Enabling Docker for HPC

Jonathan Sparks

Compute Products R&D
Cray Inc.
Bloomington, MN, USA
e-mail: jsparks@cray.com

Abstract—**Docker is quickly becoming the de facto standard for containerization. Besides running on all the major Linux distributions, Docker is supported by each of the major cloud platform providers. The Docker ecosystem provides the capabilities necessary to build, manage, and execute containers on any platform. High-Performance Computer (HPC) systems present their own unique set of challenges to the standard deployment of Docker with respect to scale image storage, user security, and access to host-level resources. This paper presents a set of Docker API plugins and features to address the HPC concerns of scaling, security, resource access, and execution in an HPC Environment.**

Keywords-component; Docker, containers, HPC scaling, security

I. INTRODUCTION

System administrators are tasked with keeping system resources available and operational for the users. This includes maintaining a common operating system environment, libraries, and applications across the system. It also means that any modifications to the operating system have to be carefully planned to ensure that they do not impact availability and the integrity of the system. As a result, system administrators control the software, configuration, and deployment of services on the system. Users wishing to run applications that need a different software environment find themselves at odds with the traditional "one size fits all" system administration model used to define the environment. Some users require a more flexible environment where they can define the application runtime and configuration environment.

Traditional administration of HPC systems has been designed to support a set of applications that are distributed, compute-intensive, and often latency-sensitive. Weather modeling and crash simulation are two of the classic examples that still follow this paradigm. The standard approach to executing these codes is to use as much homogenous processing hardware as possible to increase the simulation resolution or shorten the computational time-to-solution. Being computation-bound, these applications are typically scheduled by the number of CPUs and the anticipated wall clock time required to complete the simulation.

Over the last few years, a new model of HPC computing has begun taking root. This new paradigm uses container technologies to execute applications using HPC resources. These applications may require access to different OS libraries, configurations, or services, and may leverage different storage types, such as databases, in order to perform computation against the data. Other applications may require access to remote network resources, making network bandwidth the constraining factor. None of these contemporary applications are well-served by traditional scheduling systems that depend on the up-front request of fixed CPU and wall clock time resources. Furthermore, the varying natures of these "secondary" resource requirements (e.g. long running services, dynamic resource use) leaves these jobs prone to being preempted or killed, due to oversubscription of system resources or unconstrained runtimes.

Lightweight virtualization technologies, such as Docker [1] and CoreOS rkt[2] are becoming the standard in Enterprise computing. Google has reported that all of its infrastructure services will use container technology [3], all of the major cloud providers including Azure Container Services (AKS) [4], Google Cloud Kubernetes Engine [5], and Amazon Elastic Container Services [6], provide a Container-as-a-Service (CaaS) offering. In addition, the major Linux distributions provide bundled container environments. Examples include Red Hat OpenShift [7] and Novell SUSE CaaS Platform [8].

In order to provide better support for this new class of applications, technologies like Docker have been developed. At its core, this technology leverages core Linux capabilities for application isolation. This includes the use of cgroups [9] for resource controls and kernel namespaces [10] for providing process isolation. These container ecosystems also provide methods of deploying and distributing the application along with associated libraries and configuration.

Docker is the most well-known container platform available today and has seen wide adoption among Enterprise vendors and system software developers. Although its primary purpose is the deployment of scalable microservices, Docker offers features that are compelling to the computational science community. The ability to create a self-contained application including all the dependent libraries and configuration, packaged in a portable format, has been revolutionary. The use of Linux cgroups features to allocate and enforce resource constraints and isolation means that jobs can be scheduled to minimize or eliminate contention for system resources. Additionally, since a container can be isolated from the host, there is no need for the system administrator to be concerned about conflicts with other containers or need to deploy alternate OS packages and libraries for individual users. The National Energy Research Scientific Computing Center (NERSC) developed Shifter [11], which is an environment that can execute Docker containers using a variety of batch workload managers. Similarly, the Laurence Berkeley National Laboratory (LBNL) has developed Singularity [12], which is another container runtime focused on application mobility using a different image format. Finally, Charliecloud [13] from Los Alamos National Laboratory (LANL) developed an infrastructure to execute Docker images with no privileged operations or use of daemons. These three containers runtimes are currently the leaders in the HPC community.

This paper provides background on Linux containers and Docker, and how the native Docker platform can be extended to meet the requirements of HPC. It then describes different approaches for leveraging existing Docker APIs and plugins to address the unique constraints of existing HPC architectures. Next, we present details on a prototype implementation of the proposed approach and supporting benchmark results that compare the performance of the prototype against a standard Docker deployment; more general use cases and performance studies have been covered by previous works, such as that by Containers and virtual machines at scale [14] and HPC Containers in Use [15]. Finally, we close with a discussion of future work and conclusions.

II. Background

Docker is a tool for user-level virtualization, which is a server virtualization method whereby the operating system's kernel allows for multiple isolated user space instances, referred to as containers. This allows multiple applications/processes to execute as if they are running on a dedicated server, where in reality they are all using a shared resource. In addition, the server operator has the authority to regulate workloads and resources across these isolated containers. Because these containers are isolated, services executed in and properly resourced in one container will not be visible to other containers, even if they are running on the same host.

Docker started as a component of the 'Platform as a Server' provider dotCloud [16]. Docker originally utilized the Linux Container (LXC) Platform [19] runtime, a user space interface for the Linux kernel that allows users to create and manage Linux containers. LXC's function is to create an environment as close as possible to a standard Linux installation without the need for separate kernel. The benefits of Docker were embraced immediately by developers. One of the most immediate benefits was the use of Docker for environment standardization in development and operations (DevOps). Prior to Docker, testing organizations had to take special considerations to synchronize the development cycle, but by using Docker, developers could ensure that the environments used to develop and test the software would be consistent.

In March of 2014, Docker released an update to the environment that included libcontainer [18], which replaced LXC as the interface for access to Linux kernel isolation features. Figure 1 shows the basic architecture and the runtimes supported. This change allowed Docker to have direct access to container APIs instead of relying on third-party technology. The change also provided interface abstraction that allowed Docker to continue to support LXC as well as other execution environments, such as libvirt. Docker also teamed up with other companies such as Microsoft, IBM, Google, RedHat, and the Linux Foundation to create the Open Container Initiative (OCI), which has a charter for creating open industry standards around container formats and runtimes. This standard ensures that developers will be able to run their containerized applications on any compliant platform.

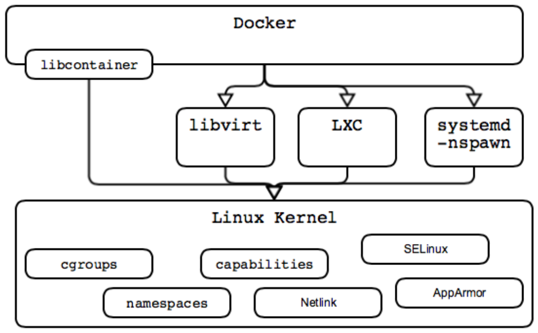

Figure 1: Basic Docker Architecture

The following section describes the main Docker architecture components.

A. Docker Client

The client interacts with the Docker daemon, typically via the command line interface *docker* command. This command actually interacts with the Docker daemon's REST API, using a UNIX socket by default (or optionally a TCP socket) to communicate with the Docker daemon. The Docker client can communicate with more than one daemon.

B. Docker Daemon

The Docker daemon accepts Docker client connections via the REST interface or UNIX socket and exposes Docker Engine functionality. The Docker daemon implements processes monitoring, container execution, and general process/image management. By default, the Docker daemon will listen on the UNIX socket, and it is generally encouraged for various security reasons [19] to be the only form of connection unless the API is required to be exported outside of the host.

C. *Docker Engine*

The Docker engine provides the execution behind the Docker daemon. The Docker Engine implements the libcontainer interface, now under the runC project which implements the Open Container Specification v1 [20]. This creates the required kernel namespaces, cgroups, handles, capabilities, and filesystem access controls.

D. Docker Objects

When using Docker, various objects are created. These objects include images, containers, networks, volumes, plugins, and other objects. This section gives a brief overview of the common objects.

1) Images: An image is a read-only template with instructions for creating a Docker container. Often, an image is based on another image, with some additional customization. For example, you may build an image which is based on the CentOS base image, but your image requires an additional application and configurations. It is common practice to use the FROM statement to "pull in" the base image and then recast the resultant image to a new image.

2) Containers: A container is a runnable intance of an image. Containers can be created, started, stopped, moved, and deleted using the Docker REST API or CLI. Containers can be connected to one or more networks, attached to storage, or can be exported to a new image based using its current state and contents.

3) Networks: Containers can be connected together via a set of network interfaces controlled by the docker engine. These interfaces can access either the host network stack (no network namespace isolation), or connect other containers together by a variety of different networking stacks, such as bridge, overlay, MACVLAN.

4) Volumes: Data volumes are the preferred mechanism for perisiting data generated and used by the container. The use of host bind mounts are dependant on the host filesystem structure while data volumes are managed by Docker and are independent of the host.

5) Plugins infrastructure: The Docker plugin infrastructure allows for out-of-process extensions which can add capabilities and features to the Docker Engine. Some common extensions are for additional filesystem support, authorization (AuthZ), authentication (AuthN) mechanisms, and for host device access (GPGPU).

III. MOTIVATION

The use of containers on HPC system has been well documented [11, 15]. The growth has been driven by a number of factors. Primary among these is the development of software and packaging of the dependent libraries and configurations into a single, portable, transferable package. In some cases, users require specific tools and applications which are difficult to install on the host system so putting them into a transferable package eliminates the need for complex environment setup on each system. In other cases, use of containers is the simplest method, just load-and-go, supporting the distribution of software that has been previously developed and packaged with the knowledge that it will work on the host environment. Case in point is Tensorflow [21], it has many build dependencies and requirements on compilers and build infrastructure support.

There are several existing HPC developed container runtimes that provide the user with the ability to execute containers. These runtimes have demonstrated that containers can be used effectively for science research and the performance is shown [11, 15, 22] to be very close to that of native execution. Technologies like NERSC Shifter, LBL Singularity, and LANL Charliecloud represent the leaders in HPC container runtimes. All of these implementations have a common set of characteristics. These include:

- Deployment of user-defined images
- Leverage standard OCI (libcontainer v1) images and use of compliant registries
- Ability to access host resources and devices
- Secure implementation (no root access or privilege escalation allowed)

The rest of this paper answers the question "why not use standard Docker" for HPC workloads. We present a strategy for how to address the requirements for running containers using HPC resources, secure the infrastructure so that users cannot request elevated privileges, provide access to host resources, and preserve standard Docker semantics and architecture.

IV. IMPLEMENTATION OVERVIEW

The Docker architecture has some drawbacks when deploying on HPC systems, namely the reliance on local disk, access to host resources, and user authentication and authorization. We will briefly describe a strategy to address container execution which remains consistent with the standard Docker framework. This strategy addresses three key objectives for executing containers, which are:

- Operate in a secure multi-user environment
- Provide scale-out using HPC storage resources
- Allow access to host level resources

By utilizing the Docker plugin infrastructure, it is possible to extend the Docker functionality to address the above and still maintain all the features of Docker. We will briefly describe some of the approaches and provide the rationale for the approach we chose.

A. Authorization Controls

The Docker command (*docker*) uses a REST API to communicate to the daemon. Docker's default authorization model implements an all or nothing policy. Any user with sufficient Linux permissions, typically members of the Unix group 'docker', can access the daemon and run any client command. Running in a shared environment we require greater access controls and the ability to set finer grain access with the environment. Fortunately, Docker allows extensions to the basic authorization framework by using the Docker Engine plugin architecture. Using a Docker authorization plugin, the daemon can be configured for granular access polices for controlling access to various commands and option usage. For example, Docker engine access ports can be secured with client access certificates to ensure that only trusted sources are allowed access.

Docker's plugin infrastructure [23] can be used to extend Docker capabilities by loading, removing, and communicating with external components using the documented API. Using this mechanism, the daemon can be configured to leverage extended authentication mechanisms. This approach can be used to authenticate user requested actions to the Docker daemon based on the current authentication context, requested command, and arbitrary payload information. The authentication context contains all user details and the authentication method. The command context contains all the request actions based on the REST API requested action, and also any additional payload information, such as flags and optional user supplied options.

The system administrator is responsible for registering plugins as part of the Docker daemon configuration and startup. When a client request is made to the Docker daemon through the CLI or via the Engine REST API, the authentication subsystem passes the request to the installed authentication plugin. The request contains the user identity and command context. The plugin is responsible for deciding whether to allow or deny the request based on the site defined policy.

Figure 2 shows the sequence of events to allow or deny a user request based on user identification, requested command (REST call), and payload options.

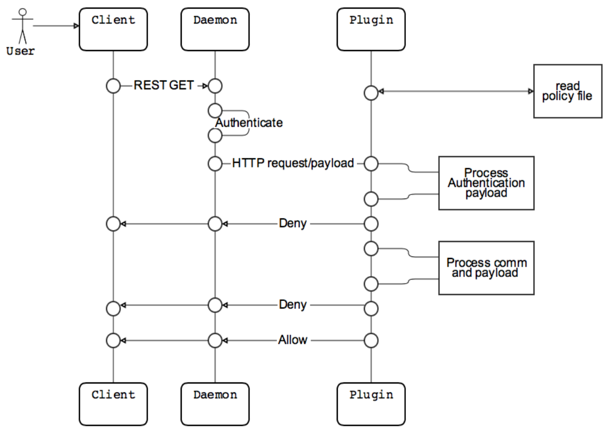

Figure 2. Authorization Request/Response Scenario

Each request sent to the plugin includes the authenticated user identity, the HTTP headers, and the request/response body. The authorization plugin will approve or deny client requests to the daemon, based on the "authentication context", command context, and optional payload attributes. The *authorization* context contains the user identity based on a CA certificate name, the authentication method in use, and other information such as the location. The *command* context contains the requested action, based on the REST function being called, and any optional arguments to the command (an example could be setting the privileged flag to *true*). The plugin algorithm first validates the incoming user identity against the configured site access policy. Secondly, the requested command is verified against the permitted operations, such as *create* container. And lastly, if command payload contains any options not allowed by the site policy, the user request is denied. This last option is important, as it is filtering on command line options, such as request capabilities or extra privilege requests. The following is an example of a policy file:

```
{
   "name": "policy_1",
   "users": [
      "authclients"
   ],
   "allow_actions": [
      "docker_*",
      "image_*",
      "container_create"
   ],
   "deny_payloads": [
      "Privileged",
      "CapAdd"
   ]
}
```

The basic plugin architecture was taken from Twistlock's [24] contributions to the Docker community and was adapted to add extra control over command payload options, such as requests for extra capabilities or elevated security. The above is an example of a site policy file for a given user using the "authclinets" certificate, allowing the caller to issue any docker_*, image_*, or *container_create* REST calls to the daemon. The added security options will deny the user from requesting any additional capabilities, such as requesting privileged operations.

B. *Docker Graph Storage*

The Docker daemon/engine stores and manages images, containers, and other meta-data information using storage graph drivers. Typically, this information is stored under */var/lib/docker*, which is located on a local disk. Many HPC nodes run diskless and use a network accessible parallel filesystem for data storage. Therefore, we need to configure an alternate on-node storage solution. Simply switching this to use tmpfs (in-memory file system) on compute nodes is also not acceptable. Even though Docker at the node level will attempt to share the container contents between several processes using the same base image, each container will incur a storage overhead for the read-write layer. Container operations will persist containers even after the containers have stopped, unless requested to be removed by the user/administrator. This can lead to storage exhaustion unless this space is carefully managed.

A good solution is to use remote off-node storage. Figure 3 shows how the out-of-process image graph can implement the different layers of the Docker image on shared storage.

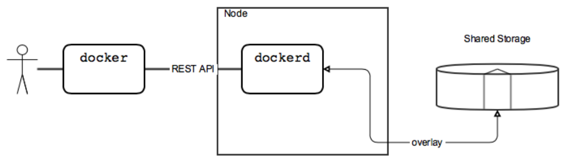

Figure 3: Out-of-process Graph Driver

All Docker images are constructed from layers using a storage (graph) driver. Dockerd can be configured to use any of the following drivers: overlayfs, aufs, btrfs, etc. The most common graph is overlayfs. Overlayfs layers two or more directories on a single host and presents them as a single merged directory. These directories are called layers and the unification process is referred to as a union mount. Figure 4 shows how a Docker image and a Docker container are layered. The image layer is the lowerdir and the container layer is the upperdir. The unified view is exposed through a directory called "merged" which is effectively the container's mount point.

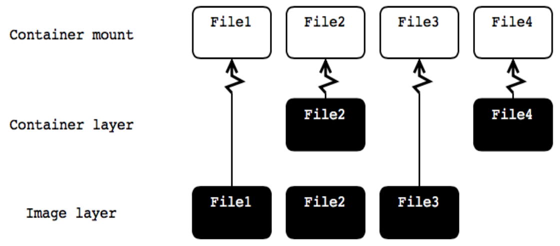

Figure 4 : Overlayfs File Structure

Where the image layer and the container layer contain the same files, the container layer "wins" and obscures the existence of the same files in the image layer.

While the standard overlay driver only works with a single lower OverlayFS layer and hence requires hard links for implementation of multi-layered images, the newer overlay2 driver natively supports up to 128 lower OverlayFS layers. This capability provides better performance for layer-related Docker commands such as docker *build* and docker *commit*, and consumes fewer inodes on the backing filesystem.

Our proposal is to implement an overlayfs2 filesystem which mimics the current standard Docker implementation, but use xfs (or ext4) to loopback mount the layers from a file on shared storage. This loopback mount is similar to that already being used on other products such as Cray Docker for compute and Shifter. Formatting this loopback file as an xfs file system provides better scaling and metadata performance than stock ext4.

One adaptation to the above concept is to use a hybrid approach, splitting out certain directories which need higher performance, namely the *container* and *overlay* directories which are used for read/write operations. Figure 5 shows the concept of using different storage options

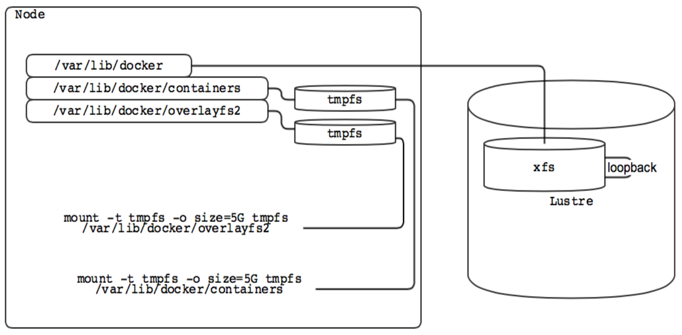
Node
/var/lib/docker
/var/lib/docker/containers
/var/lib/docker/overlayfs2
tmpfs
tmpfs
xfs
loopback
Lustre
mount -t tmpfs -o size=5G tmpfs /var/lib/docker/overlayfs2
mount -t tmpfs -o size=5G tmpfs /var/lib/docker/containers

Figure 5 : Hybrid-storage Graph

C. *Allow Containers toAaccess Host-level Resources*

Containers may require access to host-level resources that cannot be virtualized, such as non-virtual Ethernet device drivers or external storage.

Docker, in its default operation mode, implements several privileged Linux namespaces to isolate the container from the host. These namespaces can be manipulated by the operator and user to attain the correct level of isolation or pass-through device access. Using the CLI a user can enable or disable the namespace isolation used when executing the container. For example, "–net=host" disables the network namespace so that a process within a container has access to the hosts networks.

V. BENCHMARKINGS AND COMPARISIONS

In section IV we described some of the potential options for how to implement the extensions to Docker daemon and engine to support the unique needs of HPC use cases. Here we compare some of these approaches. For the authentication plugin, we do not have benchmarks, but will illustrate the components working and selectively denying user access to various commands and options. To benchmark the performance of the image graph, we use a Docker graph benchmark *docker-storage-benchmark.* This benchmark can be used to benchmark various file operations from within a container and hence measure the image graph file performance. Figure 6 shows the benchmark performance, using a local btrfs filesystem, against a loopback xfs mounted from Lustre and a hybrid filesystem using both tmpfs and a loopback filesystem. Table 1 shows the time it takes to download an image from an external image repository (dockerhub) and to create an image using these different image graph and backing store solutions.

The image graph benchmarks are testing the container graph performance, typically an application within the container would read and write directly to the parallel filesystem (PFS) via a container volume mount and not to the container's filesystem directly. That said, the following benchmarks do highlight the performance characteristics of tmpfs, loopback xfs, and the hybrid filesystems approach.

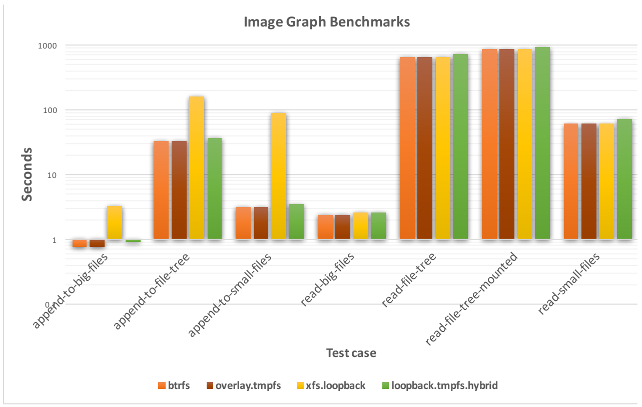

Figure 6: Image Graph Benchmarks

While these tests are not exhaustive, they do illustrate in-container filesystem performance using different types of backing store. When the hybrid graph is used, as expected we achieve close to native overlay over tmpfs performance, with a slight overhead due to metadata operations going back to the loopback file residing parallel network filesystem.

As previously mentioned, best practices suggest writing directly to the PFS filesystems using a host bind mount.

Table 1: Image Download Times

Image size	btrfs Login	overlay/tmpfs Compute	Overlay/loopback xfs Compute
800MB (c) 1.2GB	1:20.27	1:18.13	1:19.90
5.4GB (c) 15.1GB	18:10.95	9:51.14	11:57.72

The above table shows that we are achieving slightly better performance on the compute nodes for image download using the loopback device as compared to the login node's implementation of btrfs.

VI. FUTURE WORK

There are still a number of topics that require further investigation. First, the current method of passing the user identity is based on CA x509 certificates. We are investigating using the standard UNIX credentials such as user id and group id and passing these unaltered to the Docker daemon. We are also considering several alternate methods of projecting the image graph to the node, and making it read-only and shared between nodes. Finally, an investigation is being conducted into how to enable the site administrator to define site specific parameters, such as standard container mount points (/home), host device mappings, and other site wide parameters.

VII. CONCLUSION

With the increased interest in container technology, creating and managing containers for scientific and HPC community becomes much easier for developers and scientists when using an industry standard platform. It is likely that container computing coupled with new methods of images and package management and orchestration will dominate how applications are developed, delivered, and executed in the coming years. By using standard interfaces, users can easily leverage a consistent approach for executing workloads from laptop to HPC to cloud, lessening the learning required when moving between environments, and enabling the user to focus on the application and not the idiosyncrasies of each systems.

While we consider this investigation in scaling Docker as exploratory, we believe it will serve as the starting point for Cray to develop extensions for scalability that can be leveraged more broadly by the community.

ACKNOWLEDGMENT

The author would like to acknowledge the following people for their assistance and insights during this research – Kitrick Sheets, Pierre Carrier, Larry Kaplan for Cray, and Christian Kniep from Docker.

REFERENCES

[1] Docker, https://www.docker.com/.

[2] rtk, https://coreos.com/rkt/.

[3] https://www.theregister.co.uk/2014/05/23/google_containerization_two_billion/.

[4] Azure Container Services, https://azure.microsoft.com/en-us/services/container-service/.

[5] Google Clound Kubernetes Engine, https://cloud.google.com/kubernetes-engine/.

[6] Amazon Elastic Contatiner Service, https://aws.amazon.com/ecs/.

[7] Red Hat OpenShift, https://www.openshift.com.

[8] Novel SUSE Caas Platform, https://www.suse.com/products/caas-platform/.

[9] Linux cgroups, https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt.

[10] Linux namespaces, https://en.wikipedia.org/wiki/Linux_namespaces.

[11] D. Jacobsen, S. Canon, "Contain this, unleashing Docker for HPC," presented at the Cray User Group., Chicago, IL., 2015

[12] Singularity, http://singularity.lbl.gov

[13] Reid Priedhorsky, Tim Randles, "Charliecloud: unprivleged containers for user-define software stacks in HPC", SC '17 Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, Article No. 36. Denver, Colorado, November 2017

[14] . Lucas Chaufournier, Prateek Sharma, Prashant Shenoy, Y.C. Yay, "Containers and virtual machines at scale: a comparative study",Middleware '16 Proceedings of the 17th International Middleware Conference, Trento, Italy, December 12 -16, 2016

[15] Jonathan Sparks, “HPC Containers in Use”, presented at Cray User Group, WA., 2017

[16] dotCloud, https://blog.docker.com/2013/10/dotcloud-is-becoming-docker-inc/

[17] LXC (Linux Containers), https://linuxcontainers.org

[18] Libcontainer, https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/

[19] Docker security, https://docs.docker.com/engine/security/security/

[20] Open Container Initiative, https://www.opencontainers.org

[21] “Building Tensorflow”, https://www.tensorflow.org/serving/setup

[22] Donald Bahls, “Evaluating Shifter for HPC applications”, presented at the Cray User Group., London, UK., 2016

[23] “Use Docker Engine plugins”, https://docs.docker.com/engine/extend/legacy_plugins/#types-of-plugins

[24] “Docker AuthZ Plugins: Twistlock’s Contribution to Docker Security”,https://www.twistlock.com/2016/02/18/docker-authz-plugins/*)*