'foden.nikhef.nl'
Show More
Recently Recovered Services
OK 1m 48s
stbc-021.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
OK 6m 10s
stbc-080.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
OK 24m 17s
stbc-017.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
OK 26m 37s
stbc-013.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
OK 34m 2s
stbc-020.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
OK 51m 41s
stbc-032.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
OK 22:34
stbc-i2.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
OK 22:31
stbc-030.nikhef.nl: pbs_mom
PROCS OK: 1 process with args '/usr/sbin/pbs_mom', UID = 0 (root)
OK 21:26
stbc-010.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
OK 21:14
stbc-023.nikhef.nl: uninterruptible processes
ELAPSED OK: 0 processes with STATE = D, UID = 0 (root)
Show More
Host Problems
No hosts found matching the filter.

Speaker notes

This is a screenshot of our Icinga monitoring system, showing nothing particularly interesting.

- Relies on the exports mechanism discussed earlier
- Nodes specify
  - what type of thing they are, and
  - the kinds of things anyone interested in monitoring should be looking for.

The monitoring system defines how the actual monitoring is done for all of those things. It gets the list of nodes and services from the inventory.

Speaker notes

The monitoring of nodes has to know which nodes to monitor, and what to monitor for each node. The best place to put this information is in the service specific data in reclass.

To extract this information we use the new export mechanism. An inventory query will retrieve all the relevant bits from the inventory.

Reclass lists only what should be monitored, but the icinga formula details how this should be done. So the apache role would say: ”I'm a web server if you care to monitor me,” but the icinga formula then needs to understand what a web server is and how to monitor that.

DEPLOYMENT

- cobbler
- based on exports.
- supported by scripts
- hardware description of a node
  - prescriptive for VMs
  - descriptive for actual hardware

The cobbler node has to manage both production and pre-production, and is the 'odd one out' as it has no pre-production counterpart.

Speaker notes

Deployment of nodes with Cobbler means that the cobbler server has to know all the machines across all the environments. There is a special inventory query for that.

For virtual machines, the deployment scripts will talk to our virtualisation platform to create a machine with a recipe for the desired components. PXE boot and cobbler take it from there.

For real hardware, we set to PXE boot by default (except storage machines, because we are afraid of accidental reinstalls).

REPOSITORIES

The cobbler server also collects mirrors of various repositories for software installation.

- time-based snapshots
- no dependencies on external repositories in production
- support for both apt and yum repos

Speaker notes

The repository management is a home-grown solution. We manually create time-based snapshots by hard-linking the mirror directory, optionally running *createrepo* before setting it to read-only. This makes a snapshot very cheap and practically immutable.

A reference to the correct snapshot is made in the reclass data.

SYSTEMS SALTIFIED SO FAR

- dcache
- salt master
- cobbler
- torque/maui (local cluster)
- DNS (in high availability setup)
- monitoring (grafana, icinga)
- NFS server
- EOS
- Openstack (still experimentally)
- more to come

Speaker notes

There were quite a few systems to integrate as part of the core infrastructure to set up before any real systems could be tackled. But it's picking up speed now.

CONCLUSIONS

OPEN PROBLEMS

- Running the inventory with 'broken' nodes
- Performance issues with large deployments

Speaker notes

There are some sticky issues we eventually will have to deal with.

- limit exports only to machines in the same environment? Except cobbler with must have defs for all nodes. Simply ignore 'failing' nodes.

The load on the master gets very high when many minions simultaneously want to run all of their states. While we have a workaround in place where updates are staggered by bunching system together 5 at a time (for some value of 5), the proper solution would be to find out where the system is spending its time and trying to optimise or eliminate.

FUTURE

- full automated installations
- pre-provisioning keys (salt, ssh, others)
- orchestration
  - stagger kernel updates
- multi-master
- performance issues
  - where does the system spend most of its time?
  - high load on master
  - addressed by batching updates with pepper scripts
  - the monitoring box will go to 500+ states as we add more systems

Speaker notes

We've made a lot of progress already and we're still learning what this system can and cannot do. Part of large-scale deployments is the ability to automate every aspect, starting with the fully automated installation. This also means that the salt keys will have to be somehow pre-arranged or automatically accepted.

We're looking into the orchestration capabilities to see how we can exploit them. For instance for a staggered update of a new kernel instead of all at onec.

And we're taking the performance issues very seriously.

LESSONS LEARNED

- New system is a lot of work.
- Organisation of data is more important than mechanics.
- Tradeoff between flexibility in prototying and control in production.
- No truly bad choices, but many secondary factors to consider.
- Look at the specific needs of the team; better find a good match than just go with the most popular system.

Speaker notes

We've made a lot of progress setting up an new system. Setting up the core infrastructure cost the most time.

When starting with a new system, there are probably no truly bad choices. But secondary factors play a role in making the right choice.

We spent a lot of time discussing about organising our data, and that turned out to be very important to us; this in the end is what we'll end up maintaining.

No matter which way you go, there seems always to be this tradeoff between control over production and flexibility in development and prototyping. Figuring out what this balance should be in your case is an important exercise.

And finally: the choice is entirely yours to make. Look at the needs of the team and find a proper match; this may not be what others have done but since you're going to be the one to implement it and maintain it, it is important that you are comfortable with your pick.