Saltstack Config Management

Steve Anthony

HPC Support Specialist

Lehigh University

What is Configuration Management?

A system of practices and tools which allows us to centrally manage and deploy configuration to many systems. Often the same software provides for server orchestration, enabling us to interact with system functions or services in a distributed system from a single control point.

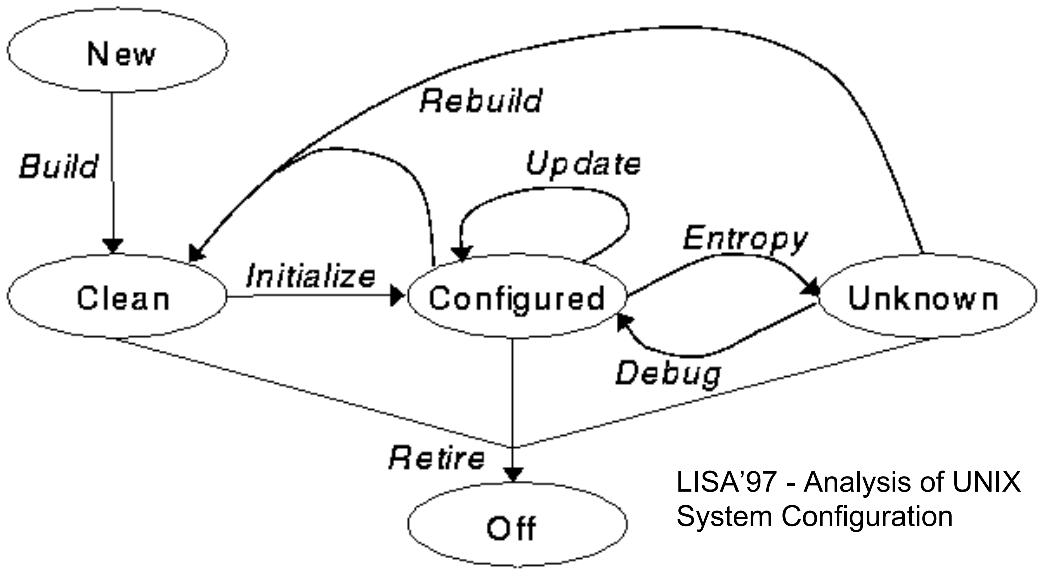
Why bother?

Lets us build repeatably identical systems.

Treat infrastructure as code.
- Version control, testing, abstraction/templating, validation.

Required to build scalable (self-scaling) systems.

Orchestration. Perform an action on many systems.



LISA'97 - Analysis of UNIX System Configuration

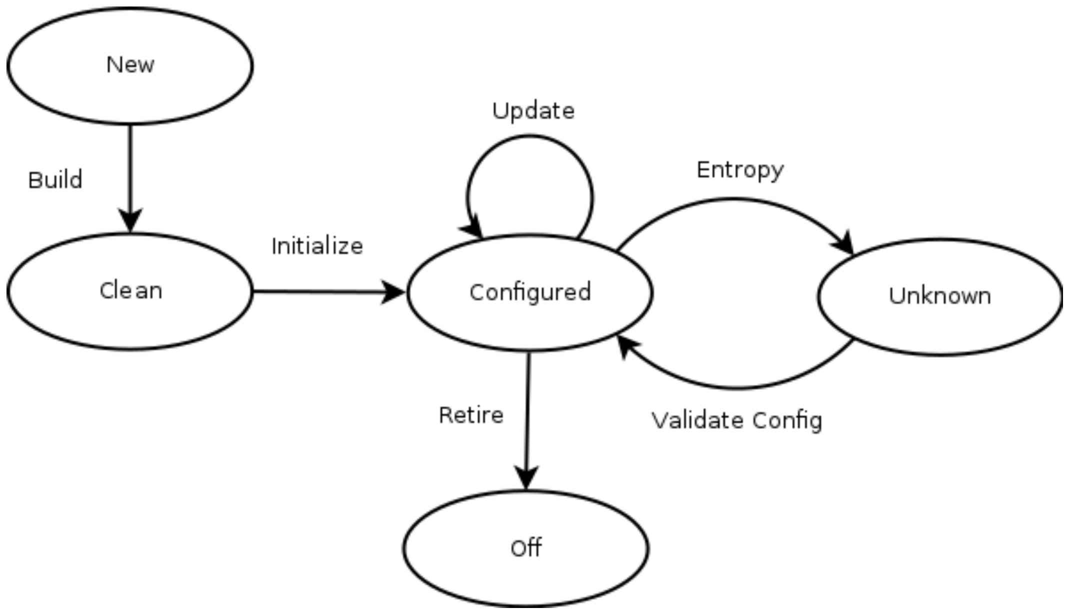
Why bother?

Lets us build repeatably identical systems.

Treat infrastructure as code. Version control, testing, abstraction/templating, validation.

Required to build scalable (self-scaling) systems.

Orchestration. Perform an action on many systems.



ANSIBLE

puppet labs

CHEF™

SALTSTACK

CFEngine

Saltstack Components

Master & Minion

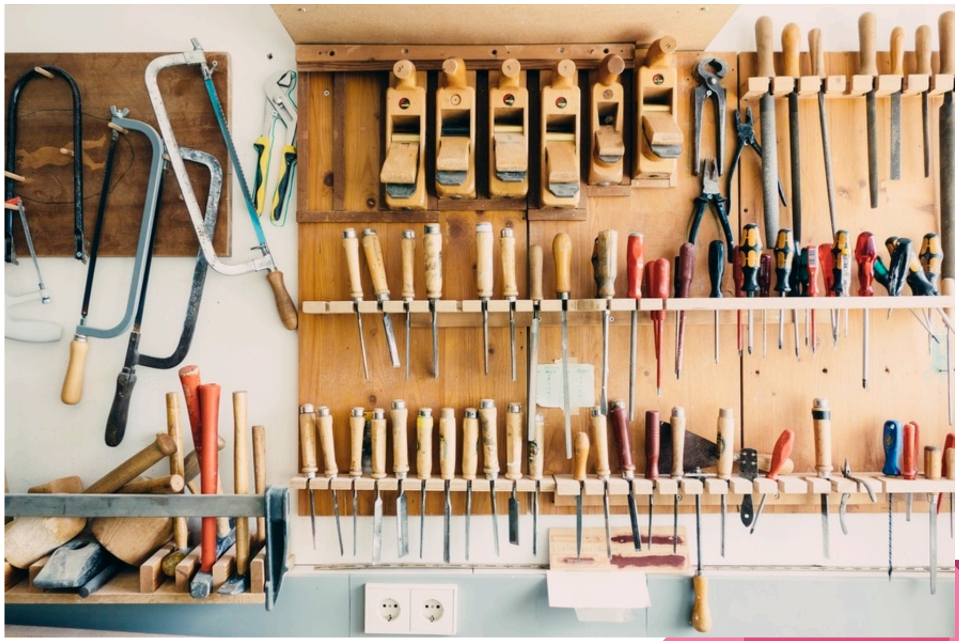
States & Modules

Nodegroup

Grains

Pillars

Reactor



States & Modules

- State files are composed of a series of state functions which enact a desired series of configuration steps.
- State and module functions are atomic units which perform an action.
  - Manage a file: contents, permissions, owner, etc.
  - Make sure a service is running.
  - Make sure a package is installed.
  - Make sure a user exists.
- Imperative and Declarative language written in YAML.
- Applied by calling the state.highstate function.

Sample State: NFS

Grains & Pillars

Grains are data points collected by the minion on each server.

Pillars are data points defined on the master and assigned to minions.

Both can be used in state files and file templates to abstract portions specific to a host or type of server.

- Ultimately expressed as a dictionary. We'll exploit this later :)
- JINJA Python templating engine is used at runtime to translate placeholders, conditionals, and flow control blocks.

Default values may be set based on a grain value, then overloaded by pillar values on per server (or group) basis.

map.jinja

```
{% set nfs = salt['grains.filter_by']({
  'Ubuntu': {
    'pkgs_server': ['nfs-common', 'nfs-kernel-server'],
    'pkgs_client': ['nfs-common', 'autofs'],
    'service_name': 'nfs-kernel-server'
  },
  'Debian': {
    'pkgs_server': ['nfs-common', 'nfs-kernel-server'],
    'pkgs_client': ['nfs-common', 'autofs'],
    'service_name': 'nfs-kernel-server'
  },
  'RedHat': {
   'pkgs_server': ['nfs-utils'],
   'pkgs_client': ['nfs-utils', 'autofs'],
   'service_name': 'nfs'
  }
}, grain='os', merge=salt['pillar.get']('nfs:lookup')) %}
```

Sample State: NFS 2.0

Reactor

ZeroMQ PUB interface on the Salt Master.

- Clients publish events to the queue on the Master.
- The Reactor system reads this events.
- Reactor then triggers actions on the master based on tags in event.

Reactor: Example

Want to add a disk logically to a Ceph storage cluster when physically inserted into a node.

- On insertion, udev rule triggers a python script to do preflight setup, eg. partitioning, finding a free journal partition on an SSD, etc.
- Publish an event to the Salt Reactor, with node hostname, hard drive identifier, and SSD partition UUID.

Python API Sample

```
import salt.client
# Fire a salt event
sock_dir='/var/run/salt/minion'
payload={'fqdn':fqdn,'host':host,'disk':disk,'journal':journal}
caller = salt.client.Caller('/etc/salt/minion')
caller.sminion.functions['event.fire_master'](payload,'ceph/addosd')
```

Reactor: Example

Reactor gets information like this

Host: ceph23

Disk: /dev/sda

Journal:/dev/disk/by-id/ata-KINGSTON_SKC300S37A60G_50026B72440139C0-part4

Tag: ceph/addosd

/etc/salt/master.d/reactor.conf

```
reactor:
  - 'ceph/addosd':
    - /srv/reactor/ceph/react_new_osd.sls
```

Reactor: Example

Reactor gets information like this

Host: ceph23

Disk: /dev/sda

Journal:/dev/disk/by-id/ata-KINGSTON_SKC300S37A60G_50026B72440139C0-part4

Tag: ceph/addosd

/srv/reactor/ceph/react_new_osd.sls

```
add_new_osd:
  cmd.cmd.run:
    - tgt: 'encephalon1.cc.lehigh.edu'
    - arg:
      - /home/ceph/kraken/salt-prepareosd.sh {{data['data']['host']}} {{data['data']['disk']}} 
{{data['data']['journal']}}
      - cwd=/home/ceph/kraken
      - runas=ceph
```

Reactor: Example

Want to add a disk logically to a Ceph storage cluster when physically inserted into a node.

- On insertion, udev rule triggers a python script to do preflight setup, eg. partitioning, finding a free journal partition on an SSD, etc.
- Publish an event to the Salt Reactor, with node hostname, hard drive identifier, and SSD partition UUID.
- Script on client decodes information and schedules disk for addition.
- When cluster is ready, disk is added.

Salt Cloud

Provision hosts on cloud hosts/hypervisors and immediately bring them under management.

Provider configuration contains provider API key/credentials and configuration global to all instances.

Profiles define templates for server instances.

- CPU/RAM/disk requirements
- Network/Security group requirements
- Base image
- Grains to set

Problem

Amazon Web Services

We want to setup an NFS server and clients in AWS. Traditionally, we'd be able to set (reserve) hostnames and IPs needed for the NFS and autofs configuration files at each end. However, AWS IPs aren't generated until the instance is created and running. How do we accomplish this?

Salt to the Rescue!

Solution Overview

Provision instance with salt-cloud. Use grains to distinguish NFS server and client. Use another grain to group workloads.

Use Salt Reactor to run a shell script on the master when a minion is added to the master. Gather IPs of server and clients we know about in this workload.

Generate a flat pillar. Execute a highstate on each client, passing in the new pillar containing IP/export information. Wait if another job is running. We could use this same approach to watch and scale a service up and down as load fluctuates.

Questions?