

Stateless Provisioning: Modern Practice in HPC

John Blaas
University of Colorado Boulder
Boulder, CO USA
john.blaas@colorado.edu

John Roberts
Argonne National Laboratory
Lemont, IL USA
jroberts@anl.gov

Abstract—We outline a model for creating a continuous integration and continuous delivery work flow targeted at provisioning CPIO based initramfs images that are used to run computational work nodes in a bare metal cluster running RHEL or CentOS.

Index Terms—High performance Computing, Supercomputers, Software tools, Configuration management

I. INTRODUCTION

As the number of nodes increases in a cluster it becomes increasingly difficult to manage configuration drift using traditional methods of configuration management which assume an eventual convergence on a desired state. Traditional methods typically involve running a configuration management agent on the node and having the node pull down configuration changes on a periodic basis with the idea that the nodes in a cluster would reach convergence after some time. In larger centers the use of configuration management tools that relied on a back end database to store cluster configuration started to emerge. These systems that relied on a database [1], [2] also typically used NFS to share a OS directory tree that was used to boot nodes in clusters with no local disk drives on the node.

This body of work focuses on a cluster software stack which utilizes RHEL or CentOS as the OS, SLURM as the resource scheduler, and NHC from the Warewulf project [2] [3] as the node health check script. Other software could be used, but the examples will be targeted at the aforementioned software. When we looked at other provisioning tools such as xCAT or Warewulf much of the functionality that was interesting (image creation) was found to be simple enough to accomplish with scripts to build out the image rather than relying on external tools. With Warewulf changes to the image needed to be done with the Warewulf tools and these are only tracked by a MySQL datastore which we found hard to query and search through changes. The rest of the functionality in these tools with regards to DNS, DHCP, and TFTP server management we already had in place and so it made little sense to reinvent those wheels with processes already in place to manage those services.

In this paper we will present a model for stateless provisioning that could be used to provide a complete continuous integration and continuous delivery workflow for building a stateless computing environment without the use of a NFS shared root [4] or network block devices by employing the usage of cpio.gz compressed images. We will also demonstrate how this process can be used to gain capabilities that would

make it compelling for use in a secure computing environment as well as reducing the mean time to repair a node in certain failure cases.

II. CREATING THE IMAGE

In previous iterations of this provisioning process we relied simply on creating an empty OS directory tree structure on the TFTP server used to boot nodes from and applying configuration within a chroot environment. We then used a script that would go through and recreate the images based on the changed OS directories. This made updating the images fairly straight forward, however this also meant that there was not a formal process in place for testing the configuration changes before deploying an image to a test environment, or being deployed to production. We relied on a few spare servers to act as our development environment where the images would be tested first before being deployed to our production resources.

We relied on Git to keep a record of which files had changed in the OS directory so that we could roll back updates or changes when we needed to. This also provided a way to keep track of the revision of images built from the OS directory, by adding the Git commit hash to the image name after a successful build. This made it easy to then check what revision of an image a node was running by inspecting that file on each node.

With the incorporation of systemd into Redhat and CentOS we now have the ability to create an entirely separate namespace from the host OS using systemd-nspawn. This is an improvement over chroot in which several filesystems from the host OS needed to be bind mounted within the chroot environment (proc,sys,dev) in order to apply some changes. This gives us the opportunity to test our configuration management on a container created by systemd-nspawn first before deploying to a baremetal staging environment. This is a huge improvement over our previous method in which the entire OS directory tree was checked in to Git which made the repositories for each image type that we created very large after some time.

A. From Idea to Implementation to Verification

Building on previous work we wanted to have a process by which the computing environment is more tightly controlled using modern coding and deployment strategies typically used by web application developers. We were heavily inspired

by the way that the Finnish Grid and Cloud Infrastructure playbooks and roles are tested using Travis CI to test deployment of the configuration management code [5]. We however wanted not only to test that the configuration management code would apply cleanly, but also that it left the target host in a state that is consistent with what we have defined for our compute environment in compliance tests. We decided to use Gitlab and the associated gitlab-runner software to fulfill the needs for providing continuous integration and continuous deployment for the images for eventual deployment on production resources.

To accomplish this we setup gitlab-runner host(s) with the ability to spawn containers on demand using systemd's socket activation in which a container can be brought up once a ssh connection is initiated to the socket of the container where SSH is listening. This also allows the containers to only be active when continuous integration tests are running against it. Each role type has a container created for it on the gitlab-runner host. During the continuous integration pipeline run the Ansible configuration management code is run against the container using the SSH executor. We found early in development that the shell executor environment would not allow for the use of ssh-agent which is needed later to add key identities to ssh agent so that the CI process can ssh into the container from the gitlab-runner host.

Each new feature or improvement is given a unique issue within the project and specific issue branches are used to then track these changes. When an issue is first created we call out what files should ultimately be added or changed, any packages that may need to be installed, and services that may need to be configured, and lastly any new security rules we may need to introduce. We take that information and devise a set of Inspec tests that satisfy all the requirements of the issue and commit these first before beginning work on the configuration management code. After the Inspec compliance tests have been written, configuration code can then start to be committed and tested on the containers on the gitlab-runner hosts.

B. Virtualization and Container Tools

When we first started to look at introducing continuous integration into the building of our diskless images we initially tried to use VirtualBox to setup a baseline box that could be distributed. This turned out to be difficult as there were many issues standardizing the box environment, and several issues for people on the team that had a windows based laptop or PC where shared directories sometimes do not get mounted properly between the host and the VirtualBox host. We decided then that we wanted to try and use tools already packaged with the primary OS we deploy (Red Hat Enterprise Linux and CentOS) and so ended up with systemd-nspawn. We ultimately chose systemd-nspawn because of the ease of use in the tool, the fact that it fully replicates a separate namespace of systemd allowing process separation, and because it would allow us to eventually accomplish more elaborate tasks such as change the underlying architecture of the container. Docker was skipped

over simply because the purpose of the tool is to run a service, not a full blown init system and other OS processes [6].

C. Systemd-nspawn

In the past we used to utilize `yum -installroot=` to setup the OS directory tree and softlinking `/sbin/init` to `/init`. Occasionally we would need to mount `(proc,sys,dev)` from the host OS to the OS directory to then apply changes to the OS directory, but for most changes just a simple chroot sufficed. This was okay but issues such as problems with startup scripts, weren't uncovered until we went to deploy the image on a staging host.

With upgrading to RHEL/CentOS 7 we can now take advantage of systemd and systemd-nspawn to create several containers on a node to test multiple images that serve different roles. Each container starts a separate systemd startup process and boots the container in a separate namespace from that of the host's OS. We also set the containers up to only start up once a ssh connection has been made to the node allowing for containers to only be up when they are needed for the continuous integration process. We do this by utilizing systemd's socket activation strategy and assigning a different port to each container and creating a ssh config file that specifies the container alias and the port that ssh should use to connect to the container on the gitlab-runner host [7]. We additionally have found that no capabilities have to be added to the systemd-nspawn container in order to facilitate testing the configuration management code or running the compliance tests on the container.

For provisioning the image environment we first use a simple script that puts together an OS directory tree, and puts in place mechanisms to enable the OS directory tree to be bootable by systemd-nspawn as a containerized service. We setup all containers on the gitlab-runner host itself as we use the gitlab-runner run the continuous integration tests. We also setup the container with password-less ssh so that we can, as mentioned in the previous section, have Ansible use the SSH executor to deploy the configuration management code on the container and subsequently run the Inspec compliance tests to verify the state after the configuration run.

It is important to keep in mind that systemd-nspawn is only meant to replicate the startup up of services and is not meant as a full stock replacement of a virtual machine or bare metal host. When running a systemd-nspawn container the Host OS kernel is used and so the testing of the kernel and kernel related modules cannot be done in a systemd-nspawn container the way it could on a virtual machine or baremetal host.

Some components of your cluster may require kernel modules that will require that you rebuild packages that provide them. We suggest building a base image, deploying that to a test node, and then using that node to build out the kernel module packages and placing them in a local repository. Afterwards you can then include newly minted kernel module packages into the image and then deploy the image en masse to your compute nodes. We do this currently to build out the GPFS kernel module and Intel Omni-Path packages.

D. Configuration management

After the OS directory tree structure has been put in place and a systemd-nspawn container started using the directory tree we can then apply the configuration management code that we want to test and verify. Because systemd-nspawn allows us to instantiate a new kernel name space we can completely test all OS configuration management code as if the container was a bare metal host.

We use Ansible to apply the configuration management to OS directory tree structure using Ansible's built in capability of applying configuration to a chroot style OS directory tree. We additionally can use Ansible as an orchestration tool to change configuration files on production hosts as well as apply updates to statefull nodes. Roles can be created for the express purposes of rolling out orchestrated updates or pushing out rolling updates, for instance upgrading a group of Slurm controllers or GPFS NSD servers.

We do not interact with the containers themselves, and instead interact with them entirely through code that is committed to the git repository. Gitlab then spawns off gitlab-runner processes on the gitlab-runner host that spins up a container, applies the configuration management code to the container, and reports back whether or not the configuration management code applied correctly or whether there were errors.

III. TESTING THE IMAGE

For testing our configuration management code we use Gitlab and the integrated testing capabilities it provides with gitlab-runner. The workflow for working through a Gitlab issue is outlined in Figure 1 showing all the steps that are taken inside a feature branch.

As mentioned above when a need or an improvement is suggested we first create an issue in Gitlab which we use to centralize discussion regarding any changes that we will make to the configuration management code. We first create a new branch for the issue that we then commit in the Inspec compliance tests needed to accurately describe the state that is desired. Once we have determined that the compliance tests accurately describe all the changes that we want to check for in the final state of the image, we then begin to start work on crafting configuration management code that can modify the image to a state that would satisfy the compliance tests. This ensures that unless the compliance tests were crafted incorrectly, the image will only be built if all the configuration management code applied cleanly, and all the compliance tests pass.

When new code changes are introduced a new pipeline is created that is then deployed to the gitlab-runner associated with the repository. The gitlab-runner then spawns the necessary containers for each node image and applies the configuration management code to the containers. If the configuration management is able to be applied to the container without producing any errors the continuous integration process moves on to the next stage. The next stage requires that Inspec compliance tests are then run on the container so that we

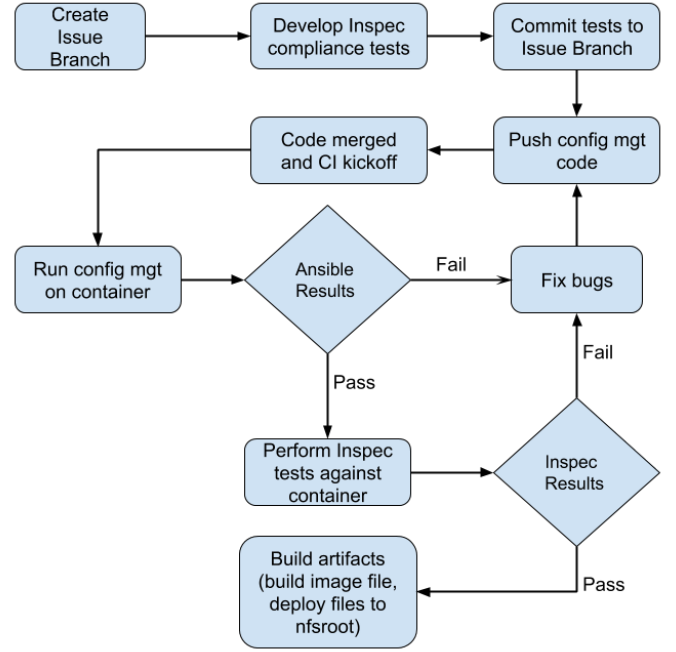


Fig. 1. Graphical representation of the workflow used in the CI process for creating node images.

can verify the state that the container is in after a successful run of Ansible. For our compliance tests we check everything about what we had just configured, from file user and group ownership, to contents of the files themselves, as well as services and whether they have been enabled.

Any failures in the application of the configuration management code would cause a failure for the job prompting that an image not be built until the errors have been resolved. Any subsequent failures in the evaluation of the Inspec compliance tests would also prevent an image from being built calling the errors to the attention of the administrator for tests that are still left unsatisfied by the configuration management code.

After the compliance tests have been completed and the tests have returned no errors, then the next stage of the continuous integration process is started which involves assigning some information regarding the git commit hash into a special file in /etc which can be used to track the image version, and packaging up the OS directory as a cpio.gz compressed image file.

IV. DEPLOYING THE IMAGE

Once the continuous integration process has completed we have our continuous delivery process then kick off which involves packaging up the OS directory tree into a cpio.gz formatted image which is then placed on the TFTPboot server or served over HTTP, and used to boot nodes into the modified image.

The pxelinux.cfg entry that we use points the kernel at the vmlinuz file we copy from the OS directory tree under /boot.

The `initrd` parameter is set to point to the `cpio.gz` image file with root set equal to `/dev/ram`. It is mounted read-write with the IP being derived from DHCP. The ramdisk size should be set to be as large or larger than the image size that is to be deployed, and a serial console specified with the right baud rate to enable SOL for the boot process.

Listing 1. Example `pxelinux.cfg`

```
serial 0 115200
prompt 1
timeout 20
ipappend 2
display pxelinux.msg
default linux

LABEL linux
    KERNEL curc/vmlinuz
    APPEND utf8 initrd=curc/curc-compute.cpio.gz
           root=/dev/ram ramdisk_size=1048386 rw ip=
           dhcp console=ttyS0,115200
```

After setting up the `pxelinux.cfg` file we can set up a DHCP entry similar to what can be seen below in listing 2 in order to have the stateless nodes boot and grab the hostname.

Listing 2. Example `dhcpd.conf` entry

```
host sknl0701 {
    hardware ethernet 00:1E:67:32:A9:73;
    fixed-address 10.225.7.44;
    option host-name "sknl0701";
}
```

Once we get the new image on the server along with the associated `vmlinuz` kernel file from the OS directory we can recreate softlinks that are setup so that the links now point to the new image and `vmlinuz`. We like to name our images by the role type followed by a git tag or git hash from the commit to differentiate the images. Any nodes rebooted at this point would now follow the link to the new image that was just deployed.

To deploy the the new image one need only reboot the node which upon reboot would grab the new image file. This makes it incredibly easy to do rolling updates for a cluster without impacting the users, and removing the need to take downtime for normal updates or to apply BIOS configuration or firmware updates.

A. Rolling Updates

We accomplish rolling updates when it is appropriate by simply pointing a softlink from the old image to the newly built one after it has been deployed. After doing so we utilize our node health check script (Warewulf NHC) [3] by setting a node to a state of DRAIN and setting a reason of REBOOT in Slurm. We modified a community provided NHC check that once NHC detects that the node has a status of DRAIN and IDLE it will execute a branch of the check that unmounts the parallel filesystems and issues a reboot [8]. Upon PXEing the node will boot into the new image that was deployed and

will set itself back online after NHC checks have cleared the node to be ready for jobs again. This rolling update strategy is ideal to also rollout security updates without having to take any downtime (this was used successfully to deploy kernel and firmware patches for Spectre and Meltdown).

B. Cold Booting a Cluster

When booting a large stateless cluster in the manner we have laid out consideration must be given to the amount of load not only on the TFTP server that is serving the images, but also to the network bandwidth available on the interface used to PXE. We highly suggest having multiple TFTP servers to boot subsections of nodes if the cluster size is greater than 400 nodes and you aim to boot the entire cluster from a cold state. We have found that a single gigabit interface can easily serve out 100 servers without taxing the network to heavily and prefer to bring racks up one at a time in most cases.

V. ALTERNATE ARCHITECTURE TESTING

One advantage of using `systemd-nspawn` over baremetal hosts is the ability to also test our configuration management code against multiple architecture types. This feature is valuable as research clusters begin to incorporate multiple computing architectures like Mare Nostrum which incorporates IBM POWER, Intel Knights Hill, and ARM v8 processors in a single computing environment [9]. This can be achieved with `systemd-nspawn` by adjusting the personality of the `systemd-nspawn` container. This feature is limited to testing of only x86 and x86-64 architectures for versions of `systemd` distributed from RHEL and CentOS repositories at this point, but as of version 233 of `systemd` [10], it now also supports ARM and ARM64 architectures among many others. We feel this will become increasingly important as certain architectures emerge as clear leaders for certain classes of computational problems and clusters become more fragmented in the architectures they provide to continue to meet researcher needs.

VI. ADVANTAGES OVER STATEFULL PROVISIONING

Updates can be applied once to an image and then the image distributed removing the need to apply the updates to X number of nodes that are in the cluster. Configuration drift is nonexistent since all nodes are using the same image [11]. Deployment of new nodes becomes as simple as gathering the MAC addresses for the interface used to boot the node via PXE and having the MAC addresses preloaded into DHCP and PXE configurations setup for each node. Replacing failed hardware becomes simpler allowing for hot spares to be provisioned and immediately deployed in some cases. It also allows for more finely controlled environments to be completely accounted for throughout the lifecycle of the image from idea to implementation.

A. Deployment of package updates

Updating statefull nodes can be a daunting task, and several problems can arise and compound on each other when having to apply updates to a large cluster of nodes. Package managers

such as yum make the task itself easy, but you can quickly get to a point where not all nodes get updates due to missing or conflicting packages which can cause an update to fail. These cases usually evolve from configuration drift where configuration management code is not applied cleanly to a node, often times by way of a configuration management agent on the node. Additionally when we want to update a package, we need to tell yum on each node of our cluster to search the configured repositories, download the updates, install, and verify the changes. This can lead to a lot of wasted time since these operations will take significantly longer to complete pending no errors arise with the update.

Even with mirrored repositories living on the local network, this can hammer on the network of your central node which in some environments perform other critical tasks. Yum also increase its cache on each node which will take extra steps to clear it out on each node after applying updates. Using stateless images, we can perform all of these actions once and be confident we have completed the changes successfully for all nodes in our cluster when we are ready to deploy the new image. We also save a substantial amount of time by performing said actions once for each node type, removing the need to wait for yum and the configuration management to complete updates and configuration changes.

B. Configuration Drift

When using stateless images, we do not have to worry about any configuration drift between nodes since all nodes are running the same image. After applying our updates and changes to a stateless image, we can pack it up, test it and deploy it to our cluster at large keeping everything in sync at all times providing a stable environment for users of the cluster. In a traditional statefull environment one can attempt to make the same changes and get them written out to disk, but often this falls short as nodes may fail to update if there are any hardware or network issues leaving nodes behind on updates that often have to be addressed by administrators (and provided these tasks do not get forgotten). Certain update failures may also result in the statefull node becoming insecure due to missed security patches or even their disk becoming corrupt due to incomplete changes if the connection drops. The failures that result in an incomplete update of a system that can result in a corrupted kernel often lead to a full node rebuild which in some cases requires the attention of a system administrator and at best requires that the node be out of production for a few hours while it rebuilds. In some cases we have even found corrupted library files from incomplete updates which caused issues in code run by users, which can damage the perception of stability of the cluster.

C. Deployment of new nodes

Clusters are often expanding over time, especially when new hardware becomes available. Being able to deploy new hardware quickly thus becomes a valuable capability for any research center that is constantly expanding. Before the hardware arrives, we can setup many of the changes that

need to be put in place to integrate the new hardware such as changes to Slurm configuration, addition of node health checks, and adding new DNS and DHCP entries. Before hardware ships we require the vendor provide MAC addresses for the interfaces on the node which we use to PXE boot the node off of. With the MAC addresses in hand we can populate DHCP with node entries for every node and link those new entries to a pxelinux.cfg that points to our stateless images. Once the nodes are received, inspected, and cabled for network and power, they can be brought online immediately for performance validation tests without having to wait through an excessive build and configuration process that is a hallmark of provisioning statefull systems.

D. Replacing failed hardware

In addition to deploying new nodes, replacing failed hardware becomes much simpler in a stateless environment. With spare nodes in a nearby rack, we can quickly swap bad hardware including an entire compute node without the need to rebuild a new OS onto disk. If even quicker turnaround time is required to get the downed node replaced, we can update our DHCP definitions to point at a spare node's MAC address that is already wired up in another rack and get it booted up within minutes. Utilizing these methods allows for a sole system administrator to manage a large environment without having to spend time provisioning nodes. From a monetary view, stateless environments seldom require hard drives creating opportunities to invest in other components of a cluster, especially in larger clusters. Removing the need for a hard drive reduces the costs in terms of maintainability and power, but also time a system administrator has to spend addressing the failed drive.

That said you could still have hard drives that are partitioned out separately to provide local scratch devices, swap, or other common filesystems and have the image mount the drive and the partitions on the drive. This can be invaluable to some sites where the user base does not have a lot of experience in checkpointing their jobs or are running codes that do not support checkpointing as they can partition out some of the disk to provide a local scratch device. The drive or drives could also be mounted by a process running on the node itself, such as BeeGFS to provide on demand scratch disk for jobs requiring a shared scratch space using BeeGFS on demand (BeeOND) [12].

E. Secure Computing Environments

Often within secure computing environments there is lengthy requirements not only on how the computing environment must be configured in order to meet security targets, but also how data gets passed to the secure computing environment. It is vitally important in these cases that all nodes be uniform in their deployment and general function as any updates can take time to fully evaluate, and eventually deploy to the cluster.

With our model, we can run a battery of tests to validate and confirm the security posture along with any capability tests.

For example we can include the CIS benchmarks [13] as part of our compliance test suite and use them in concert with our own site specific compliance tests. Using the Gitlab CI, we can confirm that all tests have passed and can create artifacts such as complete test results which can be disseminated to the appropriate stakeholders. Once we have an image we can also take a checksum of the image provide that as an artifact, which can be used for verifying the integrity of the image once it has moved over to the secure environment.

F. Configuration Management Tool migration

Having a full set of compliance tests like the ones that we have developed using Inspec means that we now have the state of our computing environment clearly defined outside the scope of our configuration management tools, making the process of migrating configuration management code from one tool far simpler. It allows us to, with a high degree of confidence, know that the configuration code written in one tool performs all the same functions as another configuration tool.

Additionally you could start off with no configuration management code at all, simply relying on a known good machine to develop the set of compliance tests that will fully describe the state of the node. One only need then plug in the tests into the Gitlab CI so that configuration code can then be written to satisfy the compliance tests ensuring that your configuration code always achieves the desired state.

VII. CAVEATS AND PITFALLS

Stateless images are not without its own set of pitfalls of which we will briefly discuss in the following subsections. Of the largest concerns is controlling the amount of space that is consumed by the host OS and the / filesystem, which resides in memory consuming space that could be used by researchers. Additionally having some sort of storage that can provide space to save the state for users and code that cannot checkpoint is also useful at sites where the user community often has new users or users that run codes that cannot checkpoint.

A. Image Bloat

Image bloat can be a concern for some sites in which there are a large amount of requirements on the computing environment that is presented to users. It can be a concern if you do not constrain your OS filesystem(s) to a set amount of memory or do not constrain memory either with custom cgroups or by configuration with your resource manager.

We have found that most of our images only consume 1 to 2 GB of memory and on our general compute nodes only consumes 1.2 GB of memory or just a little over 1% of total memory on each node. As memory density increases in nodes the root filesystem should consume a smaller fraction of memory.

In our default image we include many X11 utilities and libraries which could for most larger sites be excluded and only included on a visualization cluster. Additionally some

```
-bash-4.2# uname -a
Linux shas0101.rc.int.colorado.edu 3.10.0-862.11.6.el7.x86_64 #1 SMP Tue Aug 14 21:49:04
GNU/Linux
-bash-4.2# free -m
              total          used          free      shared  buff/cache   available
Mem:        128823          1037        124886         164         2899        126940
Swap:         0              0              0
```

Fig. 2. free -m output on a idle stateless node using our CPIO based image.

pruning can also be done to remove language packs and locales that may not be needed.

B. State for non-checkpointing codes

A large concern from a user perspective, especially for those who run codes that cannot or have not been setup to use checkpointing utilities is losing data. Because most files related to these types of jobs are stored in /tmp or other OS created filesystems it means that any work is lost when a node goes down.

It is highly recommended that you have a parallel scratch filesystem of some type to provide a way to store checkpoint files being used in current or upcoming jobs. If your cluster also has local disk attached to each node you could decide to partition it out to provide local scratch space on the node for temporary files generated by the job as well as provide swap space.

Ultimately it is necessary that you convey in as many ways as possible to your users that any information on a node that is not stored on the parallel filesystem or local scratch disk is wiped clean if the node goes down or is restarted.

VIII. ALTERNATIVES TO SYSTEMD-NSPAWN

While we do not recommend the following solutions for virtualizing or mimicking the OS environment we will talk about why we chose to use systemd-nspawn as the tool of choice to creating a development environment. It should be noted that we believe virtual box could be used as a replacement for systemd-nspawn but is seen in our eyes as wasteful since a more than capable tool is already provided by modern OS's that use systemd.

A. Docker

Docker is a container technology that piggyback's on top of the host OS allowing for programs and services to be run within the container. It has been popular for web development since you can package all files and configuration needed for running an application within the container. It is easy to think that Docker could serve as lightweight solution to test images, but it should be noted that docker was not designed to run an init system like systemd.

This is a huge disadvantage since we like many centers have services and scripts that execute as part of the init process in order to setup a node and prepare it for the production environment. These services and scripts might include mounting filesystems, or starting services related to the scratch parallel file system. The inability to effectively run systemd within a Docker container made it incompatible with our requirements for building and testing an image.

B. Virtual Box and Vagrant

Virtual Box is a tool that allows one to create virtual machines on a host and run them on hardware that support x86 or x86_64 architectures with virtualization support. Vagrant is a tool that is used to easily and quickly provision and configure virtual box based virtual machines and allows you to ssh into the virtual machine as if it were a bare-metal host. One downside of virtual box is that it can only run x86 or x86_64 guest OS's. While not a problem currently it would quickly become a problem if we needed to start supporting for instance ARM compute nodes and needed to test a OS destined to run on ARM processors [14].

A major obstacle to working with Virtual Box was difficulties getting virtual machines initially setup. We at first tried to set up the environment in such a way that a developer need only grab a vagrant configuration file and a vagrant box file and just run "vagrant up" from their development machine to setup a complete local environment. After getting a few team members setup with the vagrant configuration, meant to simplify the process of creating and destroying the box, we found some issues in that boxes sometimes would not get properly destroyed. Further we had one team member that had a windows based laptop that always had issues getting the shared directory that contained the configuration management code that would be applied locally to the box to reliably show up within the virtual machine.

IX. FUTURE WORK

One improvement that could be made to this already existing work is adding another set of tests that could be executed after the Inspec compliance tests in which we could build either all or a subset of user programs against the image to reveal any problems in software and library compatibility. Ideally this would be done on a set of development hosts: that way we can take advantage of all the cores on a production host to speed up compilation times and would be done as one of the final steps of validation for major updates to an image.

With a solid testing framework in place it would also be possible to expose a set of user editable parameters such that users or their research groups which may purchase nodes could edit the images used for their own nodes without having to consult with a system administrator. For instance they could define a set of packages that they would like to have installed in addition to what is provided in the base image. We would keep the majority of the configuration code in a separate repository and then would pull in their repository filled with variable files in order to build out their image.

Additionally the framework we have laid out could be used to teach new system administrators that are new to configuration management the fundamentals before having them take on more difficult tasks. In this use case a branch could be made (or cloned) that has tests already committed to it that tests for an expected state. The new system administrator would then have to commit into the branch the correct configuration code to get the container into the desired state as defined by the compliance tests. We used this to teach a student

system administrator some of the fundamentals of Ansible before having them port over configuration code from Puppet to Ansible.

ACKNOWLEDGMENTS

We gratefully acknowledge the computing resources provided on Blues and Bebop, high-performance computing clusters operated by the Laboratory Computing Resource Center at Argonne National Laboratory which served as the initial deployment platforms for this provisioning system. We also acknowledge the computing resources provided by RMACC Summit, a high-performance computing cluster operated by Research Computing at the University of Colorado Boulder.

REFERENCES

- [1] W. Scullin and A. Scovel, "Lessons from the ibm blue gene series of supercomputers," *Proceedings of the HPC Systems Professionals Workshop on - HPCSYSPRO17*, Nov 2017.
- [2] G. M. Kurtzer, "Documentation." [Online]. Available: <https://warewulf.github.io/warewulf3/>
- [3] M. Jennings, "mej/nhc," Jul 2018. [Online]. Available: <https://github.com/mej/nhc>
- [4] J. Garlick, "chaos/nfsroot," Mar 2015. [Online]. Available: <https://github.com/chaos/nfsroot>
- [5] "Cscfi/fgci-ansible," May 2018. [Online]. Available: <https://github.com/Cscfi/fgci-ansible>
- [6] J. Berkus, "Systemd vs. docker," Feb 2016. [Online]. Available: <https://lwn.net/Articles/676831/>
- [7] L. Poettering, "systemd for administrators, part xx," Jan 2013. [Online]. Available: <http://0pointer.de/blog/projects/socket-activated-containers.html>
- [8] J. Guldmyr, "check-reboot-slurm," Dec 2015. [Online]. Available: <https://github.com/mej/nhc/pull/6/files>
- [9] "Marenostrum iv - technical information." [Online]. Available: <https://www.bsc.es/marenostrum/marenostrum/technical-information>
- [10] Systemd, "systemd/systemd," Jun 2018. [Online]. Available: <https://github.com/systemd/systemd/blob/master/src/basic/architecture.h>
- [11] T. Vojta, "How to build a diskless cluster?" [Online]. Available: <http://web.mst.edu/vojta/pegasus/administration.htm>
- [12] "Beeond: Beegfs on demand." [Online]. Available: <https://www.beegfs.io/wiki/BeeOND>
- [13] "dev-sec/cis-dil-benchmark," Aug 2018. [Online]. Available: <https://github.com/dev-sec/cis-dil-benchmark>
- [14] Oracle, "Guest oses," Dec 2015. [Online]. Available: https://www.virtualbox.org/wiki/Guest_OSes

X. APPENDIX NOTES

Here we provide a few components of the continuous integration toolchain that we use to create a bare bones container that can be used for a specific node type that will then become an image. We also provide a sample Gitlab CI file that lays out the workflow that the gitlab-runner follows to test the Ansible configuration code, run the Inspec compliance tests, and finally package the OS directory used by the container into an image file which can be deployed to compute hosts.

APPENDIX A

SYSTEMD-NSPAWN CONTAINER CREATION SCRIPT

A. Abstract

This appendix includes information regarding the systemd-nsspawn script we use to create a container that can be used as part of the Gitlab CI process. We use this script to initially create a container and related service unit files for the host

such that the container can be accessed via SSH from the host OS. Included is the setup of some ssh keys to allow for passwordless root access from the Host OS to the container OS. You will still need to take care to also add a corresponding `/.ssh/config` entry and alias the container as you see fit.

B. Description

- **Program:** `systemd-nspawn`
- **Run-time environment:** Linux RHEL/CentOS
- **Output:** OS directory tree, `systemd` service and socket unit files

1) *How software can be obtained:* You can use the following example `systemd-nspawn` script to setup your own containers locally on any gitlabrunner host you wish to use to test images. The example script will setup a container, install the Centos 7.4 release in the OS directory we create, and install some groups of packages. Finally it will setup the `systemd` unit files within the container and on the host and will enable the socket target service on the host OS to allow SSH into the container.

Listing 3. Example script to create `systemd-nspawn` container

```
#!/bin/bash

# This script can be used to setup a container
# for a node type you wish to deploy
# The script assumes a destination to create
# the OS directories used to boot the
# container is
# fixed and located at /containers/
#
# John Blaas 2017 john.blaas@colorado.edu
#

node_type=$1
container_port=$2

if [ -z "$node_type" ] || [ -z "$container_port" ]
then
    echo "Please provide a name and port for
    the node type you wish to create."
    echo "Node type should be provided as the
    first argument and port as the"
    echo "second argument into this script"
    exit 1
else
    echo "Preparing to create $node_type
    container"
fi

mkdir -p /containers/$node_type

## This following section could likely be
## broken out into a separate script or
## scripts if you want to deploy
## against other OS's. This way you could add
## another parameter to call with this script
## that could then
## go out and find the corresponding OS
## specific script to create the base OS
## directory.

if [ ! -f centos-release-7-4.1708.el7.centos.
x86_64.rpm ]; then
    wget http://vault.centos.org/7.4.1708/os/
x86_64/Packages/centos-release
-7-4.1708.el7.centos.x86_64.rpm
fi

if [ ! -f RPM-GPG-KEY-CentOS-7 ]; then
    wget http://vault.centos.org/7.4.1708/os/
x86_64/RPM-GPG-KEY-CentOS-7
fi

rpm --import ./RPM-GPG-KEY-CentOS-7
rpm --root=/containers/$node_type -ivh centos-
release-7-4.1708.el7.centos.x86_64.rpm
cp RPM-GPG-KEY-CentOS-7 /etc/pki/rpm-gpg/RPM-
GPG-KEY-CentOS-7
cp RPM-GPG-KEY-CentOS-7 /containers/$node_type
/etc/pki/rpm-gpg/RPM-GPG-KEY-CentOS-7

yum --installroot=/containers/$node_type
groupinstall base -y
yum --installroot=/containers/$node_type
install openssh-server -y

## This portion sets up systemd unit files to
## enable us to be able to ssh
## into the containers on different socket
## activated ports for both the host OS
## and the OS running within the container
## that is provisioned

## This creates the host service file for
## starting the container
cat > /etc/systemd/system/$node_type.service << EOF

[Unit]
Description=$node_type container

[Service]
ExecStart=/usr/bin/systemd-nspawn -jBD /
containers/$node_type 3
KillMode=process
EOF

## This creates the sshd socket service on the
## host for port activation
cat > /etc/systemd/system/$node_type.socket <<
EOF

[Unit]
Description=The SSH socket of $node_type
container

[Socket]
ListenStream=$container_port
EOF

## This creates the sshd socket service in the
## container
cat > /containers/$node_type/etc/systemd/
system/sshd.socket << EOF

[Unit]
Description=SSH Socket for Per-Connection
Servers
```



```

[Socket]
ListenStream=$container_port
Accept=yes
EOF

## This creates the sshd service file to
  accept forwarded connections
cat > /containers/$node_type/etc/systemd/
  system/sshd@.service << EOF
[Unit]
Description=SSH Per-Connection Server for %I

[Service]
ExecStart=-/usr/sbin/sshd -i
StandardInput=socket
EOF

## Softlink the sshd socket to bring up the
  service on container boot

mkdir -p /containers/$node_type/etc/systemd/
  system/sockets.target.wants
ln -s /containers/$node_type/etc/systemd/
  system/sshd.socket /containers/$node_type/
  etc/systemd/system/sockets.target.wants/
  sshd.socket

## Finally create a ssh keypair for root to
  login to the container and setup
  sshd_config
mkdir -p /containers/$node_type/root/.ssh
chmod 700 /containers/$node_type/root/.ssh

echo "Generating root keypair now, leave the
  passphrase blank for passwordless
  authentication"
ssh-keygen -t rsa -b 4096 -f /root/.ssh/
  $node_type-rsa -q
cat /root/.ssh/$node_type-rsa.pub >> /
  containers/$node_type/root/.ssh/
  authorized_keys

## Here we place a basic SSHD configuration so
  that we can ssh into the container

cat > /containers/$node_type/etc/ssh/
  sshd_config << EOF
HostKey /etc/ssh/ssh_host_rsa_key
HostKey /etc/ssh/ssh_host_ecdsa_key
HostKey /etc/ssh/ssh_host_ed25519_key

SyslogFacility AUTHPRIV

AuthorizedKeysFile .ssh/authorized_keys

HostbasedAuthentication yes
PasswordAuthentication yes
ChallengeResponseAuthentication yes

PermitRootLogin without-password

UsePAM yes

X11Forwarding yes

UsePrivilegeSeparation sandbox # Default for

```

new installations.

```

AcceptEnv LANG LC_CTYPE LC_NUMERIC LC_TIME
  LC_COLLATE LC_MONETARY LC_MESSAGES
AcceptEnv LC_PAPER LC_NAME LC_ADDRESS
  LC_TELEPHONE LC_MEASUREMENT
AcceptEnv LC_IDENTIFICATION LC_ALL LANGUAGE
AcceptEnv XMODIFIERS

Subsystem sftp /usr/libexec/openssh/sftp-
  server
EOF

## Finally, lets put in place the softlink
  from sbin/init to init to make the final
  image bootable

cd /containers/$node_type
ln -s sbin/init ./init

## Great news everyone! We are done! Remeber
  to setup a ssh config alias for the
  container

echo "The $node_type container is now prepared
  "

```

2) *Software dependencies:* Redhat or CentOS based host OS and at least systemd version 219.

C. Installation

To install this script simply copy the script in full and place it on the gitlab-runner server and grant the appropriate execute permissions. Once executed you should be left with a ready OS directory tree, and the ability to ssh into the container using the socket activation provided.

D. Evaluation and expected result

You should be left with a complete OS directory located at /containers/*image name*, on the gitlabrunner host. You will need to setup ssh config into the host to allow for gitlab-runner to ssh into the container as root. After SSH'ing into the container or trying to, the container service will remain active unless you have placed a cron script within the container that shuts the machine down after a period of inactivity. To poweroff the container manually you can issue `machinectl poweroff container-name`, which will poweroff the container. As long as the socket service for the container is still running on the host OS, you should be able to spin the container up again by only starting a ssh connection to the container.

E. Future Work

It would be useful to break out the portion of code which sets up the base OS directory such that we could pass in and OS name and version to the script and have the script execute a sub-script which corresponds with the OS and version that the user has requested. For instance instead of using `rpm --root=` like we normally use to provision the base OS directory for a CentOS based image, we could have an alternate script for debian based images that would call `debootstrap`.

APPENDIX B

GITLAB CI CONFIGURATION EXAMPLE

A. Abstract

This appendix includes information regarding the `.gitlab-ci.yml` file we use to test our compute node images within the Gitlab Continuous Integration software. This file is read by the gitlab-runners which instructs them on what actions to take in order to test the code in the repository.

B. Description

- **Program: Gitlab**
- **Run-time environment: Linux RHEL/CentOS**

1) *How software can be obtained:* You can use our example `.gitlab-ci.yml` file which uses stages to test our compute image. In the first stage we apply the Ansible configuration code to a systemd-nspawn container, before we then move on to evaluate the container using Inspec compliance tests to verify the state of the container after the configuration code has run. Finally after the compliance tests have finished the CI process moves on to the next stage which packages up the OS directory for the container into a cpio root image that is compressed into a tar.gz archive. This example focuses on testing only a image named compute.

You can find the example `.gitlab-ci.yml` in Listing 2 below.

Listing 4. Example Gitlab CI file

```
# CPIORoot CI

stages:
  - build
  - compile

test-config:
  stage: build
  script:
    - ansible-playbook -i Production production
      .yaml
    - rvm use ruby-2.4.0
    - eval $(ssh-agent -s)
    - ssh-add
    - inspec exec tests/base -t ssh://
      root@compute:23 --log-level=warn

build-image:
  stage: compile
  script:
    - cd /containers/compute
    - echo $CI_COMMIT_REF_NAME $CI_COMMIT_SHA >
      /etc/image-release
    - find | cpio -oc | gzip > /images/curc-
      compute.cpio.gz
```

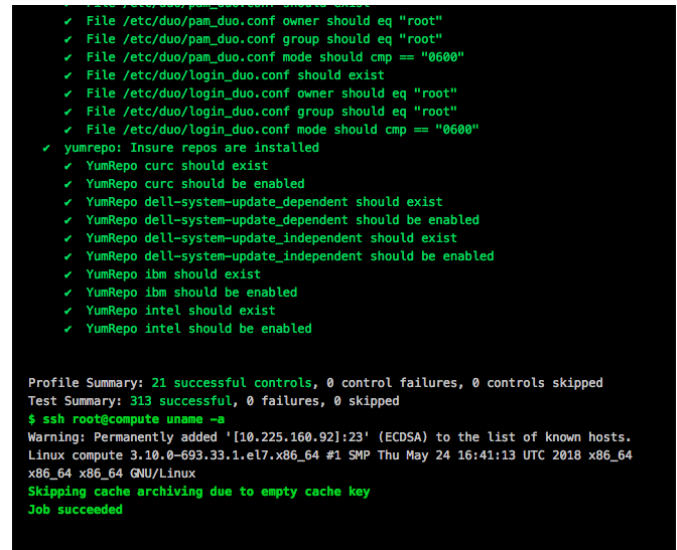
2) *Software dependencies:* Gitlab version 8 or higher

C. Installation

To install this simply create a file with the name `.gitlab-ci.yml` in the root directory of your git repository. Then commit the file to your Gitlab repository and confirm within the GUI that there are no syntax errors within the file.

D. Evaluation and expected result

This should allow you to start testing Ansible based code given that you have SSH access setup between the gitlab-runner host and the container. This also assumes that you have some Inspec tests created and placed inside the same repository as the Ansible code. You should see results like shown in Figure 2 if you have included Inspec compliance tests.



```
✓ File /etc/duo/pam_duo.conf owner should eq "root"
✓ File /etc/duo/pam_duo.conf group should eq "root"
✓ File /etc/duo/pam_duo.conf mode should cmp == "0600"
✓ File /etc/duo/login_duo.conf should exist
✓ File /etc/duo/login_duo.conf owner should eq "root"
✓ File /etc/duo/login_duo.conf group should eq "root"
✓ File /etc/duo/login_duo.conf mode should cmp == "0600"
✓ yumrepo: Insure repos are installed
✓ YumRepo curc should exist
✓ YumRepo curc should be enabled
✓ YumRepo dell-system-update_dependent should exist
✓ YumRepo dell-system-update_dependent should be enabled
✓ YumRepo dell-system-update_independent should exist
✓ YumRepo dell-system-update_independent should be enabled
✓ YumRepo ibm should exist
✓ YumRepo ibm should be enabled
✓ YumRepo intel should exist
✓ YumRepo intel should be enabled

Profile Summary: 21 successful controls, 0 control failures, 0 controls skipped
Test Summary: 313 successful, 0 failures, 0 skipped
$ ssh root@compute uname -a
Warning: Permanently added '[10.225.160.92]:23' (ECDSA) to the list of known hosts.
Linux compute 3.10.0-693.33.1.el7.x86_64 #1 SMP Thu May 24 16:41:13 UTC 2018 x86_64
x86_64 x86_64 GNU/Linux
Skipping cache archiving due to empty cache key
Job succeeded
```

Fig. 3. Screenshot showing end results from our internal Gitlab CI for the compute node image.

This Gitlab CI file assumes a very basic repository in which only one node image is created. When dealing with multiple images consideration should be taken into account with regards to the CI file such that you are effectively only building and testing against one image. There are several ways you can go about doing this, with the first approach being setting up a staging or testing branch for each node type. Each staging or testing branch would have a Gitlab CI file in it, with the production branch of the repo having no such file, this way you can rebase changes from the production branch where you should aim to store all Ansible code and possibly the Inspec compliance code. In each staging or testing branch the Gitlab CI file would only apply Ansible code for the node image type it controls and then only runs the Inspec compliance tests on the container that matches the node type.

The second approach you could take is to separate the Ansible configuration and Inspec compliance test code into a stand alone repository. Then for each node type create a repository that only contains a set of group_vars files to be used by Ansible to apply configuration, and the Gitlab CI file targeting the node type. The Gitlab CI file should be changed to then add a step to clone the Ansible configuration and Inspec compliance test code and use the node type repository's group_vars to bring in the required variables. This second approach can be useful if you want to (within reason) allow for the shareholders (condo style deployment of resources in

mind) of a certain node type to make changes to their image without the need of system administrator intervention. If for instance you have setup an Ansible play that pulls in a list of packages from `group_vars` to install, the shareholders could conceivably edit those packages within the `group_vars` and have a new image built once the new variables have been committed.

APPENDIX C

NHC ROLLING UPDATE SCRIPT

A. Abstract

NHC is a great tool for ensuring that nodes in a cluster have passed a bare minimum of tests to ensure that the node is ready to accept jobs from the job scheduler and to check throughout the duration of a job whether there has been a hardware failure or regression in configuration.

This script was modified based on work done by Johan Guldmyr as part of a merge request submitted to the NHC code base on Github to enable rolling reboots of nodes, specifically in diskless clusters [8]. To it we added site specific commands that we needed in order to safely bring down and reboot the node as well as indicating in our logs and metrics that a rolling update was taking place. With a little more effort this script could easily be used on stateful clusters as well with some thought put into site specific actions that must happen on a node before it can be safely brought in to production.

B. Description

- **Program:** Bash, NHC, Slurm
- **Run-time environment:** Linux RHEL/CentOS

Listing 5. Rolling Update script

```
#!/bin/bash
##
# SLURM health check program
# ulf.tigerstedt@csc.fi 2012
# johan.guldmyr@csc.fi 2014
# john.blaas@colorado.edu 2017
#
# Usage:
# scontrol update node=ae5 state=drain reason=
#       reboot
# What happens:
# When the node is drained slurm reason field
#   is changed to "rebooting" and then it is
#   rebooted.
# When slurm on the node is back online nhc
#   will on the next run resume the node.

FAILED=0
ERROR=""
HOSTNAME=`hostname -s`
DEBUG=""
TIMEOUT=900

STATELINE=`scontrol -o show node $HOSTNAME`
# Check if this is a SLURM worker node at all
if [ $? = 1 ] ; then
    #echo Not a slurm node
    exit
fi

if [ "$1" = "-d" ]; then
    DEBUG="1"
fi

check_reboot_slurm() {
    # The name of this function is defined in nhc.
    # conf as a check.

    for a in $STATELINE; do
        LABEL=`echo $a | cut -d = -f 1`
        PARAMETER=`echo $a | cut -d = -f 2`

        if [ $LABEL = "Reason" ]; then
            REASON=$PARAMETER
        fi
        if [ $LABEL = "State" ]; then
            STATE=$PARAMETER
        fi
    done
    if [ -n "$DEBUG" ]; then echo Slurm thinks
        $HOSTNAME has STATE=$STATE and REASON=
        $REASON; fi

    if [ "$REASON" = "rebooting" ]; then
        if [ "$STATE" = "DOWN+DRAIN" -o "$STATE" = "
            IDLE+DRAIN" ]; then
            if [ -n "$DEBUG" ]; then echo Resuming
                after reboot ; fi
            # Bring up GPFS before RESUMING service
            /usr/lpp/mmfs/bin/mmstartup
            # Sleep for a bit to wait for GPFS to
            #   make connections to the NSD servers
            sleep 30
            # Now mount all GPFS filesystems
            /usr/lpp/mmfs/bin/mmmount all
            # Create an event in our slurm InfluxDB
            #   database
            curl -i -ss -XPOST "http://influx1
                :8086/write?db=slurmlog" --
                data-binary 'slurm,host=' "$
                $HOSTNAME" title="Rolling
                Reboot",message="Host ready
                for production",tags="slurm"
            # Set node back to production
            scontrol update NodeName=$HOSTNAME state=
                RESUME reason=""
        fi
    fi

    if [ "$REASON" = "gpfs-restarting" ]; then
        if [ "$STATE" = "DOWN+DRAIN" -o "$STATE"
            = "IDLE+DRAIN" ]; then
            if [ -n "$DEBUG" ]; then echo
                Resuming after reboot ; fi
            # Bring up GPFS before RESUMING
            #   service
            /usr/lpp/mmfs/bin/mmstartup
            # Sleep for a bit to wait for GPFS
            #   to make connections to the
            #   NSD servers
            sleep 30
            # Now mount all GPFS filesystems
            /usr/lpp/mmfs/bin/mmmount all
            # Create an event in our slurm
            #   InfluxDB database
            curl -i -ss -XPOST "http://influx1
                :8086/write?db=slurmlog" --
```

```

        data-binary 'slurm,host=' "$HOSTNAME" title="GPFS
        Configuration Complete",
        message="Host ready for
        production",tags="slurm,gpfs"
        # Set node back to production
        scontrol update NodeName=$HOSTNAME
        state=RESUME reason=""
    fi
fi

if [ "$REASON" = "gpfs-restart" -a "$STATE" =
"IDLE+DRAIN" ]; then
    if [ -n "$DEBUG" ]; then echo Rebooting
    ; fi
    sleep 2
    # Shutdown GPFS and give it 30 seconds
    to complete
    mmshutdown
    sleep 30
    # Create an event in our slurm InfluxDB
    database
    curl -i -ss -XPOST "http://influx1:8086/
    write?db=slurmlog" --data-binary '
    slurm,host=' "$HOSTNAME" title="GPFS
    Configuration Starting",message="
    GPFS configuration being applied",
    tags="slurm,gpfs"
    # stop slurm to ensure another NHC does
    not run again while we are updating
    service slurm stop
    sleep 2
    # Set the slurm reason
    scontrol update NodeName=$HOSTNAME state
    =DOWN reason="gpfs-restarting"
    # Run puppet agent to pull in new config
    , part of our config forces slurm to
    be started again
    puppet agent -t
fi

if [ "$REASON" = "reboot" -a "$STATE" = "IDLE+
DRAIN" ]; then
    if [ -n "$DEBUG" ]; then echo Rebooting ; fi
    sleep 2
    curl -i -ss -XPOST "http://influx1:8086/
    write?db=slurmlog" --data-binary '
    slurm,host=' "$HOSTNAME" title="
    Rolling Reboot",message="Rebooting
    for Updates",tags="slurm"
    # Shutdown GPFS and give it 30 seconds
    mmshutdown
    sleep 30
    # stop slurm just in case
    service slurm stop
    sleep 2
    scontrol update NodeName=$HOSTNAME state=DOWN
    reason=rebooting
    /sbin/reboot
fi

if [ -n "$DEBUG" ]; then echo Health check
done; fi

}

```

1) *Software dependencies:* Slurm 15+ and Node Health Check from Warewulf

C. Installation

To install this script just copy script and place it in the directory /etc/nhc/scripts. Once you have done that you then will need to add an entry into the nhc.conf file on the node that executes the check on each node you wish to have the ability to do rolling updates on. After adding the entry and saving the file you will now be able to have the node perform rolling updates.

Care should be taken to also make adjustments to the rolling reboot script in order to effectively manage the shutdown and startup of the node. For instance in our environment we use GPFS as a parallel storage and scratch space, so we have added in commands to properly shutdown GPFS before taking down the node to avoid polluting our logs with GPFS expel notices, but to also insure that GPFS completes all pending writes on a node. Additionally we also have included commands that insert an event into an InfluxDB database which we use to provide annotations on our node view dashboards as to when a node was brought down for a rolling update and when it is returned to production.

D. Evaluation and expected result

You can use this script/check for NHC to effectively apply updates on nodes, whether they be BIOS changes, upgrading the image itself for diskless clusters, or for simply applying changes which do not need to be applied to all nodes in a cluster at once. To start the process you will only have to set a node in Slurm to have a state of DRAIN and assign the reason as "reboot". Once that has been done each time the NHC script executes on the node it will check whether the node is in a state of IDLE and DRAIN.

Once jobs have completed on the node and the node enters the IDLE and DRAIN state the NHC will start the shutdown process, change it's reason in Slurm and promptly shutdown. After rebooting, NHC will once again execute as soon as the Slurm service starts again, and the commands under the "rebooting" section of the check will be executed in order to bring the node back into a state that should allow it to return to production.

It is important to note that this script assumes that you have setup the nodes in your cluster to come up in a state that allows the slurm service to start without errors.