LA-UR-22-23444

Approved for public release; distribution is unlimited.

Title:	Deploying Cray EX Systems with CSM at LANL
Author(s):	Stradling, Alden Reid Johnson, Steven Lee Van Heule, Graham Knox
Intended for:	Cray User Group, 2022-05-02/2022-05-06 (Monterey, California, United States)
Issued:	2022-04-15 (Draft)

Deploying Cray EX Systems with CSM at LANL

Alden Stradling
Los Alamos National Laboratory
Los Alamos, NM
stradling@lanl.gov

Steven L. Johnson
Los Alamos National Laboratory
Los Alamos, NM
slj@lanl.gov

Graham Van Heule
Los Alamos National Laboratory
Los Alamos, NM
grahamvh@lanl.gov

***Abstract*—Los Alamos National Laboratory has deployed (over the last year and a half) a pair of Cray Shasta machines – a development testbed named Guaje and and production machine named Chicoma, which will soon comprise the bulk of LANL's open science research computing portfolio.**

In the process, we've encountered a number of problems and challenges in several realms – authentication and authorization, cluster health management, image management, and configuration management. Both independently and in collaboration with Cray/HPE, we've found solutions and brought the system into stable production.

The presentation will discuss the solutions and how they came about, and issues we are working to resolve in the near future.

***Index Terms*—Shasta, Kubernetes, container, HPC, LDAP, Keycloak, Slurm, authN, authZ, Chicoma, Guaje, Los Alamos, CSM, management, cluster**

I. Introduction

It has been clear for at least a decade that High-Performance Computing systems management software has been diverging from the broader stream of high-throughput management methods, both commercial and governmental. It has also been stated that such divergence represents a stagnation,

"relying on incremental changes to tried-and-true designs to move between generations of systems." [1]

And those tried-and-true designs have been, in large part, successful. Machines have been built, have run, and have been refined. At the same time, systems in the wider world have been built much larger, at much higher uptime requirements, for different but truly challenging workloads, and with a desire to be run much more efficiently.

Granted, those systems and workloads are not the tightly-coupled HPC workloads upon which we focus. Google's approach can easily handle a lost node or ten. An HPC management plane is much smaller... but it can and should benefit from similar resiliency improvements. Administrators from several national labs have argued [1] that adopting these modern technologies can improve manageability, resiliency, scalability, and security. In this paper, we'll address the first two, which go hand in hand. Scalability can't be tested well with the clusters we've deployed so far and security is worth its own paper.

Consistent with these suggestions, Cray (later HPE) has produced the Shasta Cray System Management system (CSM). Starting in 2019 and continuing through the present, LANL staff have been evaluating the combined software and hardware of CSM, River (air-cooled management hardware), and Mountain (water-cooled compute hardware) systems through several iterations. In this paper, we will discuss how CSM has addressed and improved manageability, resiliency, serviceability, scalability, and security (if at all), what we've done to adapt it and adapt to it, and what we envision for the near future. The Chicoma system comprises:

- 2 River racks (3 master nodes, 4 worker nodes, 3 storage nodes, 2 UANs, 8 LNET gateways)
- 4 Mountain racks (560 AMD Rome EPYC 71H2/512GB nodes, 118 AMD EPYC 7713 + 4 AMD A100 Tensor GPU nodes)

The cluster has been in active use for COVID-19 research since April 2021.

II. Manageability

HPC system management breaks down into two core categories: compute node lifecycle and cluster services. There is some overlap, but we'll break it down as follows:

Compute node lifecycle:

- Boot image creation, node boot (TFTP, DHCP, NFS), postboot configuration (Ansible) end user authN/authZ providers (password files and LDAP interactions), node health management, monitoring, and shutdown
- Orchestration of the lifecycle becomes more important as node counts increase

Cluster services:

- Initial implementation (installation, networking, hardware discovery, tuning)
- Basic services (NTP, logging, authN/authZ for cluster administration, mail connections, shared filesystems or the equivalent)
- Fabric implementation and management (IB, Slingshot)
- Job schedulers (Slurm, Moab, SGE, Flux), with potential fabric integration
- Service node provisioning (Ansible, version control, image creation, backups)
- Upgrades (management node replacement, software updates, security patches, compute resource addition)
- Monitoring (performance, services, network)
- Management plane resiliency (cabinet loss, service node loss, switch loss)

There are hundreds of implementations in the field that address these concerns to one degree or another. A "more manageable" system minimizes or zeros downtime for most routine interventions, reduces operational complexity for the admins, eliminates potential data loss scenarios (configuration or user job), tests new image changes rapidly, and allows those changes to scale rapidly and reversibly. It is resilient against hardware loss (whether intentional or otherwise) and orchestrates the necessary services for maximum uptime with minimum intervention.

The Shasta Cray System Management (CSM) stack is architected to these guidelines. It is based around Kubernetes (https://kubernetes.io/docs/reference/), a "portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation." Basic services and other components of the management plane are moved into containers. The necessary permissions, networking, storage, logging, and monitoring are then provided by Kubernetes (`k8s`). The containers are run in "pods", which mediate all of these configurations via declarative configuration. When pods need to be moved or restarted, or hardware fails beneath them, they are restarted on a new resource with minimal interruption. If new hardware is added, `k8s` adds workloads automatically. Services that lose connectivity or fail can be detected by pod liveness tests and restarted even if they hang/degrade.

The initial LANL experience with Shasta (at version 0.8.0) did not reach those goals. Early implementations were difficult to install, unstable, sensitive to change, could not reboot, and accumulated errors and full disk volumes over time. Despite this difficult start, the `k8s` plane and its installability and maintainability has improved over the last two years to the point that our production system, Chicoma, has accrued a reputation for stable operation since its soft launch in April 2021.

This reputation has come with some constraints, however. Major and minor upgrades to CSM have been unpredictable. Some were uneventful took a matter of hours (point releases in the CSM 1.4 family, for example), but others have taken days (Shasta 1.3/CSM 0.8) or more than a month (Shasta 1.4/CSM 0.9). This has led us to adopt a conservative attitude towards updates on the present system – at the moment, we are running CSM 0.9.6 and have allowed it to sit stably (except for a couple of security fixes) and provide user cycles. We have also assiduously avoided reboots and power cycles of the whole management plane at once. We are concerned (based on experiments with our test cluster) that full reboots in CSM 0.9 can come back badly and require significant repair.

Does Shasta provide better admin manageability for the service nodes? Breaking down the positives:

- Services like Slurm, NTP, DNS, TFTP are never down. Similarly, the databases that drive a number of the services we need are very stable (multiple instances, so even a single DB failure is unnoticeable).
- Lifecycle services (boot orchestration, image creation, etc) are more likely to fail, but are not in the critical path for system stability.
- Backing storage (an internal Ceph cluster) is very, very stable and recoverable – and `k8s` handles degraded states gracefully. In one instance, two of the three Ceph nodes were frozen, and the management plane was frozen. Jobs continued to run on the compute plane… and once the two frozen nodes were rebooted, Ceph came back without complaint and `k8s` picked right back up and continued without further intervention.
- Fabric upgrades are simple and reversible.
- Monitoring and system test scripts have revealed issues before they became critical
- We have successfully explored the potential for management node addition on the fly in the Shasta 1.1/1.2 timeframe with `libvirt`-mediated "new nodes" successfully added to the Kubernetes cluster (not in production, naturally!)

And the negatives:

- Upgrades come in as multi-gigabyte archives, even for small changes. They are unwieldy and take a long time to download and unpack.
- Fabric upgrades have presented problems on our systems routinely, and we've exercised their reversibility a few times.
- Initial installation is very sensitive to configuration files. Errors in those files can remain undiscovered problems for weeks or months.
- Service node provisioning, configuration, and updates are part of an extremely complex process with a number of opportunities for error introduction.
- Failure of the compute plane (a Cooling Distribution Unit brownout leading to EPO) can have unforeseen effects on the management plane

Cluster health monitoring is included in the Shasta product (and is being integrated into our existing Splunk infrastructure), but we have also integrated some of the solutions we use on legacy clusters as well. A more detailed discussion is included below.

III. Cluster Health Management

`lbnl-nhc` is the tool we've used internally to monitor the health of our clusters. Its framework provides an easily extendable and configurable setup. The NHC configuration file allows one to set up as many checks as one could desire on a given node. There are a large number of built-in checks for common node issues such as user processes remaining on a node, and checks to ensure mounts are correct. It also is very easy to write checks in bash to cover almost anything you would want to check. Its built-in support for marking nodes down in Slurm also allows for easy tracking of node state across the system. With NHC we've been able to build out checks for our compute nodes, non-compute nodes and clusters as a whole.

IV. Compute Health Monitoring

For node health we make use of `lbnl-nhc` on compute nodes. The compute nodes run two separate configurations of `lbnl-nhc`. One does heavier checks on idle nodes. The other is a lighter group of checks that runs on a busy compute node. This combination allows us to validate that a node is clean and ready for its next job, without interfering unduly with running jobs. We also implement checks on our Node Management Network nodes using `lbnl-nhc`.

V. Non Compute Node (NCN) Health Monitoring

Though often overlooked, the validation of non-compute nodes (NCNs) is also important to monitor the health of a cluster. NCNs include UANs (User Access Nodes), Lnet routers, Kubernetes worker and management nodes, and Ceph storage nodes. The biggest issue one runs into when trying to run NHC to validate the non compute nodes on a cluster is that there is no built-in centralized mechanism to track state on those nodes and the reasons for their problems. One can put a `slurmd` on them, with the risk of a misconfiguration causing user jobs to run on that infrastructure. Rather than take that risk, we elected to use a shared NFS mount backed by Ceph to write the state of each node. Each node thus mounts the shared node state directory and writes an empty file if their state is OK or writes the problems a node is facing as a string in that file. A `stat` can then quickly detect nodes with issues.

VI. Cluster Health Monitoring

Moving from the idea of validating nodes, we also need to look at the state of the cluster as a whole. Since a "Node Health Check" doesn't apply to a cluster as a whole, we decided to use a separate NHC "context" named "CHC" (Cluster Health Check) for this purpose. How does one validate a cluster? CHC validates a sufficient healthy ncn node count (as reported by NHC running on those nodes, checking for specific kubernetes problems, and doing critical service validations such as running sinfo to ensure slurm is responding). The big difference when dealing with nodes and the cluster as a whole is that we need more than a binary "OK or not OK". CHC was therefore split out into three different levels of problems: warnings, daytime notification, and 24x7 notification. Warnings cover minor issues that might point to something larger. Daytime notifications and critical errors require action, but differ in urgency.

VII. Compute Lifecycle Manageability

Sysadmins are judged by their system's uptime and their responsiveness. Often there's a lot of tension between these two virtues. A "more manageable" system would allow the sysadmin to make changes easily in a configuration management tool, have those changes rapidly create a new compute node image, deploy that image for testing on a representative subset of nodes, roll the changes out more broadly as the tests go well, and roll back rapidly if issues show up at scale. The basic Shasta architecture is well-designed to allow such mechanisms, but its promise has not been realized. The effort needed to build and deploy images is complex, error-prone, tedious, and poorly instrumented at the moment. New tooling (extensions to SAT in Shasta 1.6 with the sat bootprep command) is expected, but in the meantime LANL has an interim tool written by Graham Van Heule, that will be discussed in the next section.

VIII. Image Management

On a high level, Cray provides three tools for image management: Image Management Service (IMS), Configuration Framework Service (CFS), and Boot Orchestration Service (BOS). IMS handles building an image from a recipe that is largely a starting point. CFS configures said image with ansible, and BOS is what determines which image a node boots from. An image build looks like this:

1) Tell IMS to start a job to build an image from a recipe (we'll call this a bare image)
2) Wait for IMS to finish building the image
3) Launch a CFS job to configure the bare image from the prior step
4) Wait for CFS to finish configuring the image
5) Set the BOS config to use the newly configured image to boot said nodes.

The built-in cray tools tend to be impractical for three reasons: one needs to be an expert at `jq` to get the information out of it, it requires considerably more commands than necessary to perform an action, and provides no defaults to know what 'normal' expectations are. Due to this complexity it makes these tools difficult to learn, error prone, and just plain painful to use. These problems spawned the need for a wrapper to handle all of this. The first step of this was to simplify the output of commands. So for example getting the list of images displays the creation date/time, the image ID and the image name instead of all information about all images. If that information isn't enough there's the option to show all information on a given image, thus making things more manageable and understandable. Next was to reduce commands down to simpler actions. One example of this was reducing how to get logs on a given action. On the Shasta system getting logs on an action generally involves:

1) Digging into the details about the action
2) From the action get the the Kubernetes job with which it's associated
3) From the job find the Kubernetes pod with which it's associated
4) Looking at the different logs of the pod until you find what you want.

This is all reduced down to a log command that we can easily run against the given action. That image build process shown earlier is now broken down into a single command or three separate ones depending on your needs. The last step was to create defaults. In order to know where our current system is in terms of configuration, it's best everyone is on the same page in terms of each of the pieces. So we have a default recipe

to build each type of image, a default BOS config to boot each node type, and a default image name for each type of image. With this in place, we can do things like spawn off an image build that will handle all of the layers and set that image to be booted for the nodes to which it applies. We can also reboot nodes without thinking about which BOS template to use. With this tooling, the compute node manageability aspect of Shasta has become far less painful – and more flexible than any other tooling we employ at LANL. With improvements to the process and the advent of officially-supported tooling with similar characteristics, we expect that Shasta systems will indeed allow HPC sysadmins to resolve some of the tension between system stability and agility.

IX. Config Management

Correct long-term image management is only as good as the configuration management that underpins it. When setting up our ansible for these clusters our main goals were:

1) Have as much of the configuration in common as is reasonable (to reduce configuration drift)
2) Attempt to keep `lanl` configurations separate from Cray's configuration to allow for easy identification of what Cray has changed in updates and new releases and what we have changed.
3) Configuration should be stored off-cluster to simplify recovery from catastrophic failures.

With that in mind, we set up our environment to have an external Gitlab server, separate LANL repositories, and are working with Cray to get submodules to work with CFS. Over the last couple of years, we have set up an external Gitlab server to manage the Git repositories for our clusters and migrated most cluster configurations to it. This was done to ensure a centralized place to manage our configuration. As our Gitlab instance couldn't serve thousands of clients, we also created a caching deployment on each of the clusters in Kubernetes to serve the Git areas to the rest of the cluster. As an independent entity from our clusters, the Gitlab server also allows access to configurations even if the cluster is down. The biggest drawback to this setup, however, is that we need to push Cray changes from the internal Shasta Gitea instance to our Gitlab instance with every upgrade. In order to keep our changes separate from Cray's updates, we separated out all of our local changes into two separate Git repositories: diskless and diskful. The diskless repository is for image building and configuring nodes without persistent storage. The diskful repo serves nodes that boot off of local persistent storage. As it turns out, diskless and diskful node configs had very little in common. Diskful nodes manage and provide services to the cluster, and diskless nodes run user code. We're currently working with Cray to get submodule support added to CFS' git pulls. A git submodule is a function in git that allows one repo to include another repository as an internal directory. This will allow us to have individual git repositories for each cluster containing variables and inventory, and a central roles repository that's pulled in as a git submodule. An alternative to submodules is scripting in the cluster-specific inventory changes each config management is called. The Shasta-provided `AdditionalInventory` option does something along these lines, and has the virtue of existing CFS integration and direct support. We are looking carefully at it, balancing commonality with our existing systems and ease of implementation. For the moment, we just maintain a repository set for each of our two present systems. That number will go to seven by early next year, however, so a timely solution is important.

X. Developing Better Image Build Pipelines

Even this level of automation can be improved. We have work in progress to use Gitlab runners as build pipelines for automated image creation. Either manual intervention or checking in changes to affected repos can trigger launch of runner that uses one of a variety of tools (Buildah, direct Ansible configuration in a Docker container, etc) to recreate whatever layers of the image are affected by the change, upload the results to S3, and register the image with the IMS registry, ready to be used in BOS runs whenever desired.

XI. Slurm

Both Slurm and PBSPro are the supplied options for workload management. Slurm is in use at LANL in the existing clusters and we did not deviate from that selection. As is done at most sites, we brought up our own custom Slurm to meet our particular needs. From the factory Slurm in the Cray EX Software environment is implemented as three pods: `slurmctld`, `slurmdbd` and `mariadb`, in their own Kubernetes namespace. Slurm is not easily scalable in a horizontal manner, but Kubernetes can allocate and limit resources used by the single instances of these components.

When we transitioned from the vendor supplied Slurm to our locally built RPMs, we had to build our own containers. Cray had used a base SLES15 image for their `slurmctld` and `slurmdbd` images, so we would follow their lead. We have two options for this: duplicate the Cray Slurm containers and upgrade the rpms therein, or download a fresh SLES15 image and start there. We've done both and settled on the latter.

The Slurm RPMs are locally built, signed, and dropped into a local repository on the cluster. Building the `slurmctld` and `slurmdbd` images is a matter of constructing a Dockerfile to define the packages needed to run the service along with a few extra packages for convenience. We use the existing mariadb instance from Cray. After the images are built, we edit the Kubernetes Deployment specifications for the `slurmctld` and `slurmdbd` to point to the new images. When the pods are restarted, we're running our locally-built Slurm.

As part of a long-term project, the `slurmdbd` has been pulled out of the cluster and is supported as a standalone entity within the LANL HPC Division. This offered a more robust database backend to the central `slurmdbd` and removed the `slurmdbd` load from our clusters. The change was accomplished by a single-line change in the `slurm.conf`

file, along with some upstream firewall modifications. The `slurmctld`, from Cray had a `macvlan` interface defined for the internal cluster networks, but in this new configuration an additional `macvlan` interface was needed to facilitate direct communication to the remote `slurmdbd`. These interfaces effectively give the `slurmctld` a static, well-known IP address.

Much of the Slurm configuration was done by post-boot Ansible activity. When we changed to running our locally built Slurm, we also changed to "configless" Slurm. This involved an edit to the local DNS in the cluster of the form `_slurmctld._tcp 3600 IN SRV 0 0 6817 slurmctld-service`, which allows the compute nodes to find and download their `slurm.conf` from the host, `slurmctld-service`. This simplifies the post-boot configuration considerably and ensures that we have a single source of truth for the `slurm.conf`. A git repository is used for maintaining the `slurm.conf` and related files, and merges to the integration branch automatically update these files on the cluster.

XII. AuthN/AuthZ

CSM ships with Keycloak for authentication in the Kubernetes management plane of the cluster. HPE extended this to extract users from Keycloak into `passwd`, `group`, and `shadow` files with some additional effort. Initial user accounts are created from the Keycloak LDAP integration in a one-time process, running in the `keycloak-users-localize-X` Kubernetes Job, which deposits `passwd` and `group` files into the WLM bucket in S3. Since we wanted regular updates, we created a Kubernetes CronJob to do this process on a regular basis, and tooling to have each of the compute nodes pull those files from S3. This was functional and very scalable, but we don't like the idea of having end user accounts sourced from inside the Kubernetes plane. We've evaluated a couple of end-user AuthN/Z alternatives for the cluster.

For clusters to be good neighbors, one thing they should not do is spam their upstream authentication provider. During our early testing phase we had a small number of users, so our user data was relatively static. To centralize the directory information we employed a LDAP server inside the cluster for AuthZ. While we were quickly gaining knowledge of Kubernetes and how Cray assembled the various parts of it to support a cluster, the decision was made to create a new `lanl` namespace within the existing Kubernetes cluster for housing projects that we may develop.

The first of these was to bring up a 3-instance OpenLDAP service using containers. A very small base operating system image was used and its package management brought in the necessary applications to bring up an LDAP service. Some convenience applications such as `bash` and `strace` were also added to the image. The image was constructed with the standard docker/podman processes and pushed into a registry within the cluster. The OpenLDAP application is deployed as a Kubernetes StatefulSet with instance 0 as the primary server. An InitContainer is defined in the pod for the initial setup of the OpenLDAP directory. The other instances function as secondaries using standard OpenLDAP replication techniques. Each instance has its own Kubernetes PersistentVolume for storage of the directory date. X.509 certificates are in place for TLS encryption between the instances and the clients.

We were able to fully utilize Kubernetes features to roll-out new images and migrate pods to other nodes during Kurbernetes node reboots. The clustered OpenLDAP service performed quite well, especially when combined with client-side caching. It has also been easy to keep up to day. OpenLDAP happened to be the directory choice at the time when we deployed it. We could have chosen another open source directory application and followed the same process to containerize it and run it under Kubernetes. Containerizing the cluster's directory service was an extremely valuable learning experience.

Another means of providing directory service is using `nssdb` – that is, db files for `passwd/shadow/group`. For SLES, these are stored in `/var/lib/misc/` and will be consulted with the `/etc/nsswitch.conf` file containing entries of the form `passwd: files db`. This is attractive because the flat files in `/etc` are never modified, reducing the risk of locking ourselves out of systems via admin error. We've found that this performs and scales quite well (for this system size, at least), and that management of the db files from upstream sources is relatively straightforward.

For administrative access patterns in Kubernetes, we continue to use the basic single-user administrative structure provided out of the box. The next few months, however, will bring us a new LDAP source with good `memberOf` characteristics. Translating admin group membership to Shasta/Kuberentes privilege constellations will allow admins to log in as themselves rather than as root on the master nodes and operate within more role-specific constraints. This should cut down on the number of admins that need to operate in superuser mode.

XIII. Local Services

In addition to the directory service, we found it necessary to add two additional local services to help run the cluster: a basic web server (Nginx), a proxy server (Squid), and a login server (`ssh`). The configurations for each are stored in a git repository outside of the cluster.

CSM is distributed with Sonatype Nexus to provide a package repository as well as an image registry. While Nexus is an elegant solution and offers an API for advanced management, we needed something simple for some of our repository needs. Like a traditional web server deployment, Nginx serves up files from a basic directory structure, and RPM repos are easily spun up with the `createrepo` command. We use NFS storage for this. The directory is mounted read-write on our management node and read-only inside the Nginx container. Instead of a StatefulSet, a Kubernetes Deployment is used to define our instance of Nginx. A Kubernetes ConfigMap brings in the Nginx configuration. Here, too, we have been able to use

the capabilities of Kubernetes to roll out new images, scale the deployment up and down, and migrate pods between servers.

Along with the Nginx web server, we found that it was useful to have a Squid server in front of Nexus to reduce the load on the singular Nexus instance during the infrequent occasions when RPM packages were being sent to running nodes. Very similar to the Nginx and OpenLDAP services, we used a base image and had docker/podman bring in the packages essential to run Squid. Its configuration is brought in via a Kubernetes ConfigMap.

In the early days of bringing up CSM and the rest of the Shasta system, we lacked the hardware for a login node. We had two compute nodes, the Kubernetes and Ceph nodes, and nothing more. To provide login services, we used a SLES15 base container image and built up a working login environment on top of it. This was obviously considerably larger than the simple service containers above. Through the use of macvlan networking, we were able to expose the login container on the external network for direct SSH logins. The login containers were capable of supporting multiple users and carried out the standard compile/edit/debug process as though they were bare-metal systems. We used Kubernetes limits to constrain the CPU and memory usage of the containers. This worked quite well, but the concept has been retired in favor of bare-metal login servers.

These additional services filled gaps for us in CSM. They have performed quite well, and provided yet another learning experience for the staff.

XIV. Internal File Service

Our two clusters each have three Ceph storage servers. Cray intended for these to be used as the primary storage for the Kubernetes cluster.

However, we had a need for user and application storage within the cluster during early development. Eventually, our production systems would utilize centralized storage, but for the pre-production period we needed something local and simple. For this we created a new pool in Ceph along with a replicated block device. The block device would be used as the foundation for a `zpool` in ZFS, which would then be exported via NFS to the rest of the cluster. Local and remote ZFS snapshots provided adequate backup for the development period of the clusters. For convenience, we'll likely continue to use this for local scratch filesystems which are not sensitive to lower I/O rates.

XV. Conclusions

Several years' experience in building and running Shasta systems at LANL has taken our team through a wide range of challenges and solutions. From an inauspicious start to the present period of stable, almost boring operation (except for power events and upgrades), we think there's reason to believe that the Shasta platform will come to realize the goals of increased manageability and resiliency, and provides a starting point for further developments that break the constraints of existing operation models. It has also proven adaptable to our sometimes unique site services and requirements. It has certainly motivated (and been a testbed for) improvements in LDAP, Slurm, Gitlab, monitoring, and other infrastructure. We're certainly glad to have these improvements and adaptions in place with five more Shasta-based systems arriving at LANL within the next calendar year.

References

[1] B. S. Allen, M. A. Ezell, D. Jacobsen, C. Lueninghoener, P. Peltz, E. Roman, and J. L. Wofford, "Modernizing the HPC System Software Stack," 2020. [Online]. Available: https://zenodo.org/record/4324415