xCAT and Masterless Puppet: Aiming For Ideal Configuration Management

Jason St. John
Research Computing
Purdue University
West Lafayette, IN, USA
jstjohn@purdue.edu

***Abstract*—Configuration management is a major factor in the design of an HPC system. This paper provides an evaluation of the new architecture of the HPC systems operated by Purdue University, focusing on the configuration management system.**

***Index Terms*—configuration management, puppet, system administration, xcat, stateless**

I. Introduction

Configuration management is one of the primary ways of administering an HPC system as a means of ensuring a sane, stable, and consistent platform for users. An ideal configuration management system is one that

- is easy to use and is intuitive
- provides for an easy way of testing changes before deploying to production
- is quick to deploy changes
- has minimal disruption to running jobs
- is easy to debug and fix when something breaks
- is scalable to hundreds or thousands of nodes

Purdue University has gone through many different styles of configuration management [1]. The first was a proprietary administration platform for an IBM SP2 system, followed soon after by an in-house configuration tool called Master Source. An in-house, stateless image provisioning system named STACI soon followed. Due to scaling problems at the time, STACI was abandoned for a stateful CFEngine [2] system. Approximately four years later, CFEngine 2 was reaching its end-of-life and a decision was made to switch to stateful and masterful Puppet [3]. This new Puppet system involved heavy use of templates and variables to manage the entire infrastructure and all HPC systems. Five years later, the masterful Puppet system had become unsustainable, burdened by difficulties with scaling and seeing what should have been innocuous changes causing rippling outages across the infrastructure, and again, the decision was made to switch configuration management systems. A new system using xCAT [4] and GoCD [5] for continuous integration and continuous delivery (CICD) was developed. The CICD system was very brittle and changes took a very long time to deploy in production, and a change was needed again.

Within the past year, we have created a new configuration management system using xCAT and masterless Puppet that is used for all of the HPC systems at Purdue University that takes into account the many lessons learned over the years. We believe this new system meets or exceeds all of the items listed above, and we want to share our experience with the community.

II. New HPC System Architecture at Purdue

Each cluster is designed as an island with as few external dependencies as possible. Each cluster has its own LDAP replicas, static `/etc/hosts` files, DNS server, NTP server, xCAT master, etc.

A. *xCAT*

xCAT is an extensible whole-cluster management and provisioning toolkit. Our xCAT deployments PXE boot a minimal initial RAM disk that formats the local system disks on boot, downloads the stateless front-end and compute node system images, and then switches root to the downloaded system image that contains a copy of our Puppet configuration that runs after the systems boot. xCAT can do much more this to manage a cluster, but at Purdue, we use a very minimalist deployment of xCAT.

We have made a clear delineation of which things should be managed in Puppet versus which things should be managed in xCAT. xCAT is used to install the kernel, OFED, firmware, Lustre/GPFS kernel modules, and the basic packages needed to run Puppet. All other packages and system services are configured in Puppet.

When creating system images, our build scripts run Puppet in a chroot on the root file system that becomes the system image, so the system images come with most of the package and system service configuration already complete when the nodes boot.

1) Drift Reduction: The stateless model was chosen to reduce system drift and for ease of fixing broken nodes. If a node is broken, reboot it; if the node is still broken, there is defective hardware.

Additionally, we run LBNL's Node Health Check [6] to ensure that firmware, BIOS, and OFED versions are a specific version.

B. *Masterless Puppet*

Puppet traditionally has servers called "Puppet masters" that determine what the state of the client machines should be. This becomes complex and expensive to scale when the number of client machines gets into the hundreds or thousands. To bypass this, we cut out the Puppet masters and have each client machine review and apply its own configuration state, effectively distributing the entire load across the entire cluster.

1) Git Branches: The configuration for each cluster is kept in its own Git repository from which machines pull updates and then ensure the desired state via Puppet. Each repository has three branch types:

- development
- master
- deployed

The development branches are for new feature development or testing bug fixes. Once the change has been considered stable, the development branch is merged into the master branch, which is the authoritative branch of the configuration. Prior to a scheduled maintenance, a deployed branch is forked from the head of the master branch, and this deployed branch will be used in production on the cluster after pre-maintenance testing and verification.

2) Maintenances: Stateless system images are built days or weeks in advance, depending on the amount of pending changes, and a small portion of the cluster will be booted into the new images for real-world testing ahead of the maintenance. On the day of maintenance, the cluster is booted into the new images and further testing begins. In the event of an unexpected, severe problem, maintenances with stateless system images can be reverted by simply booting the cluster into the prior system image. After the maintenance, the cluster will be running on a deployed branch that is expected to stay stable—only the most important of changes should be merged into a deployed branch once in production.

3) System Roles: The Puppet configuration is split into multiple "roles". There are many ways these roles could be implemented, and the following is simply the high-level implementation that we have used so far.

The base role, named "common", is applied to every system in the cluster, and this role is responsible for managing user accounts, SSH settings, system monitoring and logging, and other generic settings that are uniform across the cluster. Other roles are then applied in a second Puppet run after "common", depending on a system's function. Table I shows the various Puppet roles we have created, along with a description of each.

TABLE I
PUPPET ROLES

Role	**Description**
common	base role applied to all systems
adm	job scheduler & InfiniBand subnet manager
aux	auxiliary services: DNS & LDAP replicas, NTP server, VM hypervisor
compute	compute nodes
frontend	front-end nodes
samba	Samba server exporting scratch file system (VM)
system	xCAT management node/master
thinlinc_master	master server for the ThinLinc desktop (VM)

4) Puppet Configuration Style: Puppet has the ability—via tools named Hiera [7], a key-value store, and Facter [8], a system profiler—to use templates in a programmatic, in-line style, similar to PHP; however, we decided to avoid using this approach unless absolutely necessary (e.g. if a configuration file or manifest needs the local machine's hostname from Facter or a decrypted value from Hiera). Puppet roles present a minimal root file system-like directory of static configuration files, and our Puppet configuration simply takes a static file from the directory and places it in the same location on the running system. This means that configuration files are where system administrators expect them to be and what they expect them to contain. There is no guesswork about what a configuration variable will be—the file is static.

Hiera is a hierarchical key-value store for Puppet that is frequently used for templates. Hiera has an extension called eyaml [9] that supports encrypted values. We use Hiera *only* for encrypting secrets.

Additionally, there is an explicit separation of package installation and configuration file installation. The Puppet roles are designed so that packages are always installed successfully first before any configuration file gets installed.

5) Stateful Infrastructure: All cluster roles from Table I are stateful except for "compute" and "frontend" which are stateless.

III. Evaluation and Thoughts

A. Maintenances

Maintenances are trivial. System images can be built easily and tested in advance, so most bugs can be caught and fixed before the day of maintenance. If major problems are encountered during the maintenance, it is trivial to just activate the previous system image and reboot the cluster. With stateful cluster nodes, reverting is much more difficult because it would typically involve a `yum downgrade` for every system package and being bottlenecked by the Puppet master and yum repository servers, in addition to a lack of confidence in our Puppet implementation successfully reverting all of the changes and updates.

B. Speed

Masterless Puppet, without complex templates and variables for every configuration knob, is quite fast compared to the opposite system we had previously. Our masterful Puppet environment would usually take several minutes per Puppet run whereas our masterless Puppet system usually completes in around 10 seconds.

The masterful Puppet system was so slow because the Puppet master servers had to calculate regularly the desired state of each machine in our environment which, at its peak, numbered around 4,000 servers; however, with masterless Puppet, this state calculation is distributed among every machine in our environment.

C. Disaster Recovery

Recovering broken nodes is trivial with stateless images. If a node is acting strange, simply reboot it. If the node is still having problems, then investigate the cause and look first for failed hardware. For example, one of our old, stateful clusters needed its kernel downgraded. A mistake during the kernel downgrade broke the GRUB bootloader configuration, causing all of the nodes to get stuck during boot, requiring manual intervention on most of the 660 compute nodes. If the cluster was stateless, simply reverting to the previous system image would have fixed the problem.

D. No Drift

Reinstalls of compute nodes have been a source of pain with stateful hosts. We would see an assortment of firmware, BIOS, and package versions depending on when a compute node was reinstalled or replaced. Now with stateless system images, every compute node is always identical to every other compute node so drift of package versions, etc. does not occur.

E. Easy On-boarding of New Administrators

The training time required to bring a new system administrator up to speed is low. Configuration files are where most people expect them to be. There is no complex new language to learn; all Puppet code used is almost entirely at the "Puppet 101" level of "take this static file and put it here".

Advanced templating made it very difficult to find where you needed to make a change. Simply changing a single value in a single configuration file would regularly take many times as long as just changing a value in a static file—first, one would have to see whether the file was properly put into a template, then one would have to check if the variable to be changed is set in the module configuration with the template or whether it is set in Hiera, then one would have to make sure the variable is being set in the correct level of the Hiera hierarchy. The significantly complex process made everything tedious compared to finding a file in a known place in the file system and changing the value directly.

F. Development Nodes

Because development occurs on Git branches, it is trivial to test changes:

1) offline a compute node
2) switch its running configuration branch
3) run Puppet on the new development branch
4) iterate over changes
5) merge the completed feature to master
6) reboot the compute node
7) re-enable the compute node in the job scheduler

After rebooting, the compute node boots from the production system image, and all traces of the development environment are gone.

G. Lightweight Impact on Infrastructure

Because of the siloed architecture and the use of system images, there are no centralized configuration masters, DNS servers, or Yum repository servers that must bear the constant load of thousands of compute nodes. The per-cluster auxiliary servers handle the brunt of this responsibility, and the centralized infrastructure must only handle the burden when an auxiliary server goes down. We have also found that PXE booting 550+ compute nodes simultaneously is trivial and non-impacting when the provisioning server has a 25+ Gbps connection.

H. Smaller Failure Domains

Our previous monolithic configuration management system regularly had problems where seemingly innocuous changes to one cluster would cause failures on other clusters because of unforeseen consequences, and the root cause of the failure may not be evident or obvious. For example, a MariaDB package was put into our Yum repositories, and over the course of several hours, MySQL servers were breaking *everywhere* in our infrastructure. The root cause was that the MariaDB package obsoleted MySQL. Isolating configuration management systems from each other limits the scope of such a mistake.

I. Inter-cluster Drift

Because each cluster's configuration is siloed off independently, there will be some drift between clusters in the configuration. We have accepted the risk of differences between cluster configurations, and we are not worried about it. Important changes will get the proper care needed to make sure they propagate everywhere, but minor or cosmetic changes will simply go in the most up-to-date configuration repository and may not be pushed to all repositories.

IV. Conclusion

While there are many ways to architect an HPC cluster, we have found this model to be excellent to work with because of these main points:

- xCAT provisioning stateless nodes
- masterless Puppet reducing load on infrastructure
- minimal use of templates

We have had such positive experiences with this model that in May and June of this year, we upgraded five of our HPC clusters from the old, masterful Puppet system running RHEL 6 to CentOS 7 using the new xCAT/masterless Puppet model.

References

[1] P. M. Smith, J. St. John, and S. L. Harrell, "There and back again: A case study of configuration management of HPC," in *Proceedings of the HPC Systems Professionals Workshop*, ser. HPCSYSPROS'17. ACM, 2017, pp. 5:1–5:7. [Online]. Available: https://doi.acm.org/10.1145/3155105.3155110

[2] "CFEngine," https://cfengine.com.

[3] "Puppet documentation," https://puppet.com/docs/puppet/5.5/puppet_index.html.

[4] IBM, "xCAT: Extreme cluster/cloud administration toolkit," https://xcat.org.

[5] "GoCD," https://www.gocd.org.

[6] M. Jennings, "LBNL node health check," https://github.com/mej/nhc.

[7] "About Hiera," https://puppet.com/docs/puppet/5.5/hiera_intro.html.

[8] "Facter documentation," https://puppet.com/docs/facter/3.11/index.html.
[9] T. Poulton, G. Meakin, S. Hildrew, and R. Fielding, "Hiera eyaml," https://github.com/voxpupuli/hiera-eyaml.

APPENDIX

A. Abstract

This artifact is an extremely minimal example of the Puppet configuration highlighting some of the primary design goals. This artifact illustrates:

- how the run_puppet script works with the multiple roles
- how Hiera is used as a secrets store for SSH host private keys
- the basic Hiera, Puppet, and eyaml configuration files
- the separation of package installation versus configuration files
- the basic method of installing configuration files without using templates

B. Description

1) Check-list (artifact meta information): Fill in whatever is applicable with some informal keywords and remove the rest

- **Program: Puppet, Hiera, hiera-eyaml, eyaml**
- **Run-time environment: Linux, yum**
- **Publicly available?: yes**

2) How software can be obtained: https://github.com/HPCSYSPROS/Workshop18/tree/master/xCAT_and_Masterless_Puppet_Aiming_For_Ideal_Configuration_Management

3) Hardware dependencies: None

4) Software dependencies: This system was developed using Puppet 3.7.5, hiera-eyaml 2.1.0, and CentOS 7; however, a Linux distribution with yum as the package manager should still work with the example artifact.

5) Datasets: None

C. Installation

- install Puppet
- remove the directory `/etc/puppet`
- extract the artifact into the directory `/etc/puppet` so that the directories `hieradata`, `modules`, `manifests`, etc. are directly under `/etc/puppet`
- install the hiera-eyaml Ruby gem
- create eyaml keys and put them in the default location so Puppet can see them
- copy `modules/common/files/etc/eyaml/config.yaml` to `/etc/eyaml/config.yaml`
- look at the SSH host keys under `modules/common/files/etc/ssh`, in `hieradata/common.yaml`, and in `modules/common/manifests/config.pp`, and replace these with local SSH host keys that have been encrypted with the eyaml keys from the previous few steps
- execute the command `puppet apply --hiera_config=/etc/puppet/hiera.yaml /etc/puppet/manifests/common.pp`
- execute the command `puppet apply --hiera_config=/etc/puppet/hiera.yaml /etc/puppet/manifests/compute.pp` (optionally replace `compute.pp` with `frontend.pp`
- execute the command `/usr/site/rcac/sbin/run_puppet`

D. Experiment workflow

Complete the installation section, above, and add new configuration files and packages to the Puppet manifests based on the provided examples.