Next-generation cluster management architecture and software

Paul Peltz Jr., J. Lowell Wofford
High Performance Computing Division
Los Alamos National Laboratory
Los Alamos, NM, USA
{peltz,lowell}@lanl.gov

***Abstract*—Over the last six decades, Los Alamos National Laboratory (LANL) has acquired, accepted, and integrated over 100 new HPC systems, from MANIAC in 1952 to Trinity in 2017. These systems range from small clusters to large supercomputers. The high performance computing (HPC) system architecture has progressively changed over this time as well; from single system images to complex, interdependent service infrastructures within a large HPC system. The authors are proposing a redesign of the current HPC system architecture to help reduce downtime and provide a more resilient architectural design.**

***Index Terms*—Systems Integration; Systems Architecture; Cluster**

I. Introduction

High performance computing (HPC) systems have always been a challenge to build, boot, and maintain. This has only been compounded by the needs of the scientific community upon the HPC architectural design because FLOP-centric applications are no longer the driving force behind modern HPC procurement. The modern scientist's complex applications have necessitated the need for burst buffers, network gateways (visualization and real-time instrument data), I/O forwarders, and file system caching support services which have in turn greatly increased the complexity of the HPC systems. Vendors are required to provide a flexible HPC architecture for a diverse customer base to integrate into their infrastructure. However, this flexibility comes at a cost. The variety of support services that are needed now have led to a more complex HPC architecture. Many of these new support services are interdependent with other services and these services have to start up in the right order. This complexity has greatly increased the time it takes to boot the HPC system. Previous generations of HPC systems would boot up in a matter of minutes, where the current generation of HPC systems take approximately two hours to boot. The long and unpredictable nature of booting large-scale HPC systems has created an expensive problem to HPC institutions. The downtime and administrator time required to boot has led to longer maintenance windows and longer time before these systems can be returned to users. A fresh look at HPC system architecture is required in order to address these problems.

II. Current HPC System Architecture

The current HPC system design is no longer viable to the large-scale HPC institutions. The system state is binary, because it is either up or down and making changes to the system requires a full system reboot. This is due to a number of reasons including tightly coupled interconnects, undefined API, and a lack of workload manager and system state integration. Overcoming these problems require a redesign of the current HPC architecture. Typical HPC systems are designed as shown in Figure 1.

The problem with this design is that even though it is designed to provide redundant management services, only one of these management systems are active at a given time. Active-Passive failover adds additional risk and burden to the management of the system, which normally outweighs its benefits of preventing single-point failures. Only having one active management node also places the entire burden of managing, booting, and supporting the HPC system onto one server. In order to scale to larger node counts, secondary boot servers (sub-masters) are necessary in order to provide a scalable boot solution; otherwise there is too much congestion from one server and will cause the boot to fail. This can also be addressed by means of a tiered boot, in which the system is incrementally booted in units in order to not overload the management server. All of the aforementioned support service types (such as burst buffers) only add to the complexity of booting the system.

III. Future HPC System Architecture

In order to address the problems detailed in the previous section, we need to take a fresh look at HPC system design. By "fresh look" we do not mean a reinvention of the wheel, however. We can leverage much of what has been done with previous generations of HPC systems and also what the

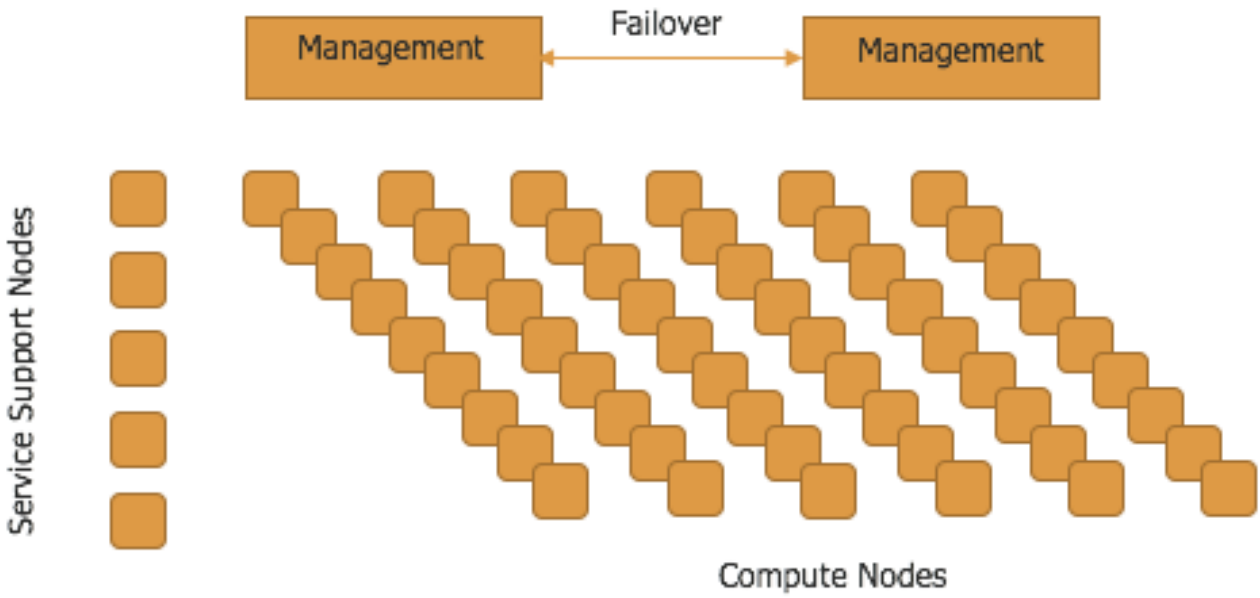

Fig. 1. Current HPC Design.

cloud management community are doing. One of the things that cloud management systems do well is scale horizontally by providing load-balanced communication paths from the underlying node farm to the management servers. This type of design removes the single management server bottleneck and provides scalability to handle ever-growing system sizes. LANL's Roadrunner [7] system was able to boot in under 20 minutes, and used industry standard tools xCAT [19] and Warewulf [18] to accomplish this. Its success can be directly attributed to the division of duties of service nodes from compute nodes. Service nodes were managed separately from compute nodes, but from the same management system. Speed increases in compute node boot times were achieved by close collaboration between IBM and LANL to develop a multicast boot system which treated all compute nodes as boot servers to help distribute the images across the system. This greatly improved the speed and reliability of system boot. Building from the lessons learned in these two examples, we propose the following design to meet the future needs of large HPC systems.

This design alleviates many of the shortcomings of the current system and provides a scalable framework to expand upon. Starting from the top of Figure 2, Active-Passive HA is replaced with a management cluster that handle system management functions. This management cluster will provide distributed services to the rest of the system and to systems administrators; this may be sized as needed according to the scale and criticality of the HPC system under management. All system functions should be provided through both CLI and GUI interfaces that interact through an ABI with the system software. This provides conflict protection; for example, against multiple administrators executing the same commands on different management nodes. It should be possible to upgrade individual management nodes and provide backwards compatibility for minor upgrades of the system software. Although it is understood that major upgrades have a more drastic impact upon the system and will necessitate the rest of the system being shut down, these should happen infrequently, no more than two times a year.

A. Scalable Service Nodes

The scalable services nodes (SSN) will be the major workhorses for the management cluster, providing such services as boot support, log aggregation, metrics collection, workload scheduling and job launch, etc. These SSNs should be backed by an object storage system such as CEPH [3] to provide a scalable storage system that can scale in capacity/bandwidth with the number of SSNs required to support the system. SSNs would provide distributed services with backward-compatible ABI interfaces so that minor upgrades can be performed via rolling upgrades of each SSN, again with major and more disruptive upgrades requiring a shutdown of the management cluster. Such major upgrades should be limited to no more than six times a year. It may also be beneficial for these nodes to be stateful and not rely on any other service to maintain state. A configuration management system should be employed to maintain the state of the SSNs to ensure proper configuration.

B. Service Nodes

Service nodes that provide near-compute resources to the HPC system, such as I/O forwarders, will not necessarily be supported by this distributed SSN model. However, they should reroute services to other available service nodes that provide the same functionality. These service nodes would likely require more frequent patches and configuration changes so rebooting of these may be more frequent. Again, they

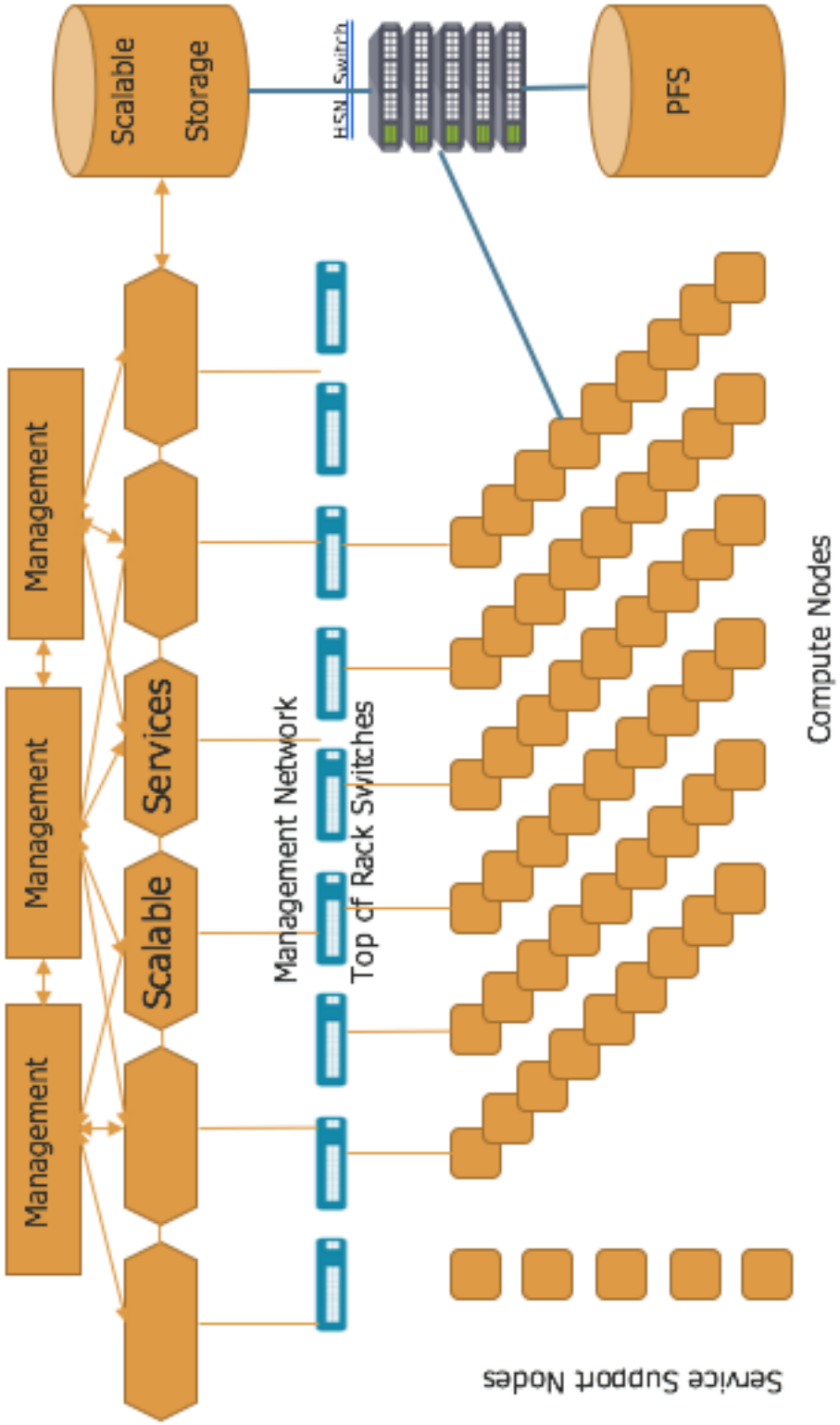

Fig. 2. Future HPC Design.

should utilize rolling updates with backward-compatible ABIs so that changes do not require a reboot of the entire system.

C. Compute Nodes

Compute nodes are the most volatile components of the system, and therefore the priority on these components should be to return them to service as quickly as possible with no disruption to the running system. Compute nodes need to be rebooted for a variety of reasons such as job failure, OS image changes, and BIOS changes.

D. Management Network

An upgrade of the management network is also desirable for future HPC systems. The management network has been a long neglected technology that has traditionally been under-provisioned and therefore underutilized. With an upgrade of the core management network with smart switches, it would allow the site and vendor to offload certain I/O traffic off of the HSN and onto the management network. This would include but not limited to: pub/sub services for logging, metric collection for performance data, scheduler communication, and booting of nodes while the system is running jobs to prevent disrupting the HSN. In order to insure that the site does not overload the management network and prevent critical data communication over the management network, appropriate QoS settings should be placed on the network.

IV. Boot Optimization Strategies

This design provides a separation of services and eliminates the binary nature of the HPC system state. This allows compute nodes to be able to be rebooted freely for any purpose that may be required by users or administrators. Because of this requirement it is paramount that these nodes can be rebooted quickly, and that should occur whether rebooting one or hundreds of thousands of nodes. In addition to being quick, boots must also be reliable and tolerant of failure. Accomplishing this task will require research, both of past successful strategies and innovative ones.

A. Local Storage

The strategy that provides the greatest scalability is to simply boot a compute node from local storage. Previously this meant having a large hard disk drive local to the system; these were frequently slow, used too much power, generated heat, and consumed valuable space within the chassis. With the introduction of the M.2 [10] form factor, all of these concerns are substantially diminished. Such local storage could be used in two different ways. The first would be to boot the entire Operating System(OS) directly from the local storage device. However, in this scenario configuration drift between nodes becomes a problem. The storing of the OS on the local device would also hamper the ability to boot multiple OS images because the device has limited space. The second, preferred option would be to use the local storage to hold a stage-one [9] boot image, which gets the node up enough for it to the pivot into its final OS image. Distributing this initial stage-one image to all nodes may be challenging as the scale of the system increases. Possible solutions are to use the Redfish [16] API over the management network to seed the M.2 device initially and also to verify its presence when the compute node is booted after being serviced. If using the Redfish API is not possible then distribution of the stage-one image could fall to the top-of-rack switches that control the management network, which leads to the next option.

B. Top-of-Rack Switches

Top-of-rack (TOR) Ethernet switches are now capable of doing much more than just handling network traffic. They are now mini-servers capable of running a customized Linux environment. TOR switches also represent a scalable system unit, as they tend to scale with the number of racks in an HPC system. With switches now providing advanced functionality and being a scalable unit design, we should use them to help boot the system. Using a project such as ONIE [12] as an example, we can hijack the DHCP and PXE boot process on the switch and deliver a small stage-one image to the compute nodes to boot from. Using the TOR switch would also provide a solution for smaller HPC systems that may not need the local storage or a motherboard that does not have a M.2 slot, therefore both strategies should be investigated.

C. Multicast

Another option that has been successful for LANL in the past [17] was to use multicast TFTP to distribute the work of booting the compute nodes across the system. Each compute node would be activated as a TFTP server for a short time and then continue to boot. This method helps distribute the work required to distribute the stage-two image to every compute node in the system.

D. kexec

A more innovative approach may be to use kexec [8] to reboot a compute node into a new OS without the node having to reload from the BIOS on up. Booting from the BIOS up adds a large amount of additional time that is not necessary under most circumstances. If there are no BIOS changes required for the reboot, there is no reason why the node should have to incur the additional time to reload the BIOS. This would only be effective once the system is booted, but it could be a very fast method to switch between running OS environments on compute nodes.

E. Torrent

Using the torrent protocol to do peer-to-peer distribution of the boot images is another method that has been attempted in the past by ROCKS [2] in their Avalanche installer. This could provide the ability of all compute nodes to help deliver the stage-two images to each of its peers for a short period of time and then continue booting. This is similar to the multicast method mentioned previously.

F. Coreboot

A research project that was originally developed at LANL and now being pursued out of the Google realm is Coreboot [4]. This replaces the existing BIOS with a lightweight, secure, and highly optimized Linux payload to drastically increase the boot times of individual nodes. This alone has shown that boot times can be down to as little as 3 seconds. Coreboot also has support for Ethernet, Myricom, and Quadrics interconnects for netbooting. This solution has great promise for speeding up boot times, but processor vendors would need to support this project to make it a viable solution.

V. A Review of Current Solutions

Now that we have an architectural design for the system, do any of the current HPC system management solutions meet our requirements? We will review a few of the major offerings that are available, which are both commercial and open source solutions. This is not a comprehensive list of solutions, as many large organizations have also developed their own software to meet their own site's unique requirements. LANL is also using a customized version of Perceus [15] which is a defunct branch of warewulf and Sandia National Laboratory uses oneSIS [11] for example.

A. Warewulf

Warewulf is an open source cluster management toolkit originally developed at Lawrence Berkely National Laboratory. It has proven very successful in the past for managing small to mid-sized HPC systems, but it has been difficult to adapt to the large system space due in part to it's lack of focus on scalable management infrastructure. For instance, warewulf has only marginal support for distributed services nodes and the current main branch only supports x86 architecture, though there is a development effort to incorporate ARM as well.

Warewulf is written in object-oriented Perl, and its management data store can be backed by multiple database services through a database abstraction layer. The data model is extensible, allowing for feature additions and modification of basic system functionality. The core functionality of warewulf is limited to keeping an inventory of nodes, managing filesystem images, and implementing a PXE boot process. It provides boot provisioning by auto-configuring basic services such as DHCPD, BIND and in.tftpd to provide the relevant portions of the boot process. Images are provided to nodes through an Apache web service that pulls them from the backing database.

While warewulf was designed to be modular and extensible, it's underlying design was not geared towards some of the current challenges such as multi-architecture clusters and alternative boot methods. Incorporating these features would require software architecture changes that extend beyond the extensibility of the platform. Additionally, there has been little movement in the open source community to continue to develop and adapt warewulf to newer HPC system challenges.

B. OpenStack

OpenStack has gained a lot of popularity in the hyperscale world for managing both private and public cloud systems. OpenStack does extremely well at handling elastic demand of resources to quickly respond to load requirements for companies to manage their infrastructure. However, these are loosely coupled systems tend to scale well horizontally, but not as a tightly interconnected HPC system. There have been some forays into crossing over HPC into OpenStack [6]. Most of these systems are focused on diverse scientific workload to address a multitude of use cases. HPC system sizes that have deployed OpenStack are not typically large either and until ironic fully supports diskless booting, the compute nodes do require to be provisioned by Ironic which would greatly increase the boot time of compute nodes. The complexity and rapid development pace of OpenStack and its ever growing number of components is also a concern. However, only a subset of the OpenStack components could be used to reduce that complexity.

C. xCAT

xCAT is an open source cluster management suite that has a somewhat active development community and is backed by IBM. Compared to warewulf, it aims to be much more feature complete, integrating control of switches, PDUs and BMCs, and providing support for x86 and PowerPC. It also has native support for multiple modes of deploying service nodes for large scale HPC systems.

The degree to which it is endorsed by IBM has proven to be both a benefit and a drawback. Major vendor support means that development on xCAT has continued to advance, and has incorporated many advanced features. On the other hand, the association with IBM tends to push other vendors away who want to avoid adding value to a competitor. As a result, many of the more advanced features of xCAT are specific to systems running IBM hardware. Additionally, there is a push in xCAT development to emphasize technologies that advance IBM, such as PowerPC architecture.

xCAT has a large code base, and has not been designed for easy extensibility in the way that warewulf and other platforms have been. As a result, adding new major boot methodologies, image management features and hardware platforms requires a good deal of direct modification to the code. This makes modifying xCAT with new major features a daunting task. xCAT has been performing scalability tests to reach exascale systems, but to date these tests have only been performed with IBM hardware.

D. Cray Linux Environment Software

Cray Linux Environment (CLE) [5] system management software has historically been specially designed to support its unique hardware platform. Because of this fact, it is debatable if the software should be mentioned in this paper. However, they are one of the few companies that can demonstrate scalability of their software design on large capability class HPC systems. Cray's release of CLE 6.0 was intended to

conform to industry standards and open source software solutions to manage their HPC system. In some ways this was a successful venture because of the use of RPMs, YAML files to define configuration options, Ansible [1] for configuration management, and chroot based images for different node types in the system. Unfortunately, each of these solutions were implemented in a non-standard way or obfuscated by custom tools to manage these various tools. The complexity of Cray's implementation of these tools have caused a steep learning curve for HPC administrators which creates a challenge for training new employees into organizations that operate Cray machines.

There are two important lessons learned from Cray's CLE 6.0 however that are important to capture here. The first is the concept of post- vs. pre- configuration of the node state on the system. Cray decided to try and implement a purely post-configured node state as defined by Ansible after the node boots. The node in the system boots a completely generic node image and then intends to specialize the node post-boot to provide a specific service such as an LNET router. There were many challenges to this approach that were discovered including: failed boot due to broken Ansible plays, slow performance of Ansible on KNL nodes, and Ansible play bloat which increases boot time. The lesson learned here is that disk space is cheap, but time is not. The disk space required to host node specific images greatly outweighs the added time to do post-configuration of a node. Cray's monolithic boot procedure compounds the other issues and adds to a fragile and long boot process if there is a system interrupt event.

Node image management and state is a complex problem and Cray developed two robust tools to address this, the Image Management and Provision System (IMPS) and Node Image Mapping Service (NIMS). These two tools do an excellent job at managing the images by defining them through recipes and by mapping those images and configuration definitions to individual nodes. Unfortunately, these tools are unique to the Cray, but their philosophy and design are important to consider for the future.

VI. Future Architectural Solutions

As HPC systems move into exascale and beyond the management and service node infrastructure required to support the system could be large enough to be considered their own cluster. Therefore they should be treated as such instead of trying to manage both the management system and the compute system under the same software stack. Because of this, it is necessary to also rethink the system management software as well as the hardware architecture. The aforementioned software is either targeted for small to medium clusters, custom hardware, or hyperscale systems. The future of HPC systems management could and probably should be a hybrid of these management systems. The proposed future HPC system architecture could be divided into three separate management zones.

A. Management Cluster

Traditional HPC management systems are monolithic and difficult to install and maintain. All of the components that are required are loosely connected and dependencies are not well defined. If instead these components were separated out and managed independently as micro-services it would allow the entire management stack to run on a basic OS and all of the components necessary to administer the system would be deployed as containers for virtual hosts. Two approaches to this could be OpenStack or a micro-service cluster solution such as OpenShift [14] which is a Docker and Kubernetes deployment engine. This methodology could enable some of the requirements listed above which would allow for rolling reboots of management services when there are updates.

A scalable design also needs to take into account for smaller machines that do not want to incur a large cost in system management overhead. Uptime may not be a priority for some institutions, so this should be taken into consideration in the design requirements. A 3+ node management cluster does not necessarily need to be running on physical hardware. If the clustering of the services were also run in containers or virtualized then the three node cluster could all be running on one physical node. Even the scalable service nodes could be virtualized on the same physical hardware. OpenStack however does not scale down as well which may make it more costly at a small scale.

B. Service Support Nodes

The services these nodes provide tend to be more closely tied to running jobs on the system. They provide support services such as I/O forwarding, network routing, burst buffers, etc. Since these services are typically tied to physical hardware such as network interconnects and SSDs that the services these nodes provide can be shared by multiple jobs running on the system, it might not lend itself well to being treated as a micro-service. The services these nodes provide also tend to require the most updates and changes to them. Therefore, these nodes need to be drained and once idle reboot them into a new configuration or software version and then returned to service.

C. Compute Nodes

Compute nodes are extremely volatile and have the highest failure rates of any component of the system. Therefore, expectations are that they will be rebooted the most often. Other than post-boot configuration of the compute node to prepare it for service, the next item that continues to cause long delays in boot time is the mounting of various file systems either within the HPC system or from external file systems outside of the HPC system. Depending on the number of clients attempting to mount and the number of file systems there are to mount, it could cause delays in the order of 1-5 minutes of a compute node to mount everything that is required for it to be healthy. Research needs to be done to either reduce the number of mount requests or remove the file system mount requests completely by the use of I/O forwarding layers.

D. System Management Software

In order to scale up to 10s of thousands of nodes and support varying hardware vendors and CPU architectures, the solution must support and run under different CPU architectures, e.g. arm, ppc, x86_64, and be modular in design. The modular design component is important so that it can interface into proprietary boot systems that vendors may provide. If this is the case then the vendors need to provide an API into their boot system and a module can be built to work with the vendor's API. This modular design will also be useful for heterogeneous system designs in which different boot optimization techniques would be required. Separate boot implementation may also be useful for in-production reboots vs. system quiesced reboots of compute nodes. While the system is running jobs it would be desirable to use the management network rather than the high-speed network to boot the node to keep from disrupting running jobs.

VII. Future Software Management Solution

In order to address the needs of the future systems and replace legacy cluster management solutions, LANL has designed a new cluster management tool named Kraken [20]. Kraken addresses the needs above as a modular and cluster management solution capable of supporting multiple architectures. Kraken is written in Go and supports running on any architecture for which the Go language has a compiler.

Kraken takes a novel approach to cluster management. Rather than acting as a front-end to basic cluster booting services, as traditional cluster management solutions have done, Kraken provides its own scalable microservices for basic tasks like system PXE boot. This approach allows Kraken to easily scale out boot services on demand.

Kraken provides a framework for distributed system automation. At its core, Kraken is a highly flexible distributed state engine. It maintains the latest intended state of the cluster, tracks the current state of the cluster, and is capable of determining how to converge current state into the intended state. It achieves asynchronous state convergence through an eventual consistency model that has proven its ability to scale in large environments [21], [22]. The state engine approach provides a high level of flexibility and modularity while also providing new ways to handle difficult automation tasks on clusters.

Any module introduced into the Kraken system can do any number of the following tasks: 1) create new system state variables to track; 2) declare its ability to discover the current value for a system state variable; and, 3) declare its ability to "mutate" a state variable from one value to another, e.g. the ability to mutate the system power from *OFF* to *ON*. This provides for a powerful and flexible mechanism to extend management capabilities. By creating a compact module API, we are able to rapidly develop, test and assess different boot and management approaches and make per–cluster determinations on the desired management methodologies.

In addition to providing a flexible module framework, the design of Kraken provides a centralized mechanism for achieving automation of cluster management tasks. By providing the ability to automatically mutate the system state, and further to define and extend what system state is comprised of, a wide array of system automation are easily attainable. For instance, an image management module can handle a rolling update process by providing the necessary state mutation steps required to safely transition the state of a node from running one image version to running another. Similarly, functionality can be added to handle automated recovery from failure states (where possible), use different methodologies during different phases of system life-cycle (e.g. cold vs warm boot), and provide different management capabilities to hardware of different architecture or purpose within the same cluster.

VIII. Conclusion

One of the big challenges in redesigning the HPC system is the balance between "reinventing the wheel" and using "the right tool for the right job". There are several options available for managing clusters, some are commercial solutions, some open source, or vendor specific solutions. Many of these solutions can manage and boot a system, but at the cost of complexity, scaling limitations, or solutions that are difficult to maintain. A common system management solution would be ideal, but reaching such an agreement among institutions and vendors on the solution has historically been difficult. However, there needs to be a movement in evolving the system management framework to make it more scalable and more resilient. OpenHPC [13] provides a promising start to this attempt, but at present it only contains xCAT and Warewfulf as system management solutions. LANL intends that this new development effort with Kraken can improve the current offerings within OpenHPC for cluster management. In order for this to be successful however a collaborative effort between institutions and vendors is critical to the success of this new ambitious project.

Acknowledgments

The authors would like to thank Quellyn Snead, Daryl Grunau, Cory Lueninghoener, Michael Jennings, Sean Blanchard, and Tim Randles for their input on this paper. Many of the topics and ideas came out of a number of meetings and brainstorming sessions that lead to the writing of this paper.

References

[1] Ansible, 2018. https://www.ansible.com.
[2] Base users guide, 2018. http://central6.rocksclusters.org/roll-documentation/base/6.1.1/roll-base-usersguide.pdf.
[3] Ceph, 2018. http://ceph.com/.
[4] Coreboot, 2018. https://www.coreboot.org/.
[5] Cray publications, 2018. https://pubs.cray.com/discover.
[6] Hpc openstack, 2018. https://www.openstack.org/assets/science/OpenStack-CloudandHPC6x9Booklet-v4-online.pdf.
[7] Ibm roadrunner, 2018. https://en.wikipedia.org/wiki/IBM_Roadrunner.
[8] Kexec, 2018. https://en.wikipedia.org/wiki/Kexec.
[9] Linux startup process, 2018. https://en.wikipedia.org/wiki/Linux_-startup_process.
[10] M.2, 2018. https://en.wikipedia.org/wiki/M.2.
[11] onesis, 2018. https://en.wikipedia.org/wiki/OneSIS.
[12] Onie, 2018.
[13] Openhpc, 2018. http://www.openhpc.community/.

[14] Openshift, 2018. https://www.openshift.com/.
[15] Perceus, 2018. https://github.com/perceus.
[16] Redfish api, 2018. http://redfish.dmtf.org/.
[17] Roadrunner: Hardware and software overview, 2018. http://www.redbooks.ibm.com/redpapers/pdfs/redp4477.pdf.
[18] Warewulf - scalable, modular, adaptable systems management, 2018. http://warewulf.lbl.gov/.
[19] xcat, 2018. https://xcat.org/.
[20] kraken, 2018. https://www.github.com/hpc/kraken/.
[21] W. Vogels, "Eventual Consistency," Communications of the ACM, vol. 52, no. 1, 2009.
[22] D. Terry, "Replicated Data Consistency Explained Through Baseball," MSR Technical Report, 2011.

APPENDIX

A. Abstract

"This paper is not paired with an artifact."