Decoupling OpenHPC Critical Services

Jacob Chappell
jacob.chappell@uky.edu
University of Kentucky

Bhushan Chitre
bhushan.chitre@uky.edu
University of Kentucky

Vikram Gazula
vgazu2@uky.edu
University of Kentucky

Lowell Pike
pike@netlab.uky.edu
University of Kentucky

James Griffioen
griff@uky.edu
University of Kentucky

ABSTRACT

High-Performance Computing (HPC) cluster-management software often consolidates cluster-management functionality into a centralized *management node*, using it to provision the compute nodes, manage users, and schedule jobs. A consequence of this design is that the management node must typically be up and operating correctly for the cluster to schedule and continue executing jobs. This dependency de-incentivizes administrators from upgrading the management node because the entire cluster may need to be taken down during the upgrade. Administrators may even avoid performing minor updates to the management node for fear that an update error could bring the cluster down.

To address this problem, we redesigned the structure of management nodes, specifically OpenHPC's System Management Server (SMS), breaking it into components that allow portions of the SMS to be taken down and upgraded without interrupting the rest of the cluster. Our approach separates the time-critical SMS tasks from tasks that can be delayed, allowing us to keep a relatively small number of time-critical tasks running while bringing down critical portions of the SMS for long periods of time to apply OpenHPC upgrades, update applications, and perform acceptance tests on the new system.

We implemented and deployed our solution on the University of Kentucky's HPC cluster, and it has already helped avoid downtime from an SMS failure. It also allows us to reduce, or completely eliminate our regularly scheduled maintenance windows.

KEYWORDS

OpenHPC, Singularity, HPC, clusters, containers

ACM Reference Format:
Jacob Chappell, Bhushan Chitre, Vikram Gazula, Lowell Pike, and James Griffioen. 2019. Decoupling OpenHPC Critical Services. In *HPCSYSPROS '19: SIGHPC Systems Professionals Workshop, November 22, 2019, Denver, CO.* ACM, New York, NY, USA, 7 pages. https://doi.org/TBD

1 INTRODUCTION

High-Performance Computing (HPC) cluster toolkits are often designed around a *management node* that hosts a variety of services used to manage and control the HPC cluster [2, 5, 7, 9, 15]. Example services performed by the management node include provisioning compute nodes, configuring and managing cluster networks, synchronizing users and groups, scheduling jobs, and monitoring the state of the cluster. While a centralized management node simplifies management tasks, it increases the cluster's vulnerability to availability and reliability issues. Because the management node is typically the "brains of the cluster", failure of the management node or loss of connectivity to the management node usually leads to widespread failure across the entire cluster. Even planned maintenance windows of the management node could result in downtime for the entire cluster, and downtime can be disastrous for HPC centers where jobs are long-lived and constantly compete for access to an insufficient number of compute cycles. Unused compute cycles translate to lost time that could have been used for research.

OpenHPC is a relatively new cluster toolkit that has been growing in popularity. OpenHPC relies on a centralized management node called the *System Management Server (SMS).* Like other cluster toolkits, the SMS is a complex and critical node that provides a long list of services ranging from provisioning, user and group management, file management, scheduling, monitoring and more. In OpenHPC, if the SMS goes down the whole cluster would come to a halt due to the fact that the Slurm controller daemon on the SMS must communicate with the Slurm daemons on each compute node at least once every five minutes, or the compute nodes will kill their own jobs. Furthermore, if the SMS is in the process of routing traffic or performing other network operations on behalf of compute nodes, these services will be interrupted even if the SMS looses network connectivity.

In order to avoid SMS downtime due to hardware failures, we execute the SMS in a *Virtual Machine (VM)*[1] so that it can be quickly restarted on a new host node when the current host node fails or needs to be taken down for maintenance. We typically execute the SMS on a "primary" host node, with a "standby" host node ready to take over in case the primary fails. Ideally the standby would be a hot standby that is kept in sync with the primary, but that feature is not supported by OpenHPC SMS nodes. Another benefit of a virtualized SMS is the ability to take snapshots of the SMS VM – i.e., copies of the VM's virtual disk drives and hardware device settings. These snapshots give us peace of mind knowing that we

[1]We used VirtualBox [8] for our SMS, but other virtualization software could be used.

can quickly restore the SMS to a previous point in time if the SMS itself becomes corrupted due to human or hardware errors.

While virtualizing the SMS is a good first step, it still does not prevent updates or minor changes from going wrong and bringing down the entire cluster, nor does it help shorten the amount of time the SMS must be taken down to do a complete cluster upgrade. Although VM snapshots of the SMS allow the SMS to be easily rolled back and restarted, cluster administrators are hesitant to take frequent VM snapshots because they require shutting down the VM to ensure a consistent snapshot. Infrequent snapshots, if used, result in a system that is out of date (e.g., missing user/group accounts, lost logs and scheduling information, and so on). Taking frequent snapshots, on the other hand, not only causes increased downtime, but also increases the amount of storage needed to hold past snapshots.

Given these problems, our goal was to decouple the fate of the SMS from the fate of the HPC cluster such that SMS downtimes would have little or no effect on the availability of the HPC cluster as a whole. To achieve this, we developed an approach that decomposes the SMS node into multiple components to enable greater availability and reliability of the cluster. Our hypothesis was that by decoupling the SMS' time-critical services – i.e., services that need to respond immediately to requests from compute nodes – from the SMS' non-time-critical services – i.e., services that could be delayed, such as account creation or software updates – we could dramatically reduce the risk and day-to-day challenges of operating and upgrading the cluster. While the ability to perform minor updates may not be all that impressive, the ability to perform major upgrades of the SMS without any interruption of service to users is a major win for both users and administrators. Cluster administrators who typically work frantically to upgrade the SMS during a planned downtime can now take their time to complete the upgrade carefully and correctly.

2 OPENHPC

OpenHPC is a system designed to reduce the effort required to install and manage an HPC cluster. It is based on a collection of package repositories containing software development tools, scientific libraries, resource managers, and other utilities common to HPC environments. OpenHPC currently provides repositories for the CentOS and SLES distributions of GNU/Linux, periodically releasing new versions after they have been carefully validated and tested on real HPC clusters.

To setup an OpenHPC cluster, the cluster administrator must first install CentOS (or SLES) on a node that will become the SMS management node. After CentOS is installed, the OpenHPC package repository must be enabled, and a base set of OpenHPC packages – and any additional optional packages needed by the cluster – must be installed and configured. Once the software is in place, the SMS is able to provision the login, compute, and data transfer nodes in the cluster with the appropriate operating system and configuration files. An illustration of the OpenHPC architecture is shown in Figure 1.

The SMS requires at least two Ethernet interfaces, one for communicating internally among the provisioned nodes and another for communicating with the public Internet. A third Ethernet interface

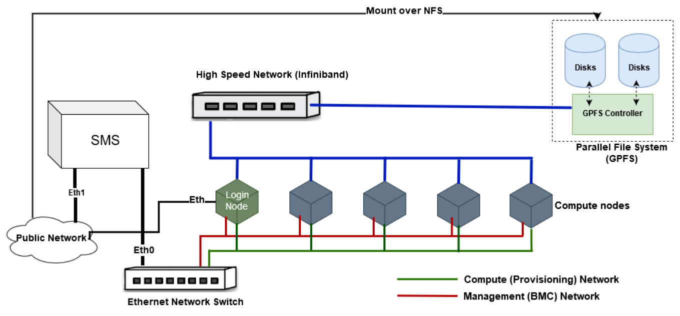

Figure 1: OpenHPC cluster architecture

can be used to communicate with the *Baseboard Management Controller (BMC)* of the compute nodes if they support it. The Ethernet interfaces can be virtual by using 802.1Q VLAN tagging which is supported in the Linux kernel and server-grade Ethernet switches. In addition to the Ethernet interfaces, high-speed InfiniBand interfaces are commonly used to support *Message Passing Interface (MPI)* and data-transfer applications, as well as access to a parallel file system like Lustre [6] or IBM's *General Parallel File System (GPFS)* [4]. While OpenHPC provides optional components for easily connecting to Lustre, GPFS is not natively supported.

To provision compute nodes and synchronize files and users, OpenHPC packages and leverages the time-tested Warewulf [13] system. Warewulf is a scalable systems management suite designed to manage large-scale HPC environments. Cluster administrators can configure Warewulf to be used for provisioning, BMC-management through the *Intelligent Platform Management Interface (IPMI)*, file and user synchronization, triggered event handling, and more. Warewulf supports both stateful and stateless (diskless) configurations for provisioned nodes.

Warewulf also supports configuring and provisioning individual compute nodes or groups of compute nodes with selective software. Cluster administrators can compile blob images which encapsulate a *Virtual Node File System (VNFS)* that consists of the necessary software and startup services for individual compute nodes. When a compute node powers on, it reaches out to the SMS via the *Preboot Execution Environment (PXE)* [14] protocol by issuing a *Dynamic Host Configuration Protocol (DHCP)* request to get its *Internet Protocol (IP)* address, followed by a request for a kernel bootstrap and subsequent VNFS image. The VNFS image is downloaded via PXE and loaded into the memory of the compute node, so it is vital to keep the VNFS as lean as possible. Once downloaded, the compute node boots from the VNFS image. After booting, files which change frequently can be synchronized using Warewulf's file database.

With a systems management suite like Warewulf, a scheduler like Slurm, and a variety of common scientific software, OpenHPC makes it relatively painless to spin up a quick and simple HPC cluster. However, it is then up to cluster administrators to selectively tweak and configure their own cluster environments. Maintaining and upgrading an OpenHPC SMS can be challenging, as will be discussed in Section 3.

3 UPGRADE CHALLENGES

At the University of Kentucky, our *Lipscomb Compute Cluster (LCC)* is the university's flagship supercomputer, servicing hundreds of researchers across the state of Kentucky. LCC has become the critical infrastructure that many users depend on for their day-to-day work and research. As such, we struggle to take the cluster offline for even short periods of time. Also planned maintenance windows represent a significant disruption to jobs – many that run for days or weeks.

Our previous cluster was powered by the Rocks cluster toolkit, which is slow-moving software with infrequent updates, and so there was little need for maintenance windows. LCC, on the other hand, is powered by OpenHPC which releases updates quarterly. While one can argue that quarterly updates may be a bit excessive, OpenHPC is advancing rapidly with each new release, and so there is a reason to try to apply upgrades as soon as possible and stay on the bleeding edge.

While some cluster administrators may be willing to risk live cluster upgrades, they are not recommended and even seemingly small glitches in the upgrade can lead to major problems and cluster downtime. For example, one of the previous OpenHPC releases provided an upgraded version of Slurm which broke compatibility with some of our custom workflow utilities. We thankfully performed said upgrade during a maintenance window. Had we performed the upgrade live, it would have resulted in killed jobs and major corruption to our Slurm accounting database.

Because of the risk in performing live upgrades and the fact that lost jobs is more undesirable than scheduling a maintenance window, we have historically taken the entire cluster down for up to a week to upgrade system software, build new VNFS images and bootstraps, upgrade associated application software, and test the upgrade to ensure it is working correctly and hasn't broken any custom workflow scripts. Furthermore, we would take the downtime as an opportunity to snapshot our SMS VM and backup other software as needed.

Having four scheduled maintenance windows per year would result in a massive number of compute cycles being wasted while the cluster is down. Therefore, we needed a way to safely perform live upgrades without the risk of affecting running jobs or the overall health of the cluster.

4 DECOUPLING SMS SERVICES

In order to take down portions of the SMS for long periods of time while allowing others portions of the SMS to continue supporting the compute nodes, we began by identifying the SMS services that are time-critical and must remain operating at all times (i.e., services that compute nodes interact with regularly and expect an immediate response from). Our idea was that if time-critical services can be separated and operated independently from the non-time-critical services (i.e., services that can be delayed and performed at times of the cluster administrator's choosing), then we can perform the upgrades of the two sets of services independently. In particular, we predicted that the most time-consuming tasks of an upgrade would be associated with the non-time-critical services which could be taken down without impacting the entire cluster.

To partition services into time-critical services and non-time-critical services, we monitored the provisioning network to detect network traffic originating from or destined to the SMS. SMS services that initiated new network connections to compute nodes were identified as non-time-critical and placed into the group we called *SMS-mgmt*, while SMS services that accepted new network connections from compute nodes were placed into the group we called *SMS-services*.

We began our analysis by selecting a compute node to monitor, and we traced its network traffic from the moment it was powered on. Because our compute nodes are stateless, the first network action a compute node takes while booting is to issue a DHCP request and download a small bootstrap image and subsequent VNFS image from the SMS. While the compute node initiates the provisioning request, the request happens rarely and only at the discretion of a system administrator who reboots or powers on a new compute node. Therefore, we placed the provisioning service into the SMS-mgmt group.

After the compute node had been fully provisioned, we ran tcpdump on our SMS, capturing only the network traffic destined to, or originating from, the MAC address of the compute node we selected to monitor. We kept the tcpdump session open for an extended period of time in order to capture both frequent and infrequent network traffic in order to better profile the network services utilized on our cluster. Upon terminating tcpdump and analyzing the captured packets, we identified network traffic corresponding to the following services:

(1) HTTP requests to synchronize files from the Warewulf database,
(2) Traffic from the forwarding of syslog messages via rsyslog,
(3) *Network File System (NFS)* traffic to access files stored on the SMS,
(4) Traffic to external destinations being routed through the SMS to the public Internet,
(5) Ganglia node health monitoring messages,
(6) Slurm messages communicating between the compute node daemons and the master Slurm controller on the SMS,
(7) Mail traffic notifying users of completed jobs and other noteworthy events, and
(8) Infrequent *Network Time Protocol (NTP)* updates.

While Warewulf file updates occur every five minutes, they are only necessary if files have actually changed in the Warewulf file database on the SMS. Because changing of master files is done by, and at the discretion of, a cluster administrator, we placed this service into the SMS-mgmt group along with the provisioning service. Our goal was to create a new SMS-mgmt node executing the services identified as SMS-mgmt services. The remaining services were identified as candidates for being decoupled and placed into a new SMS-services node containing time-critical services.

It should be noted that while trying to decouple the SMS services into two new nodes, we decided to replace Ganglia with a custom monitoring framework that writes to syslog. Therefore, we were able to leverage the rsyslog service for monitoring and eliminate the Ganglia traffic. We were also able to eliminate NFS traffic by moving the files being accessed via Warewulf hybridized paths (man pages and locales – which we compressed to reduce storage space) into the VNFS image that is loaded onto each of the compute nodes.

A summary of OpenHPC services that were partitioned into the SMS-mgmt and SMS-services groups is provided in Table 1.

SMS-mgmt	SMS-services
Provisioning	Rsyslog
File synchronization	NFS
	Routing
	Ganglia/monitoring
	Slurm
	Mail
	NTP

Table 1: OpenHPC services as partitioned into groups

4.1 The SMS-services Node

After logically partitioning services into the SMS-services group, we needed to implement our design by migrating time-critical services off the SMS and onto different infrastructure. An obvious strategy was to build a second VM which would run alongside the SMS VM but contain only the services belonging to SMS-services. However, we wanted the SMS-services infrastructure to be reliable, secure, quick to start, quick to snapshot, and quick to restore – in other words, it needed to be lightweight. Furthermore, due to VirtualBox lacking native InfiniBand drivers, we needed to mount our GPFS via an NFS gateway on our SMS, which we found to be slow. After evaluating our options and considering past challenges, we decided to build SMS-services as a Singularity [10] container.

Singularity makes use of native features of the Linux kernel such as control groups and namespaces, allowing us to take full advantage of the native performance of the underlying hardware on which a Singularity container is running. Furthermore, a Singularity container is encapsulated in a single, read-only file that can be cryptographically signed and verified – as opposed to other container platforms such as Docker [3] – providing assurances that the SMS-services node is authentic. As one of the newer technologies growing in popularity among HPC centers, and with active development and support for orchestration platforms such as Kubernetes and integration with many other utilities, Singularity was an obvious choice for us.

While cloning a VM requires copying all of its virtual disks and hardware device settings, cloning a Singularity container is as simple as copying the container file. Because the container is read-only (unlike a VM), any read-write data must be stored external to the container and mounted in at runtime. Consequently, the data can be backed up separately at any time. Furthermore, it is apparent which files have been modified during the runtime of the container since they must be mounted in externally.

We based our SMS-services Singularity container on CentOS 7, which is the operating system our SMS runs. We began by identifying which of the SMS-services services could be installed via yum packages found in existing repositories. Our center maintains a local mirror of the OpenHPC repositories, and we included a reference to this local mirror in our container definition file which allowed us to install the `ohpc-slurm-server` package provided by OpenHPC. In this way, we could be assured that our Slurm controller was fully compatible with the Slurm clients running on our compute nodes. We found that the OpenHPC Slurm packages also depended on an OpenHPC build of Munge that differed from the Munge provided by the vanilla CentOS repositories. Thus, pulling Slurm from the OpenHPC repositories circumvented several issues.

As mentioned, our SMS-services container runs in a read-only mode. However, some of our services need to write files. For example, Slurm writes a lot of state files, NTP writes drift files, and many services write log files. Thus, we needed to provide a writable location for this and other data. The first option we considered was using bind mounts. Upon starting the container, an administrator can specify one or more locations outside the container which are bound inside the container at specified paths. Thus, when the container attempts to write to those bound locations, it is in reality, writing to a location outside of the container. This approach works well in many situations. However, we ran into problems with single-file bind mounts and Warewulf.

The SMS-services container needs up-to-date copies of several files stored on the SMS such as the *passwd*, *group*, and *shadow* files and a copy of the Slurm configuration files. Because our SMS-mgmt node must maintain a master copy of these files anyway, it made sense for us to synchronize these files from our SMS-mgmt node to the container. In order to achieve this, we setup the container as a Warewulf client for the sole purpose of synchronizing files. However, under the bind mount strategy, we had to bind mount some single files like `/etc/passwd` into the container in order for them to be writable. We couldn't bind mount the entire `/etc` directory since that would eclipse important configuration files in the container. However, Warewulf file synchronization works by creating a temporary file with a dynamic filename extension and then moving the temporary file over the destination file. This moving operation would result in a new inode being created which, due to the complexities of the Linux kernel, would unlink the bound file from outside the system. Consequently, we needed to find a different strategy for our writable files.

Using Singularity's native support for Linux overlays [12], we created a zeroed out 100 GB file and formatted it as an ext3 filesystem. Then, we mounted this file as an overlay after starting the container. This allowed the container to remain read-only, but changes could be made to files which were then stored in the overlay. In this sense, the overlay stored the "difference" of all changes since the container was created. By using overlays, we circumvented all of our existing problems with writable files. Furthermore, we developed a backup strategy whereby we could make periodic copies of the overlay that stored the written files such as the critical Slurm state files and MySQL accounting database.

In order to implement networking for the container, we utilized Singularity's support for the *Container Network Interface (CNI)*, a container networking standard developed in part by Docker. We utilized the CNI bridge [1] plugin to connect the container to two existing Linux software bridges on the underlying host which allowed the container to effectively join our provisioning and public networks as though it was a physical machine [11]. Because one of our services was routing and *Network Address Translation (NAT)*, we needed to make sure the container could route and masquerade traffic. Thus, we utilized iptables and sysctl rules to allow the container to serve this function. A pleasant consequence of our bridging was that we could migrate the container between physical nodes quickly and without affecting anything on the running cluster. Upon starting, the container, picks up static IP addresses on our provisioning and public networks allowing it to move around freely.

While implementing the container, we discovered we needed several auxiliary services in order to support some of our critical services. For example, iptables was required to support routing and NAT, and cronie was required to support synchronizing Warewulf files and rotating log files. Also, we placed a Splunk forwarder inside the container to forward `/var/log/messages` to our Splunk server. Through rsyslog, the contents of `/var/log/messages` is every syslog message of every node on our cluster, so running this service inside the container made sense.

See Figure 2 for a pictorial view of the overall architecture of our infrastructure.

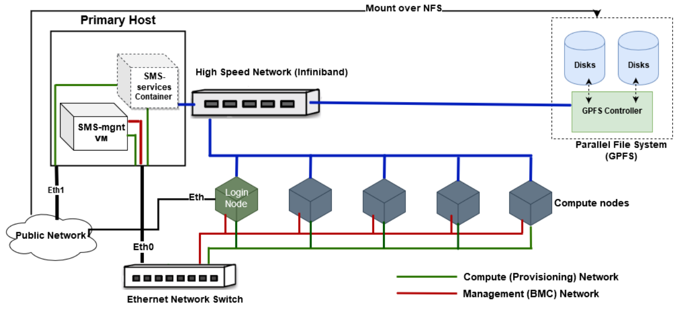

Figure 2: Decoupled LCC architecture

4.2 The SMS-mgmt Node

After partitioning services into the SMS-services group and implementing a container to realize those services, the functionality of the SMS-mgmt group became clear. The SMS-mgmt group was simply all services remaining on our SMS VM. In that sense, the SMS-mgmt node is like "the old SMS". It contains all the provisioning services and file system state information of "the old SMS". It is where VNFS images, applications, and user/group information resides, and is the part of "the old SMS" that requires the most work, and longest downtime, to update and configure during a system upgrade.

From a service perspective, Warewulf is the most critical service on the SMS-mgmt node. The Warewulf database stores all information about how to provision the compute nodes and configure their networks. Furthermore, the SMS-mgmt node contains the master copies of all files, which are synchronized to compute nodes throughout the cluster and to the SMS-services node.

After decoupling the SMS-mgmt services, we found that we could power off the SMS-mgmt node almost indefinitely at will. As long as no files needed to be changed and synchronized between nodes, and as long as no nodes required to be re-provisioned, the SMS-mgmt node did not need to be online. This gave us extreme flexibility in taking down SMS-mgmt, moving it between servers if necessary, and upgrading it slowly and carefully, testing each step of the upgrade, with essentially no impact on the cluster overall.

5 A PRODUCTION DEPLOYMENT

The University of Kentucky's *Center for Computational Sciences (CCS)* is the primary center for high-performance computing research across the state of Kentucky. CCS operates LCC, a heterogeneous compute cluster consisting of nearly 10,000 compute cores and 100 graphical processing units with a combined theoretical peak performance of 950 teraflops. The cluster supports approximately 100 faculty researchers and 600 users by providing about 80 million core-hours per year for their research.

LCC consists of six login nodes, two administrative infrastructure nodes, one high-speed data transfer node, and 450 heterogeneous compute nodes, all attached to a shared 1.3 PB high-speed parallel filesystem (GPFS). The compute nodes vary from 16 cores to 48 cores each with memory ranging from 64 GB to 3 TB. For low latency, the compute nodes are interconnected with an InfiniBand network, with speeds ranging from 56 Gbps to 100 Gbps. In addition to the high-speed InfiniBand network, LCC is also attached to a provisioning network, a management network, and the public Internet.

Of the two administrative infrastructure nodes, one serves as the primary which runs both the SMS-services node and the SMS-mgmt node, while the other serves as a backup for these services. We have successfully decoupled OpenHPC's SMS services and are running both the SMS-services node and the SMS-mgmt node on our production cluster. In order to deploy the decoupled services, we needed a scheduled (and hopefully our last) complete cluster downtime.

When we first began implementing our service decoupling approach, we needed a way to test that our idea would work – i.e., we needed to know it would work before scheduling the downtime required to implement the decoupling fully on our production cluster. To do our testing, we prototyped the SMS-services container on our secondary administrative infrastructure node, which was not being used for anything else at the time. In order to test the decoupled services on real compute nodes, we drained a select group of compute nodes from our cluster and re-provisioned them with a development VNFS and new network settings that would have them only talk to our new SMS-services container for the decoupled services. To verify that the decoupling had worked successfully, we submitted some example compute jobs from the SMS-services container. At this point, SMS-services was running its own Slurm controller separate from the SMS-mgmt Slurm controller. We used different ports to ensure that the two services would not overlap. We also ran tcpdump on the network and verified that the drained development compute nodes were indeed no longer communicating with the SMS-mgmt node after booting. Having convinced ourselves that the decoupled approach worked reliably, we scheduled the downtime and deployed the SMS-mgmt and SMS-services nodes on our production cluster.

A couple of weeks after successfully deploying our decoupling idea in production, the secondary disk on our primary infrastructure server began having RAID errors. The SMS-mgmt VM was running on the affected secondary disk, but the SMS-services VM was running on the unaffected primary disk. During that time, we lost the SMS-mgmt, but the SMS-services continued running successfully. As a result, the production cluster was not affected. Although no administrator likes to learn about hardware failures – and this failure would certainly have taken down our old SMS node (and the entire cluster) had it been running on the failed disk – we were delighted to find out that our decoupling had indeed worked and saved us from a real disaster scenario.

6 DISCUSSION & FUTURE WORK

Our idea of decoupling critical services from the SMS led to a successful implementation on a large, production cluster. Our work has enabled us the ability to pick up and move both our SMS-services container and SMS-mgmt VM between physical nodes of the underlying infrastructure without affecting the running cluster. We have tested this in production and found it to be highly successful and reliable.

After a successful deployment, we have put some thought into what we learned from the overall experience and how we could have done some things differently. For example, some of the services require state information and need access to a writable file system. Examples include the Slurm controller and NTP. However, other services such as the routing and NAT service do not need to write any state information. We could perhaps further decouple these stateless services into their own containers, which may improve security and reliability.

Taking the routing and NAT service as an example, currently, we have a bottleneck whereby the single container which runs this service is bound to the *Network Interface Card (NIC)* of the underlying physical server. There is a possibility of an accidental or intentional denial of service attack if too many compute nodes are attempting to route traffic outside of the cluster. Furthermore, this same NIC is used for provisioning nodes, so provisioning too many nodes at once could also introduce a denial of service. While our login and data transfer nodes have dedicated public Internet NICs, all compute nodes are single-homed and must route through SMS-services if access to the public Internet is needed. We could potentially break this routing service into its own container, or multiple containers, and distribute instances across underlying physical servers to create a distributed load balancing solution.

Our approach can theoretically be expanded upon to break out individual services into their own containers which can potentially be treated as microservices and orchestrated through platforms like Kubernetes. This could lead to better resiliency, especially in the case of services which can be operated without the state. However, as mentioned before, implementing true high availability that really works can be challenging, and we would need to architect a solution we were confident in before spending the time implementing such a design.

Furthermore, the focus of our work was to break out select time-sensitive services, but we could potentially containerize even the SMS-mgmt services, although it is not yet clear to us that doing so will be beneficial. In hindsight, VirtualBox may not have been the best virtualization choice for our SMS and subsequent SMS-mgmt node. However, part of our architectural decisions include the technical considerations of our staff, and many of our staff are comfortable with VirtualBox whereas few are comfortable with other, perhaps more suited, virtualization platforms such as KVM and Xen.

Another possible area of improvement is the overlay file system that the SMS-services container mounts at runtime. A corrupt overlay would result in a significant loss of data, particularly with regards to Slurm scheduler information. Therefore, the overlay must be backed up regularly. However, if the underlying file system that the overlay itself is stored on is faulty, this will also affect the cluster. Currently, we store our overlay on our GPFS. However, we have thought about using loopback devices to create a virtual RAID 1 array across multiple file systems and storing our overlay on that virtual RAID 1 array. In that case, one of the underlying file systems going down would not cause the overlay to be lost.

7 CONCLUSIONS

HPC cluster-management nodes, such as the OpenHPC SMS node, play a critical role in the availability and reliability of an HPC cluster. Taking down the SMS node for any reason, including for much needed upgrades, is often avoided by cluster administrators because it leads to significant cluster downtime and loss of productivity.

In order to circumvent this, we decoupled critical services that must be active at all times from other services that could potentially go down for long periods of time without affecting the overall operation of the cluster. The decoupled services were moved into a portable Singularity container which gave us assurances of reliability and security, and the remaining services were separated and placed in a VM that could be taken down at will.

The implementation of our approach and deployment on a university production cluster with hundreds of users has been highly successful. We have been running our decoupled services for two months now and have tested multiple disaster scenarios. We have been able to migrate both the SMS-services container and the SMS-mgmt VM between underlying physical infrastructure on the live, production cluster without affecting running jobs and without any users knowing the migration occurred. Furthermore, because we are now able to take the SMS-mgmt offline essentially at will, we can take full snapshots of the SMS-mgmt VM more frequently. Before decoupling services, we could only take full snapshots during a scheduled maintenance window with downtime. In the past that occurred so infrequently that we would be losing months of data if the SMS VM had become corrupted. Our new solution allows us to roll back to a relatively recent snapshot with far less loss of data.

We are in the process of fully documenting our approach for the OpenHPC development community in hopes that it will be adopted by them and become a suggested and supported way of installing OpenHPC. Our approach would allow other OpenHPC clusters to reap the same benefits we have observed. In the meantime, we can distribute internal documentation upon request to our HPC colleagues, instructing them how to retrofit their current OpenHPC systems to support decoupled services.

Lastly, although we have not tested our approach on other cluster management systems, most of these systems operate the same, or a similar, set of time-critical services which, if decoupled, would lead to the same benefits. Therefore, our solution can be extrapolated to service the broader needs of the HPC community.

REFERENCES

[1] CNI Bridge. 2019. CNI Bridge Plugin Documentation. https://github.com/containernetworking/plugins/tree/master/plugins/main/bridge

[2] Bright. 2019. Bright Computing Bright Cluster Manager Web site. https://www.brightcomputing.com/product-offerings/bright-cluster-manager-for-hpc

[3] Docker. 2019. Docker Web site. https://www.docker.com/

[4] GPFS. 2019. IBM General Parallel File System. https://www.ibm.com/support/knowledgecenter/en/SSFKCN/gpfs_welcome.html

[5] Intel. 2019. Intel HPC Platform Software. https://www.intel.com/content/www/us/en/high-performance-computing/hpc-orchestrator-overview.html

[6] Lustre. 2019. Lustre parallel file system. http://lustre.org/

[7] OpenHPC. 2019. OpenHPC Web site. http://openhpc.community/

[8] Oracle. 2019. VirtualBox Web site. https://www.virtualbox.org/

[9] P. M. Papadopoulos, M. J. Katz, and G. Bruno. 2001. NPACI: rocks: tools and techniques for easily deploying manageable Linux clusters. In *Proceedings 2001 IEEE International Conference on Cluster Computing*. 258–267. https://doi.org/10.1109/CLUSTR.2001.959986

[10] Singularity. 2019. Singularity v3.3 Documentation. https://sylabs.io/guides/3.3/user-guide/

[11] Sylabs. 2019. Network Virtualization in Singularity. https://sylabs.io/guides/3.3/user-guide/networking.html

[12] Sylabs. 2019. Working with Persistent Overlays in Singularity. https://sylabs.io/guides/3.3/user-guide/persistent_overlays.html

[13] Warewulf. 2019. Warewulf Web site. http://warewulf.lbl.gov/

[14] Wikipedia. 2019. Preboot Execution Environment (PXE). https://en.wikipedia.org/wiki/Preboot_Execution_Environment

[15] xCAT. 2019. xCAT Documentation. https://xcat-docs.readthedocs.io/en/stable/