2017 IEEE International Conference on Cloud Engineering

Enabling Docker Containers for High-Performance and Many-Task Computing

Abdulrahman Azab
Department of Reseach Computing, University Center for Information Technology (USIT)
University of Oslo, Norway
Email: abdulrahman.azab@usit.uio.no

***Abstract*—Docker is the most popular and user friendly platform for running and managing Linux containers. This is proven by the fact that vast majority of containerized tools are packaged as Docker images. A demanding functionality is to enable running Docker containers inside HPC job scripts for researchers to make use of the flexibility offered by containers in their real-life computational and data intensive jobs. The main two questions before implementing such functionality are: how to securely run Docker containers within cluster jobs? and how to limit the resource usage of a Docker job to the borders defined by the HPC queuing system? This paper presents Socker, a secure wrapper for running Docker containers on Slurm and similar queuing systems. Socker enforces the execution of containers within Slurm jobs as the submitting user instead of root, as well as enforcing the inclusion of containers in the cgroups assigned by the queuing system to the parent jobs. Different from other Docker supported containers-for-hpc platform, socker uses the underlaying Docker engine instead of replacing it. To eveluate socker, it has been tested for running MPI Docker jobs on Slurm. It has been also tested for Many-task computing (MTC) on interconnected clusters. Socker has proven to be secure, as well as introducing no additional overhead to the one introduced already by the Docker engine.**

Keywords - Slurm, Docker, containers, HPC

I. Introduction

Linux containerization is an operating system level virtualization technology that offers lightweight virtualization. An application that runs as a container has its own root file-system, but shares kernel with the host operating system.

Docker [1] is the most popular platform among users and IT centers for Linux containerization. A software tool can be packaged as a Docker image and pushed to the Docker public repository, Docker hub, for sharing. A Docker image can run as a container on any system that has a valid Linux kernel. There are other options in the field, e.g. Rocket [2] and LXD [3] for Linux, in addition to Drawbridge [4] for Windows. Despite the existence of different alternatives, Docker is still the most robust, user friendly, and well supported platform. To make the best use of containers, it has become demanding to enable containers on HPC platforms. HPC is widely used by researchers in different fields, e.g. physics and bioinformatics, to speedup applications that may take months or even years on one PC. Enabling containers on HPC platforms is beneficial for system administrators, removing the burden of software installation and maintenance hell, since a container that runs on a Linux system should run on any other Linux system. This is also beneficial for users, speeding up the process of software deployment on the HPC system since system administrators would need to exert minimal effort to deploy a containerized software. Before deciding to deploy Docker containers on a queuing system in production, there are two main issues to resolve: First, the default configuration of Docker is to run containers as root. For the container process to be bounded with the user privileges and for sys-admins to keep record of who is running what, the container process must run as the submitting user. Second, the resource consumption (CPU and memory) of a Docker container running inside a job must be bounded with the limitations for resource consumption set by the queuing system for this particular job. This paper introduces *socker*, a secure wrapper for running Docker containers on Slurm [5] and similar queuing systems, e.g. Moab [6] and PBS [7]. Socker is tested by the research computing group at the University Center for Information Technology, USIT. Socker is not a replacement of the Docker engine for running and managing containers. It runs Docker containers as the submitting user in addition to enforcing the membership of a running container, as part of a HPC job, in the cgroups [8] assigned by the queuing system to this job. Elementary testing of socker has been carried out to run MPI jobs on a Slurm cluster. Socker has been also tested for Many-Task computing (MTC) [9] running container jobs on a system of interconnected clusters. Socker has proven to be secure, in addition to introducing almost no computational overhead. Further developments of Socker are ongoing, and it is now available on github [10].

II. Related Work

Numerous efforts have been carried out in the direction of enabling Docker containers on HPC platforms. Docker Swarm [11] is the container orchestration platform offered by Docker. Swarm offers great scalability, but it is very poor in terms of security. In addition, the scheduling and resource management is too basic which makes it insufficient for large HPC systems in production. Google's Kubernetes [12] is a competitor to Docker Swarm. It offers more control over running containers, in addition to self-healing, but it is still poor in terms of scheduling and resource management. In addition, it is very complex to install and maintain. A common issue for both Docker Swarm and Kubernetes is that they are both designed to be standalone queuing systems, not runners or wrappers. So, one cannot have Swarm or Kubernetes plugged into already installed queuing system, e.g. TORQUE or Slurm, but has to replace the entire queuing system. HTCondor [13] has recently provided support for Docker through the *docker universe* [14]. HTCondor is more trustworthy for HPC systems since it is a well known and well tested platform for both HPC and HTC. We have deployed Docker with HTCondor

978-1-5090-5817-4/17 $31.00 © 2017 IEEE
DOI 10.1109/IC2E.2017.52

on a test environment, and the implementation was proven to be very user friendly. But it lacks flexibility, e.g. in terms of mounting volumes. In addition, it does not have any control over resource usage by the running containers. Singularity [15] is an engine for running Linux containers on HPC platforms. It is installed on many HPC clusters in production. Singularity containers are lightweight, different from the Docker layered containers. We have tested singularity containers on Abel and they prove to be faster and less resource consuming than similar Docker containers. The issue with singularity that it is still not as popular as Docker among the scientific community which we are serving. It is not even any close too. Singularity supports conversion of Docker images to singularity images. But after deeply testing this feature, it is still not stable. Many conversion attempts failed miserably. Therefore we cannot rely on singularity for production deployment until it proves that all docker containers, or at least most, can be converted to singularity containers. Shifter [16] is a platform to run Docker containers on Slurm. It is developed by the National Energy Research Scientific Computing Center (NERSC) and is deployed in production on a Slurm cluster of Cray supercomputers. Shifter is however not following the Docker standards and is not using the Docker engine for running and managing containers. It has its own image format to which both Docker images and VMs are converted. The Docker engine is replaced by *image manager* for managing the new formated images. Previously NERSC introduced MyDock [16] which is a wrapper for Docker that enforces accessing containers as the user. MyDock however did not provide a solution for enforcing the inclusion of a running container in the cgroups associated with the Slurm job. In addition, both shifter and myDock enforces accessing as the user by first running as root and mounting `/etc/passwd` and `/etc/group` in each single container, then lets the user access the container as him/herself. Socker adopts a more secure and less complex strategy by running containers as the user from the very beginning, avoiding any threats that may result in running containers as root. Similar to singularity, shifter being immature, and relying on its own image format and own engine, doesn't make it dependable for large production system. It places the whole development and maintenance responsibility on the Shifter development team. Developing a runner that uses the Docker engine is more realistic since Docker is a well known and maintained platform in addition to being used by millions of users and a number of IT research support centers worldwide.

III. Docker

Docker is a new tool that automates the deployment of applications inside Linux containers. It provides a layer of abstraction and automation of operating-system level virtualization. Docker is not itself a novel technology, but is a high-level tool which was initially built on the top of LXC [17] Linux containers API, and provides additional functionality. Docker containers are executed and controlled by the Docker engine, which is different from the hypervisor for VMs. Since it does not include a full guest OS, a Docker container is smaller and faster to start up than a VM. A Docker container instead mounts a separate root file-system, which contains the directory structure of the Unix-like OS plus all the configuration files, binaries and libraries required to run user applications. The boot file-system which contains the boot-loader and kernel is not included in the container, containers instead use the host boot file-system [18]. When Docker mounts the container root file-system, it starts read-only, as in a traditional Linux boot, but then, instead of changing the file system to read-write mode, it uses union mount [19] to add a read-write file system over the read-only file system. It is also possible to have multiple read-only file systems stacked on top of each other. Each of those file-systems can be considered as a layer [20].

Since not all file-systems are layered, e.g. Ext, Docker builds its layered file-system on the top of the host file-system (known as *backing* filesystem in the Docker terminology). Docker supports different layered filesystem drivers, e.g. Overlay FS, AUFS, and BTRFS. A typical setup is to have Docker with OverlayFS on the top of Ext4 backing filesystem.

IV. The HPC Infrastructure

Socker is planned to be deployed in production as a runner for Linux containers on Two clusters at the University of Oslo:

A. Abel

Abel [21] is the high performance computing facility at the University of Oslo hosted by the University Center for Information Technology (USIT) and maintained by the Research Infrastructure Services group. Named after the famous Norwegian mathematician Niels Henrik Abel (1802 — 1829), the Linux cluster is a shared resource for research computing capable of 258 TFLOP/s theoretical peak performance. At the time of installation Abel reached position 96 on the Top500 list.

1) Hardware: Today, Abel's compute capacity is 702 nodes, 11392 cores, total memory of 40 TiB, and total local storage of 400 TiB using BeeGFS [22]. Compute nodes are all dual Intel E5-2670 (Sandy Bridge) based running at 2.6 GHz, yielding 16 physical compute cores. Each node have 64 GiBytes of Samsung DDR3 memory operating at 1600 MHz, giving 4 GiB memory per physical core at about 58 GiB/s aggregated bandwidth using all physical cores. Compute nodes are connected to a large common scratch disk space. All nodes have FDR InfiniBand running x86_64 CentOS 6.7. The storage is provied as two equal size partitions */cluster* and */work* each capable of a performance of about 6-8 GiB/s when doing sequential IO [1].

2) Queuing System (Slurm): Abel uses the Slurm (Simple Linux Utility for Resource Management) workload manager [5] as the queuing system. Slurm has three main functions. First, allocating access to compute nodes users for a predefined time duration. Second, it provides a framework for starting, executing, and monitoring submitted jobs. Finally,

[1]The hardware information is collected on Dec 20 15:33:51 CEST 2016

it manages a queue of pending jobs. To ensure that running jobs won't consume more resources on the compute node(s) than what is assigned by the system, Slurm creates three main cgroups [8] for each running job (cpuset, memory, and freezer) then enforces inclusion of each process inside a running job in the three cgroups. If a process X in a running job consumes e.g. more memory than the value stored in the job's memory cgroup, it gets killed by Slurm.

B. Colossus

Colossus is compute cluster designed to be as similar to Abel. Colossus is the research computing within the Services for Sensitive Data (TSD) [23] which is a secure e-infrastructure for storing and processing sensitive data. Abel and Colossus have almost the same installed software modules.

V. Motivation

With the use of Linux containers and the very user friendly Docker engine, researchers can now install tools on their platform of choice, e.g. Ubuntu for bioinformaticians and CentOS for physicists, as a Docker image and publish it on the Docker hub or just share the Dockerfile with collaborators. Then anyone who has a Docker engine and a proper Linux kernel may run the image and get the same functionality. This makes life easier for software developers in that they no longer need to write multiple installation guides and test on different Linux distributions. It also makes life easier for system administrators, as instead of receiving software requests of type: "I need software X, and here is the installation guide, please install it!", requests will be of type: "I need software X, here is the name of its Docker image, please pull it". In addition, no software maintenance will be needed and no dependency conflicts will arise when installing new software.

On our Abel and Colossus clusters, we receive large number of software installation requests due to which we created a software request queue. In addition, we have a lot of already installed modules to maintain. Enabling Docker containers on Abel and Colossus is very beneficial for both our users and system administrators.

VI. Socker

Socker is a wrapper for running Docker containers securely inside HPC jobs. It is mainly designed to enable the users of our Slurm clusters (Abel and Colossus) to run Docker enabled jobs. Since our compute nodes are running CentOS 6.7, which standard kernel is 2.6, Socker is designed to be able to work with Docker 1.7 (since support for CentOS 6 is dropped for newer Docker releases). It can also deal with newer Docker versions [2]. There are mainly two functions provided by socker: First, run each container process as the user who submitted the job in order to make the container bounded by the user's capabilities. Second, bound the resource usage of any container, called inside a job, by the limits set by the queuing system for the job. Slurm (and recently Moab) use cgroups to control job resource consumption. Socker is also designed to be simple and portable. Figure 1 shows the structure of Socker. Users are categorized into system administrators and regular users. System administrators are given privileges to run the *docker* command, i.e. members of the `docker` group. Regular users can run Docker only through Socker. Socker is composed of two executables:

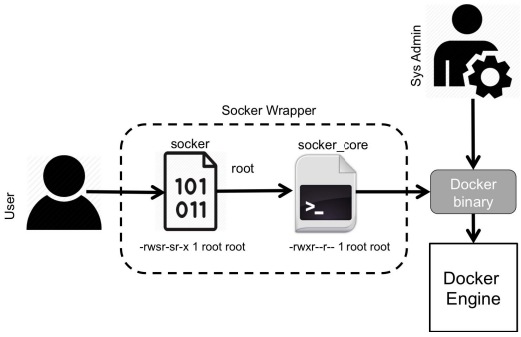


Fig. 1: Socker Wrapper

socker binary, and *socker_core* script (currently bash). socker binary is the entry point for users to run inside Slurm job scripts. Socker binary performs the following:

1) Collects the image name and the command to be executed. These are provided by the user as command-line arguments.
2) Collects the user information, i.e. UID and GID.
3) Change the UID to 0, i.e. run the following command as root.
4) Calls socker_core passing the previously collected values as arguments.

To make socker binary executable by regular users and run commands as root at the same time, it needs to be configured as follows:

- Enables `setuid` for all: `chmod +s socker`. This will make it executable by everybody and give it permission to set the user ID, e.g. to root.
- socker binary must be a 'binary', not a script (that starts with #!), since most Linux systems for security reasons ignore the above configuration for scripts.

Socker binary internally runs socker_core, which is executable only by root. socker_core performs the main Socker functionality:

1) Runs the Docker image as the user, using `docker run -u UID:GID`, and detach from the container.
2) Enforces membership of the container process in the cgroups created by Slurm for the job.

Algorithm 1 describes the overall Socker functionality for Slurm deployment.

The functionality is divided between binary and bash script, to support flexibility. Maybe for another Slurm cluster, system administrators would prefer to use their own, e.g. Python script, as socker_core to support different settings. Listing 1

[2]Socker doesn't currently support user namespaces since it is still an immature feature in the Linux kernel and may contain bugs

Algorithm 1 Container launching procedure (Slurm implementation)

Initialization:

IMG {Docker image to run}

CMD {Command to execute on the container}

Start:

uid ← getuid() {Store UID of the submitting user}

gid ← getgid() {Store GID of the user's default group}

jobid ← $Slurm _JOB_ID {Store the ID of the Slurm job}

Set UID ← 0 {Change the user to root}

call SOCKER_CORE(uid,gid,jobid,IMG,CMD) {Run socker_core as root}

Procedure **SOCKER_CORE**(UID,GID,JOBID,IMG,CMD): {$socker\_call$ script procedure}

Start:

cid← **call docker run**(detached=$true$,user=UID:GID, image=IMG, command=CMD) {Start container as detached and store it's ID}

pid←getPID(cid) {Store container's process ID}

if ∄ **process**(pid) **then** {Failed to start the container}
  Report Error
  Exit

for all cg ∈ **CGroup**($slurm/UID/JOBID$) {for each cgroup created by Slurm for job: JOBID}

  classify process(pid) ∈ gc {Set container's process as a member in gc}

call docker wait(cid) {Wait for container cid to run the command and exit}

call docker remove(cid) {Clean up after the container exits}

End

presents an example for using socker to run parallel jobs in a Slurm script.

Listing 1: Example using socker to run a parallel Slurm job

```
#!/bin/bash
#
#SBATCH --image=image_name
#
#SBATCH --nodes=1
#
#SBATCH --partition=regular
#
#SBATCH -N 2

module load socker

srun -n 32 socker python myScript.py args
```

VII. Performance Evaluation for HPC

To enable Socker on our main cluster, Abel, Docker 1.7 has been installed on compute nodes. Currently Docker doesn't support shared image repositories among hosts, so each compute node has it's local image repository (layered filesystem). To avoid storing images on the local disks of compute nodes, all image repositories has been allocated on the shared BeeGFS filesystem. Since Docker doesn't currently support BeeGFS as a backing filesystem, we mounted Ext4 images as loopback devices on compute nodes as repositories for Docker images. Another issue is that we cannot rely on the compute nodes pulling images from the Docker hub, as this might be very slow, especially for images with a lot of layers, in addition to requiring all compute nodes to have Internet access (which is not possible for Colossus). To resolve the issue, we installed a local Docker registry. Upon user requests, images are pulled from the Docker hub and pushed to the local registry. System layout is depicted in Figure 2.

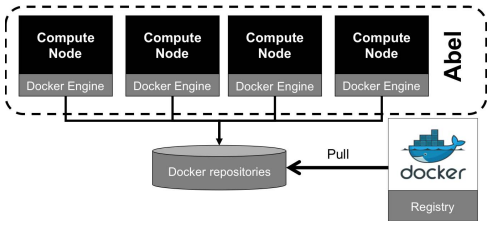


Fig. 2: Abel Cluster Configuration for Running Docker Containers

To evaluate the performance of socker for HPC applications, we performed an experiment on our main Slurm cluster, Abel. The aim of this experiment was to indicate whether socker introduces considerable computational overhead. We used one image from Docker hub, `genomicpariscentre/bowtie1`, for running sequence alignment using Bowtie [24]. The input dataset used is ≃5 GiB *Fastq* file, to be aligned against human genome *hg19*. We used the pMap package [25] to divide the alignment into parallel MPI jobs. Three compute nodes (2*16 + 20 cores) were reserved for this docker test. The experiment included the following:

1) Alignment with pMap using Native Bowtie binary.
2) pMap running the Bowtie Docker image using `docker run` as a system administrator account that is a `docker` group member.
3) pMap running the Bowtie Docker image using Socker and a regular user account.

The Bowtie image has been pulled on the three nodes in advance, to avoid any additional latency pulling the image

from the Docker hub. All Docker runs (both direct docker and socker) were configured to remove the container after exiting. Figure 3 presents the total run time against the number of parallel processes. 1, 8, 16, and 32 processes were used. It is

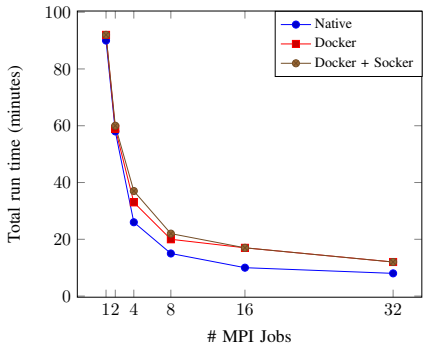


Fig. 3: Total run time of parallel pMap MPI jobs using Bowtie aligner as: Bowtie native binary, Bowtie Docker image using `docker run`, and Bowtie Docker image using socker

clear that running container jobs (for both docker and socker) introduces very small overhead compared to running native jobs. It is noticed that the overhead grows with increasing the number of parallel processes. This is resulted from: 1) the overhead of starting new containers, 2) Removing exited containers is enabled, which introduces additional overhead (but is necessary for cleaning up and saving disk space on compute nodes). It is also noticed that socker overall introduces almost no additional overhead compared to `docker run`.

VIII. Performance Evaluation for MTC

To evaluate whether socker is useful for MTC (Many-Task Computing), where relatively large number of jobs are submitted. We performed another experiment using larger system setup. The system is composed of three interconnected cluster portions: Three nodes in Abel, 10 nodes in our test Slurm cluster *Snabel* [3], and a HTCondor cluster *Condor01* installed on the cPouta cloud [26] [4]. The system layout is depicted in Figure 4, and the dedicated resources for the Docker experiment on each cluster are described in Table I. Throughout this section, we consider each cluster portion as a cluster. Inter-cluster job submission between the three clusters has been enabled using the Grid Universe setup of HTCondor [27], having a Condor-C component installed on a node in each cluster. This setup enables, e.g. a HTCondor job to be submitted to Slurm and vice-versa. To achieve job load-balancing, two methodologies have been tested and evaluated:

1) *HTCondor Flocking* [28]. There is a fixed list of clusters on the Condor-C node on each cluster, e.g. Abel. When the local cluster is saturated with jobs, new job submissions are pushed to the first cluster in the list, e.g. Snabel. When that is saturated as well, jobs submissions are pushed to the second, and so on. The Flocking list in this experiment is: 1) Abel, 2) Condor01, and 3) Snabel, based on the number of available cores ascending.
2) *Slick inter-cluster scheduling* [29], [30]. Each cluster has a *Slick gateway* [29] installed on the Condor-C node. When there are waiting jobs with no matching resources in the local cluster, a fuzzy logic based job-to-cluster matchmaking is carried out for each new job submission to determine the best-matching cluster [30], and the job is pushed to that cluster.

[3]Snabel is made similar to Abel in the setup but smaller in size
[4]cPouta is the main production IaaS cloud at CSC (www.csc.fi)

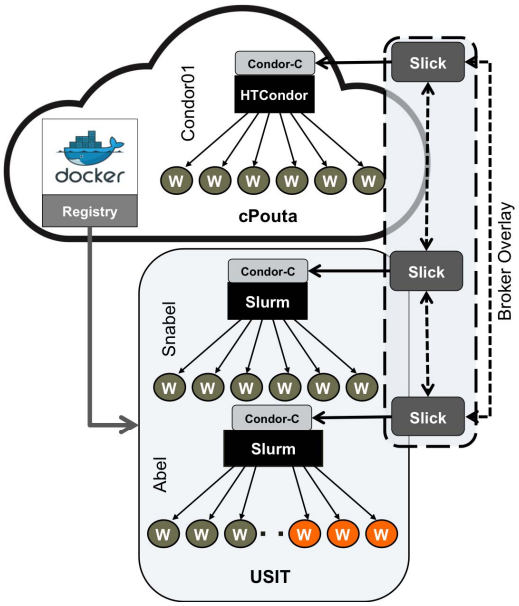


Fig. 4: Inter-Cluster container scheduling using HTCondor Flocking and Slick

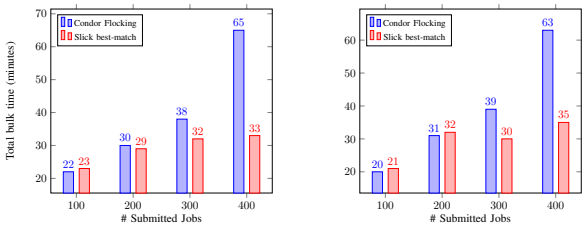
Slick has been evaluated against HTCondor flocking and other inter-cluster scheduling techniques using large peer-to-peer network simulation. In this experiment we evaluate Slick against HTCondor flocking on real system setup and job workload of docker containers that run using socker. The

TABLE I: Resource information of the interconnected docker-enabled clusters

	♯Docker-nodes/ ♯Total-nodes	♯ Cores	Scheduler	Infrastructure
Abel	3/712	52	Slurm	CentOS 6 Machines
Snabel	10/28	224	Slurm	CentOS 6 Machines
Condor01	16/16	160	HTCondor	CentOS 7 VMs

experiment included four bulk submissions: 100, 200, 300, and 400 container jobs. Each submission is divided into collections of 100 jobs each. Each collection is assigned to one cluster in the following order: Condor01←B1, Snabel←B2, Abel←B3, and Condor01←B4. Each job runs the Bowtie Docker image, described in Section VII, as a single CPU core job for mapping 1 GiB `fastq` file. Using each of the inter-cluster job load-balancing mechanisms (Flocking and Slick scheduling) long waiting jobs are pushed from the local cluster to the other two. All compute nodes are connected to a private Docker registry on cPouta for pulling the Bowtie image. Input files are stored on a Condor01 node that is accessible to all compute nodes in the clusters through SSHFS [31] mounting. Figure 5 presents

a comparison between HTCondor flocking and Slick inter-cluster scheduling. The job bulk size is depicted against the total bulk time [last-job-completion-time - first-job-submission-time]. The described procedure has been carried out using both `docker run` and socker. The whole experiment has been repeated three times, and the depicted results represents the median. The results are described as follows: In case of



(a) docker run (b) Socker

Fig. 5: Job bulk size against the total bulk time

100 jobs bulk, all jobs in both cases (slick and flocking) are submitted to Condor01 that has sufficient cores already. No bottleneck occurs, so the results are similar. In case of 200 jobs bulk, 100 jobs are submitted to Condor01 and 100 jobs to Snabel. both have sufficient number of cores. No bottleneck occurs and the results are similar again. In case of 300 jobs bulk, 100 jobs are submitted to Condor01, 100 to Snabel, and 100 to Abel. Abel has only 52 cores for docker jobs, so the inter-cluster load-balancing needed to work. Flocking pushed the remaining jobs to Condor01, according to the list, and Slick submitted to Snabel (the closest). Slick achieved shorter bulk time. In case of 400 jobs bulk, 200 jobs are submitted to Condor01 and 100 jobs go to each of Abel and Snabel. Flocking pushes the waiting Abel jobs to Condor01, waiting Condor01 jobs to Snabel, while Slick pushes both to Snabel and achieves even shorter bulk time. The results for `docker run` and socker are similar which draws a conclusion again that socker almost introduces no additional overhead.

IX. Known Issues

There are known Issues and limitations for running Docker on HPC in general and for Socker in particular:

- Containers won't solve the Hardware architecture incompatibility. For instance if there is image X with application Y that is compiled for a specific CPU architecture, Docker portability won't be useful here. Docker works above the kernel not below.
- Containers won't clear operational maintenance mess. For instance if application A is compiled using a specific version of MPI, and application B is compiled using a different MPI version, then application C has both A and B as dependencies: C won't work even if A and B are Docker images. This is a reason multiple base images is not recommended by Docker despite the fact that it is technically supported.
- Containers are mainly designed for one application per container, while VMs can have a large collection of applications. Not all users publishing images on Docker hub understand this. Our system administrators may receive a request to pull image X which contains application Y, and image X is huge containing a lot of other stuff and would cause considerable memory overhead when loaded.
- Socker enforces inclusion of a running container in the Slurm cgroups by assigning the container's process as a member of these cgroups. This is based on the assumption that a container runs only one process. Some images are configured to start multiple processes, e.g. database servers. However, such images are not interesting for us in HPC. But again, we need to double check the Dockerfile to ensure that only one process is initially started.
- Socker is designed for Docker 1.7, in which running container as user X can assign X to only one user group. In Docker 1.8+, the `--group-add` option enables assigning additional groups. This is important in some cases. For instance on Abel, we have a shared area `/projects` where each research project has a directory accessible only by this project's members. This is done by creating a file group X for each project. A Slurm user who is a member of project X may want to mount `/projects/X` in his/her containers. To do so, the user needs to be configured (in the container) as a member of his/her default group (to access his/her home directory) in addition to the file group X (to access the project area). This is currently not supported due to the described limitation.

X. Conclusions

Linux containers is a useful technology for both users and system administrators supporting portability and less overhead than traditional VMs. Docker is a user friendly platform for creating and managing containers and is used by millions of users worldwide. Enabling Docker containers on HPC clusters for scientific computing is demanded by a lot of researchers. Socker is a wrapper for running Docker containers on Slurm. Socker is designed to be simple and portable so that it can be used on any Slurm cluster. Socker runs containers securely as the user, and enforces bounding the resource usage of running containers by the amount of resources assigned by Slurm. socker introduces almost no additional computational overhead. This has been tested for both HPC and small scale MTC. We are planning to make socker queuing system independent. It is also planned to use runC [32] instead of `docker run` for running containers, since the Docker engine itself is now based on runC.

XI. Aknowledgements

This work is funded by The University Center for Information Technology (USIT) at the University of Oslo, and the nordic Tryggve [33] project which is part of NeIC (Nordic e-Infrastructure Collaboration) [34]. The infrastructure is provided by the Department of Research Computing, Services for sensitive Data (TSD) [23], and cPouta IaaS at CSC [26].

REFERENCES

[1] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[2] "rkt - app container runtime," https://github.com/coreos/rkt, accessed: 2016-05-21.

[3] "Lxd - linux containers," linuxcontainers.org/lxd, accessed: 2016-05-21.

[4] "Microsoft research - drawbridge," http://research.microsoft.com/en-us/projects/drawbridge/, accessed: 2016-05-21.

[5] M. A. Jette, A. B. Yoo, and M. Grondona, "Slurm: Simple linux utility for resource management," in *In Lecture Notes in Computer Science: Proceedings of Job Scheduling Strategies for Parallel Processing (JSSPP) 2003*. Springer-Verlag, 2002, pp. 44–60.

[6] J. Corbet, "Notes from a container," https://lwn.net/Articles/256389/, October 29, 2007, accessed: 2016-05-21.

[7] H. Feng, V. Misra, and D. Rubenstein, "Pbs: A unified priority-based scheduler," ser. SIGMETRICS '07. New York, NY, USA: ACM, 2007, pp. 203–214.

[8] J. Corbet, "Notes from a container," https://lwn.net/Articles/256389/, October 29, 2007, accessed: 2016-05-21.

[9] I. Raicu, I. T. Foster, and Y. Zhao, "Many-task computing for grids and supercomputers," in *Many-Task Computing on Grids and Supercomputers, 2008. MTAGS 2008. Workshop on*. IEEE, Nov. 2008, pp. 1–11.

[10] A. Azab, "socker: A wrapper for secure running of docker containers on slurm," https://github.com/unioslo/socker, accessed: 2016-12-01.

[11] "Docker swarm," https://docs.docker.com/swarm/, accessed: 2016-05-21.

[12] D. K. Rensin, *Kubernetes - Scheduling the Future at Cloud Scale*, 1005 Gravenstein Highway North Sebastopol, CA 95472, 2015. [Online]. Available: http://www.oreilly.com/webops-perf/free/kubernetes.csp

[13] M. Litzkow, M. Livny, and M. Mutka, "Condor - a hunter of idle workstations," in *Proceedings of the 8th International Conference of Distributed Computing Systems*, June 1988.

[14] "Docker and htcondor," https://research.cs.wisc.edu/htcondor/ HTCondorWeek2015/presentations/ThainG_Docker.pdf, accessed: 2016-05-21.

[15] G. M. Kurtzer, "Singularity 2.1.2 - Linux application and environment containers for science," Aug. 2016. [Online]. Available: https://doi.org/10.5281/zenodo.60736

[16] D. Jacobsen and S. Canon, "Contain this, unleashing docker for hpc," in *Cray User Group 2015*, April 23, 2015.

[17] "Lxc - linux containers," linuxcontainers.org/lxc, accessed: 2016-05-21.

[18] ""file system". http://docs.docker.com/terms/filesystem/. retrieved 2015-04-22."

[19] J.-S. Pendry and M. K. McKusick, "Union mounts in 4.4bsd-lite," in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, ser. TCON'95. Berkeley, CA, USA: USENIX Association, 1995, pp. 3–3. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267411.1267414

[20] "Layers," https://docs.docker.com/terms/layer/, accessed: 2015-04-22.

[21] "Moab hpc suite," http://www.adaptivecomputing.com/products/hpc-products/moab-hpc-basic-edition/, accessed: 2017-01-21.

[22] "Beegfs: The parallel cluster file system," http://www.beegfs.coml, accessed: 2017-01-21.

[23] "Tjenester for sensitive data (tsd)," http://www.uio.no/tjenester/it/forskning/sensitiv/, accessed: 2016-05-21.

[24] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short dna sequences to the human genome," *Genome Biology*, vol. 10, no. 3, p. R25, 2009.

[25] "pmap: Parallel sequence mapping tool," http://bmi.osu.edu/hpc/software/pmap/pmap.html, accessed: 2016-05-21.

[26] "cPouta IaaS Cloud," https://research.csc.fi/cpouta, Aug. 2016, accessed: 2016-12-01.

[27] D. Thain, T. Tannenbaum, and M. Livny, "Condor and the grid," in *Grid Computing: Making the Global Infrastructure a Reality*, F. Berman, G. Fox, and A. Hey, Eds. John Wiley & Sons Inc., April 2003.

[28] X. Evers, J. F. C. M. de Jongh, R. Boontje, D. H. J. Epema, and R. van Dantzig, "Condor flocking: load sharing between pools of workstations," Delft, The Netherlands, Tech. Rep., 1993.

[29] A. Azab and H. Meling, "Slick: A coordinated job allocation technique for inter-grid architectures," in *7th European Modelling Symposium (EMS)*, Nov. 2013.

[30] A. Azab, H. Meling, and R. Davidrajuh, "A fuzzy-logic based coordinated scheduling technique for inter-grid architectures," in *Distributed Applications and Interoperable Systems - 14th IFIP WG 6.1 International Conference, DAIS 2014, Held as Part of the 9th International Federated Conference on Distributed Computing Techniques, DisCoTec 2014, Berlin, Germany, June 3-5, 2014, Proceedings*, 2014, pp. 171–185.

[31] "A network filesystem client to connect to ssh servers," https://github.com/libfuse/sshfs, accessed: 2017-01-21.

[32] "runc - cli tool for spawning and running containers according to the oci specification," https://github.com/opencontainers/runc, accessed: 2017-01-17.

[33] "Neic tryggve: Collaboration on sensitive data," https://wiki.neic.no/wiki/Tryggve, accessed: 2016-05-21.

[34] "Neic: Nordic e-infrastructure collaboration," https://neic.nordforsk.org/, accessed: 2016-05-21.