D12.1 - Container-as-a-service analysis report

Deliverable No.:	D12.1
Deliverable Name:	Container-as-a-service analysis report
Contractual Submission Date:	30/04/2018
Actual Submission Date:	26/04/2018
Version:	v2.0

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 730897.

COVER AND CONTROL PAGE OF DOCUMENT	
Project Acronym:	**HPC-Europa3**
Project Full Name:	Transnational Access Programme for a Pan-European Network of HPC Research Infrastructures and Laboratories for scientific computing
Deliverable No.:	D12.1
Document name:	Container-as-a-service analysis report
Nature (R, P, D, O):	R
Dissemination Level (PU, PP, RE, CO):	PU
Version:	V2.0
Actual Submission Date:	26/04/2018
Author, Institution: E-Mail:	Niall Wilson, Irish Centre for High End Computing niall.wilson@ichec.ie
Other contributors	Raül Sirvent, Marta Renato, Oleksandr Rudyy: BSC Giuseppa Muscianisi: CINECA Yonny Cardenas: CNRS Risto Laurikainen, Ari-Matti Saren: CSC Dimitris Dellis: GRNET

ABSTRACT:
The main goal of this report is to evaluate the use of container technology in order to streamline the deployment of HPC end-user applications. Each application to be deployed typically has a number of dependencies in terms of supported versions of the OS, compilers and linked libraries, which make the maintenance of the different installations complex and error prone on typical HPC systems. The implementation of a lightweight virtualization strategy can strongly improve both the management of the HPC systems and the portability of end-user applications to different HPC centres. This report reviews the considerations necessary and the current status of some of the competing implementations and proposes a suitable architecture relevant to HPC environments.

KEYWORD LIST:
High Performance Computing, HPC, Virtualization, Containers, Docker, Singularity

MODIFICATION CONTROL			
Version	**Date**	**Status**	**Author**
0.1	02/03/2018	Template	PMT
1.0	25/04/2018	Complete Draft	Niall Wilson
2.0	27/04/2018	Final version	PMT

TABLE OF CONTENTS

1 Executive summary

Packaging and running applications via containers has the potential to enable extreme mobility in computational modelling and form an integral part of the effort to enable transnational access to HPC resources and collaboration in research. It offers a method of simplifying and standardising the building and execution of applications on diverse hardware platforms without compromising on performance so that researchers can develop applications on their laptop or local HPC system and easily and quickly get that application running on different HPC system in other institutions or countries. An important additional benefit of this is that containerisation can also effectively capture and preserve the exact tools and environment in which the results of an analysis or simulation were derived and so help to advance the implementation of Open Science policies.

This deliverable reports on the evaluation of different container/virtualization technologies in terms of supported features, known advantages and drawbacks, together with the level of community acceptance. Integration with higher level batch scheduling and workload orchestration is reviewed from the point of view of integration within existing HPC centres as well as the ability to implement pipeline workflows using containers. Finally, a recommended architecture and workflow is presented which captures the requirements of many typical use cases along with advice on where alternative strategies and additional research may be appropriate.

2 Introduction

2.1 Problem Statement

The past 10 years have seen a large expansion in the use of High Performance Computing (HPC) as a tool to enable greater accuracy and throughput of simulation and data analysis across an expanding range of disciplines[1]. This has resulted in a corresponding increase in the numbers of users and application codes running on HPC platforms. Whereas once, HPC systems were predominantly used for executing numerical models and simulations for research in a small subset of the physical sciences, it is now widely used across most disciplines including the life sciences for large scale data analysis as well as modelling. Increasingly, HPC is just one of many tools researchers use as part of their research. Hence deep expertise in traditional HPC skills is not a core competence for many researchers and HPC is gradually being viewed from a software-as-a-service viewpoint.

A consequence of this shift in usage is that the number and complexity of software applications using HPC resources has greatly increased. Similarly, research output is increasingly dependent on the results of software applications which in effect become vital parts of any results of simulation or numerical analysis. If research is to be replicated, validated and built on by other researchers (a key objective of the European Open Science Cloud[2]), it must be possible to exactly reproduce the same results at a later date on different computer systems. However, each application depends on hundreds of other independent packages (from programming language, through support libraries and modules and down through the operating system stack) which each have independent release and update histories. Each individual HPC system will most likely be different in terms of the versions of software installed and will itself change over time. These differences present researchers and HPC centres with a number of challenges in being able to treat applications as independent and deterministic components of research:

- How can applications be built in a reproducible manner?
- How can an application be run in a reproducible manner (i.e. with the exact same software stack)?
- How can we be sure that the application will continue to work with future base OS updates?
- How can we enable the same application to run on multiple HPC systems easily?
- How can we enable users to develop applications on their laptop and then run them without significant modification on HPC resources?
- How can the application be archived such that simulation results can be corroborated and reproduced by others at some point in the future?
- How can multiple versions of the same application be maintained at an HPC site?
- How to enable easier deployments for HPC centres by wrapping up all environment settings, etc in a single application “executable”?

As with any set of requirements, a corresponding set of constraints exists within which a solution must operate:

- The same levels of performance in terms of time to solution must be preserved as would be the case with building and running applications in traditional HPC environments.
- Most HPC systems operate as a shared, multi-user environment and so any new methodology must accommodate the requirements of supporting diverse user requirements for building and running applications.
- As a shared resource, security is very important to protect the integrity of the system as a whole and the applications and data of individual users.

- HPC systems typically are composed of some exotic hardware and software such as parallel file systems, GPU accelerators, RDMA networks, etc. The software must be able to fully exploit the capabilities of these components.
- HPC centre staff must be able to support and assist users with their applications and so the simplicity and ease of use of any proposed solution is important.

2.2 Virtualization

From a strictly functional point of view the requirements above could be met through traditional virtualization technology whereby a snapshot of an application, dependency software and operating system can run on virtual machines (VM) which are in turn run on top of various physical servers. Virtualization can be implemented in various ways but since roughly 2006 most commodity x86, x86-64 and ARMv7 microprocessors from Intel, ARM and AMD offer a hardware-assisted virtualization through special CPU instructions[3]. Thus, the problem of preserving and migrating applications to different platforms could be achieved by installing them in virtual machines, exporting a VM image and then booting that VM on whatever virtualization infrastructure is available.

However, this approach presents a number of disadvantages in the context of HPC applications:

- Image sizes can be very large as they include a full bootable OS.
- Performance is non-optimal due in part to additional overhead of context switching and memory copying involved in running a second kernel inside the VM.
- Access to HPC specific hardware such as InfiniBand networks or GPUs is non-trivial.
- Interaction with high performance parallel file systems such as GPFS or Lustre requires additional site-specific configuration which reduces portability.
- The vast majority of HPC systems do not support running VMs.

2.3 OS-Virtualization

OS level virtualization (known as “Containers” in Linux or LXC) is a technology which enables many of the advantages of virtualization but does so in a more lightweight and efficient way which is much more suitable to HPC applications and systems. It uses a shared kernel across the host and guests and as it does not virtualize the hardware devices, speed and efficiency are improved. Due to this shared kernel, the guest is limited to an identical OS kernel as the host, although all user space software can be unique and even an entirely different Linux distribution can be used within the container. Start-up times for containers are very quick (less than one second) compared to VMs, and in principle the performance achievable is the same or very close to running applications on “bare metal” servers with no virtualization layer.

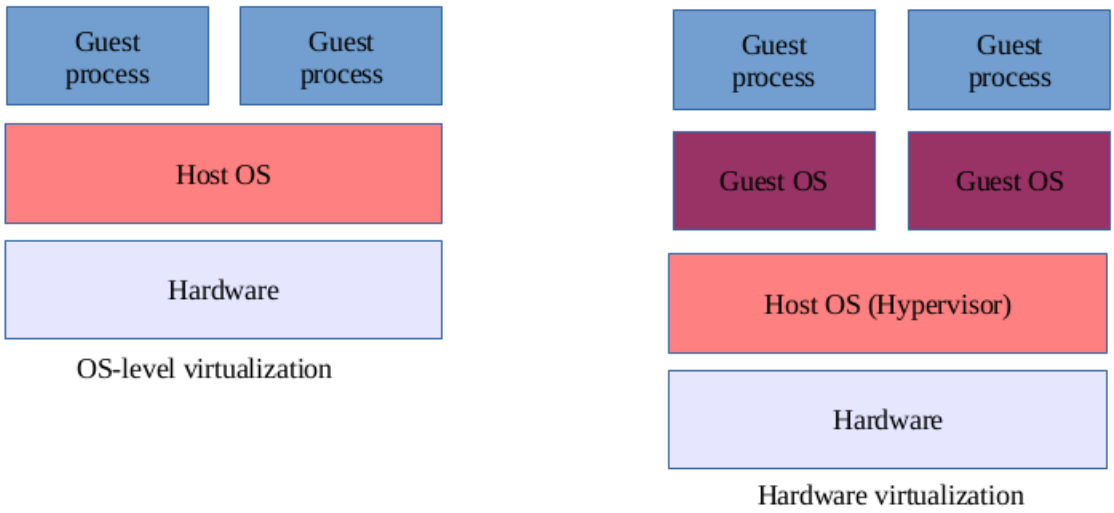

Figure 1: OS-Level vs Full Hardware Virtualization

OS-virtualization is not new and has been implemented in many different systems such as FreeBSD Jails (2000), Linux OpenVZ (2005), Solaris Zones (2004), and AIX Workload Partitions (2007). However, it is only in the past number of years that the set of kernel capabilities for controlling resources and namespaces, management tools and standard file formats have emerged in the Linux ecosystem in order to enable its widespread adoption. In large part, these developments have been led by large cloud scale organisations aiming to streamline the management of web services. Google began developing cgroups (control groups) in 2007 which enabled strict control of resource usage by processes (and hence containers) and so made possible the use of containers as multi-tenant lightweight VMs sharing a common host. But it was Docker (see Section 3.1), originally released in 2013, which led to a huge increase in the use of containers across the IT spectrum. Docker combined cgroups and kernel namespace isolation features with low level LXC utilities to provide an easy to use and comprehensive toolset for creating and running Linux containers as well as an image format to preserve and distribute these container images. Since then, various alternative container management systems have emerged, often targeted at specific requirements (such as rkt[4] for micro-services or Singularity for HPC).

In subsequent chapters we will examine in detail the features of various container technologies. However, based on what container technology can provide in principle, it can be seen that this technology can help to solve the problems of enabling static packaging of HPC applications which can be migrated easily between systems with negligible performance impact. In particular:

- The use of specification files (e.g. Dockerfiles) allow application images to be built through a procedural script which takes a base minimal OS and installs specified packages and executes specific commands
- This image can then be copied to any system which supports the container runtime technology and executed
- These images can be maintained under version control along with various metadata, thus enabling the preservation and documented evolution of the application independently of external dependencies.

2.4 Containers as a Service

While OS level virtualization, i.e. Containers, offer a promising solution for encapsulating an application and its software dependencies, it only solves a part of the problem of enabling better application migration and preservation. To be widely adopted and facilitate change for the better, a whole ecosystem of tools and services

must be provided to enable an effective workflow for researchers. Apart from the core runtime technology, a crucial component of this workflow will be the ability to publish application container images on central repositories and optionally enable the wider community to search and re-use these images. The ultimate aim should be for applications to be the primary concern with a rich set of metadata associated with them and the HPC platform offering containers as a service becoming less visible or unique. Researchers would build their applications on their laptops or search a central image repository for the required container image, pull it down to the HPC system along with their input data and run it with a minimum of site specific modifications.

2.5 Topics and structure of the remainder of the document

This document focuses on the evaluation of different container technologies in terms of supported features, advantages and drawbacks from the perspective of HPC requirements. In particular, aspects such as security considerations in a multi-user environment, potential performance limitations, integration with existing HPC platforms and technologies and management features are examined. To be widely adopted, these technologies must also be easy to use and must facilitate access to persistent storage for input and output datasets as well as to enable workflows. It must be possible to launch multiple containers in concert so that they can make use of parallel computation frameworks such as MPI and it must be possible to sequence tasks such that the output generated by one is used as the input data for the next.

In today's non uniform memory hardware architectures, the correct CPU placement of containers and processes within that container is important for optimal performance. The level of support for enabling this will be analysed also. We will also examine the tools available for managing the lifecycle of persistent container images via build systems and repositories.

In the final section, a generalised container-as-a-service architecture will be outlined which enables the easy creation, migration and execution of applications for the end-user. The aim will be to provide guidance to HPC sites enabling a service like this on the most appropriate tools and technologies to be used depending on requirements.

3 Overview of Current Container Technology

Containers are a feature of the Operating System (i.e. the Linux kernel in most cases relevant to HPC today). However, a large number of choices and options exist in how these kernel features are used and managed in practice to provide a user facing service. These choices manifest themselves in terms of features, security, flexibility, performance and usability. Various alternative software projects have built tools to create, execute and generally manage the lifecycle of containers and a subset of these is presented below. For the purposes of the descriptions below it is important to distinguish between containers and images. *Images* are a set of data (files) from which containers can be created and so multiple, independent containers can be created from the same image. *Containers* are running instances of images along with any modifications made since they were created and started. Containers can be stopped and this state will be preserved or they can be deleted in which case the per-container data is deleted but the image from which it was created still exists.

3.1 Docker

The name "Docker" is almost synonymous with containers in the IT world as it was the software tool which accompanied the mass adoption of this technology. Docker[5] was released as open source in March 2013 after being initially developed as an internal project within dotCloud, a platform-as-a-service company and is now maintained by Docker Inc. Docker is largely used as shorthand for Docker Engine which is the software which manages the creation and execution of containers. A number of additional tools, including Docker Compose and Docker Swarm, build higher level multi-container. Clustered services are also developed by Docker Inc.

3.1.1 Architecture overview and workflow

Docker uses a client-server architecture. The Docker *client* talks to the Docker *daemon*, which does the heavy lifting of building, running, and distributing the Docker containers. The Docker client and daemon *can* run on the same system, or you can connect a Docker client to a remote Docker daemon. The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface. The Docker daemon must run as user root in order to be able to perform its functions of creating and managing containers and associated network and storage resources. As a consequence, the Docker client and hence end user effectively has root permissions on the server node and this can be an unacceptable security issue at most HPC sites.

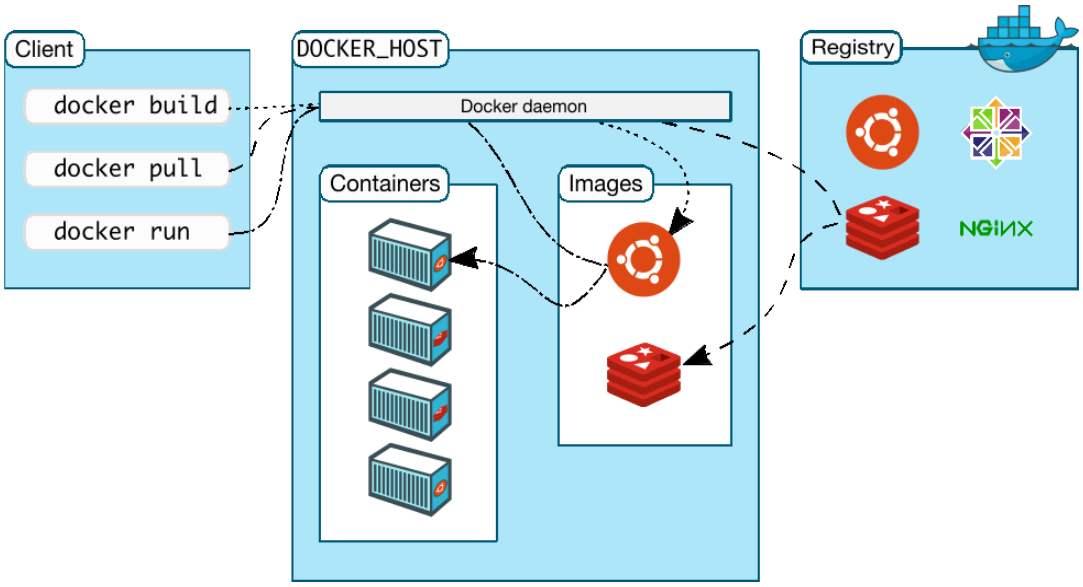

Figure 2: Docker components and architecture

An important and distinguishing aspect of Docker concerns how containers and images are stored on disk. Fundamental to this is the concept of layers. A Docker image is built up from a series of layers stacked on top of each other and each layer is only a set of differences from the layer before it. When you create a new container, you add a new writable layer on top of the underlying read-only layers of the image. This layer is often called the "container layer". All changes made to the running container, such as writing new files, modifying existing files, and deleting files, are written to this thin writable container layer. Thus, multiple containers can be created using the same underlying image but yet each have their own independent state. This is implemented using a Copy on Write (CoW) strategy whereby files are copied into the container layer only when they are written to and thus decrease space utilisation. Numerous Docker storage driver choices are available which implement CoW via a Union mount filesystem service such as aufs, overlayfs, device mapper, btrfs. The different storage drivers are only supported on certain Linux distributions (eg aufs only on Debian or Ubuntu, btrfs only on SUSE, etc) and each has different performance characteristics. Additionally, each has different requirements for the backend storage device (ext4 or xfs filesystem for aufs, overlayfs, direct LVM for device mapper). Currently no storage driver exists for parallel file systems such as Lustre and so a requirement for using Docker is that each node has a local disk on which the images are stored. Furthermore, a mechanism must be put in place to enable users to copy images to the local disk of every compute node and the space on these disks must be managed to ensure they do not fill.

The Docker daemon by default manages the network settings for the containers it creates and acts as a bridge interface back to the host. For parallel jobs involving multiple container instances, this presents an additional challenge in which participating Docker daemons will all have to be configured into a virtual cluster possibly using an overlay network.

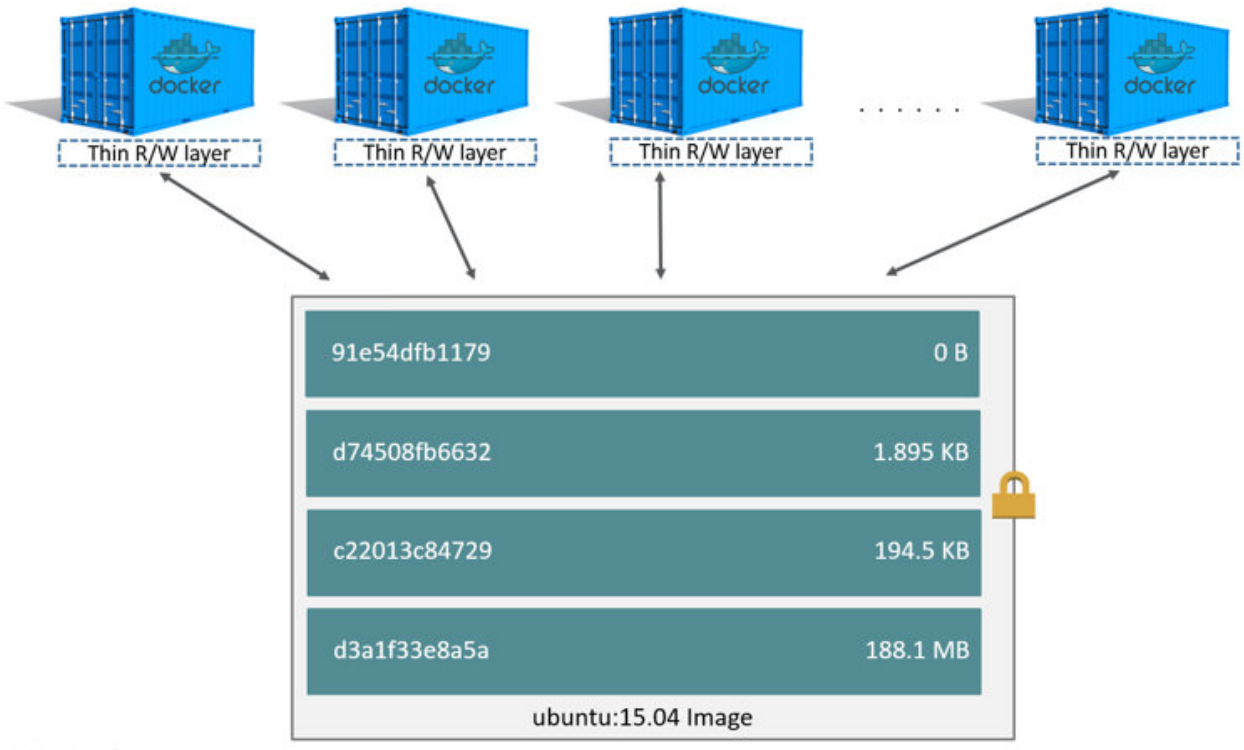

Figure 3: Docker image layers

3.1.2 Integration with GPUs

Docker containers are platform-agnostic, but also hardware-agnostic. This presents a problem when using specialized hardware such as NVIDIA GPUs which require kernel modules and user-level libraries to operate. As a result, Docker does not natively support NVIDIA GPUs within containers.

One of the early work-arounds to this problem was to fully install the NVIDIA drivers inside the container and map in the character devices corresponding to the NVIDIA GPUs (e.g. /dev/nvidia0) on launch. This solution is brittle because the version of the host driver must exactly match the version of the driver installed in the container. This requirement drastically reduced the portability of these early containers, undermining one of Docker's more important features.

To enable portability in Docker with images that leverage NVIDIA GPUs, NVIDIA developed nvidia-docker[6], an open-source project hosted on Github that provides the two critical components needed for portable GPU-based containers:

1. Driver-agnostic CUDA images; and
2. a Docker command line wrapper that mounts the user mode components of the driver and the GPUs (character devices) into the container at launch.

nvidia-docker is essentially a wrapper around the docker command that transparently provisions a container with the necessary components to execute code on the GPU.

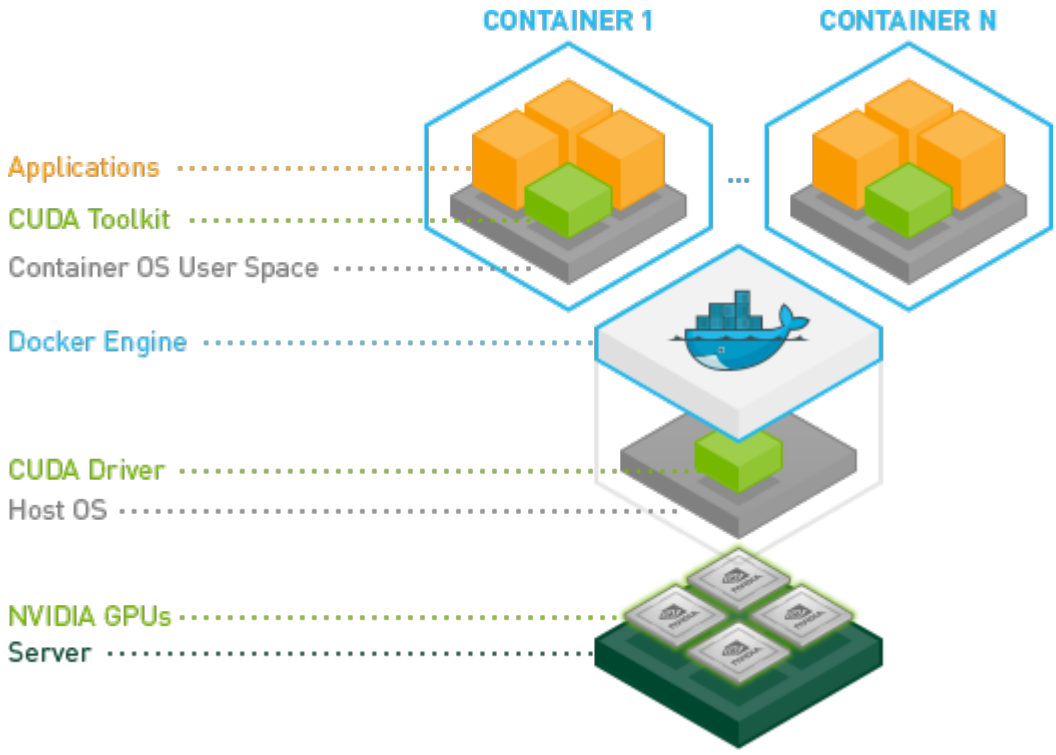

Figure 4: nvidia-docker architecture

3.2 Shifter

Initially developed as a private User Defined Image (UDI) technology by NERSC (National Energy Research Scientific Computing Center) and then released as open source in December 2015, Shifter focuses on providing a fitting implementation of Linux containers for HPC applications and centres.

Currently Shifter is a prototype implementation with seven main goals[7]:

- Scalability and performance of running applications.
- Scalability and performance during the running/building process of the container/image.
- Accessibility to host's features: shared resources, parallel file systems, interconnection networks.
- Compatibility with batch system resource management.
- Adaptability with Docker as well as other container/image formats.
- Full leverage with Docker images and Docker push/pull functionality.
- Robust, secure implementation.

3.2.1 Architecture overview

Regarding its architecture, we can discern four principal actors in Shifter: the ImageGateway, the command-line utilities, the *udiRoot* and the Workload Manager[8].

Due to the rich ecosystem of Docker images available through DockerHub and Docker's simplicity when building them, Shifter leverages Docker utilities instead of implementing its own image workflow. The ImageGateway is used in order to achieve Docker's "pull" functionality, but without the need of a Docker deployment within the HPC system host.

The ImageGateway service provides the user with the login and pull commands, the same as in Docker interface, and then performs the user request, establishing a connection with DockerHub (unless the requested image is located in the host's cache) pulling the image and converting it to a common format used by Shifter (usually squashfs). Once the pulling is completed, the new image is stored in the Shifter image cache keeping the same name as in DockerHub. All these steps are automatic after the user makes the request.

The command-line utilities are "shifterimg" and "shifter" which allow users pull and launch containers via the Shell.

Currently Shifter only provides this capability with Docker repositories, although at NERSC it is also implemented with private repositories and therefore in the future there could be releases offering more alternatives for storing and sharing images.

Before running an image, two requirements have to be fulfilled: Shifter must be installed on all cluster nodes and every node must have access to Shifter's images (for example to the ImageGateway cache folder through a GPFS file system). Root permissions are only required during the installation of Shifter on each node.

When launching a container, Shifter needs to check if the selected image is compatible with the requested features. After the validation Shifter is ready to set up the virtual system by mounting the file system via the loop block device and disabling setUID capabilities to avoid root escalations within the container while keeping the same user id as outside. During the mount process you can also bind-mount host directories and thus get access to your personal data, a feature that is very practical when working on various nodes which have a parallel file system. The steps described are performed according to *udiRoot.conf* configuration file which the administrators can modify to obtain the desired container environment. The container's creation and deletion are run using the udiRoot set of components which are invoked manually or via the Workload Manager.

Finally, in order to simplify and ease application launching some Workload Manager (e.g., Slurm) have been integrated with Shifter. As a result, the scheduler will manage the udiRoot functions as well as the host's resources allocation (cgroups) when submitting the jobs.

3.2.2 Shifter Usage Workflow

The end user should only need to follow a few simple steps combining Docker and Shifter functionalities:

1. Using Docker, the user should build and test their container image with all the required software stack and at the end push the image to DockerHub. This step is performed on the user's laptop.
2. Inside the cluster, with a properly configured Workload Manager, the user has to submit their job specifying the Docker tag of the image, the amount of resources they want to use and extra configuration if the scheduler supports it (MPI ABI compatibility, CUDA drivers, etc.).

The above scenario is the ideal process and the one used at NERSC (see Figure 5), however, it is also possible to run Shifter without Workload Manager integration.

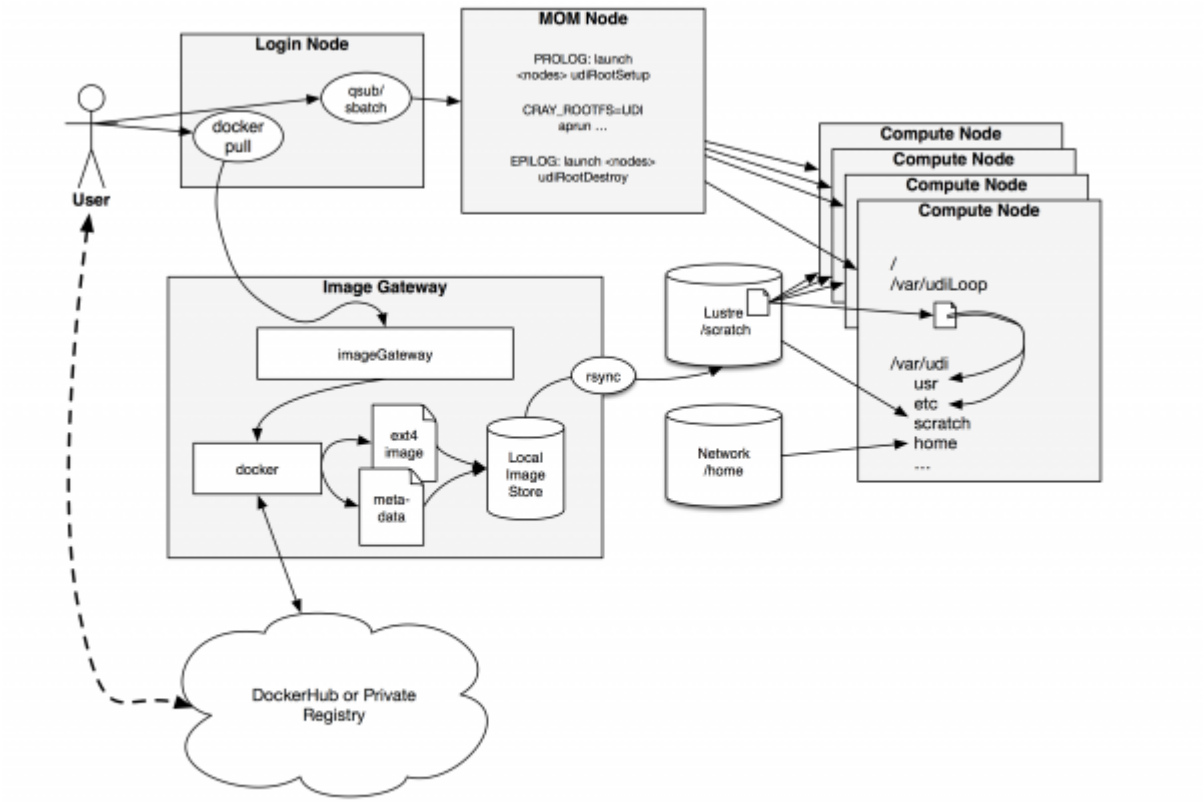

Figure 5: Shifter usage workflow with Workload Manager. Source: https://www.nersc.gov/research-and-development/user-defined-images/

Without a scheduler that does all the work, launching applications with Shifter becomes a bit more complex, as it can be seen in Figure 6. The first step of creating and pushing the image to DockerHub does not change, but the second does as the user is responsible for configuring the parameters and submitting the container to all nodes, e.g.:

mpirun <mpirun options> -hostfile myhostfile shifter <shifter options> --image=<Image tag> command arguments

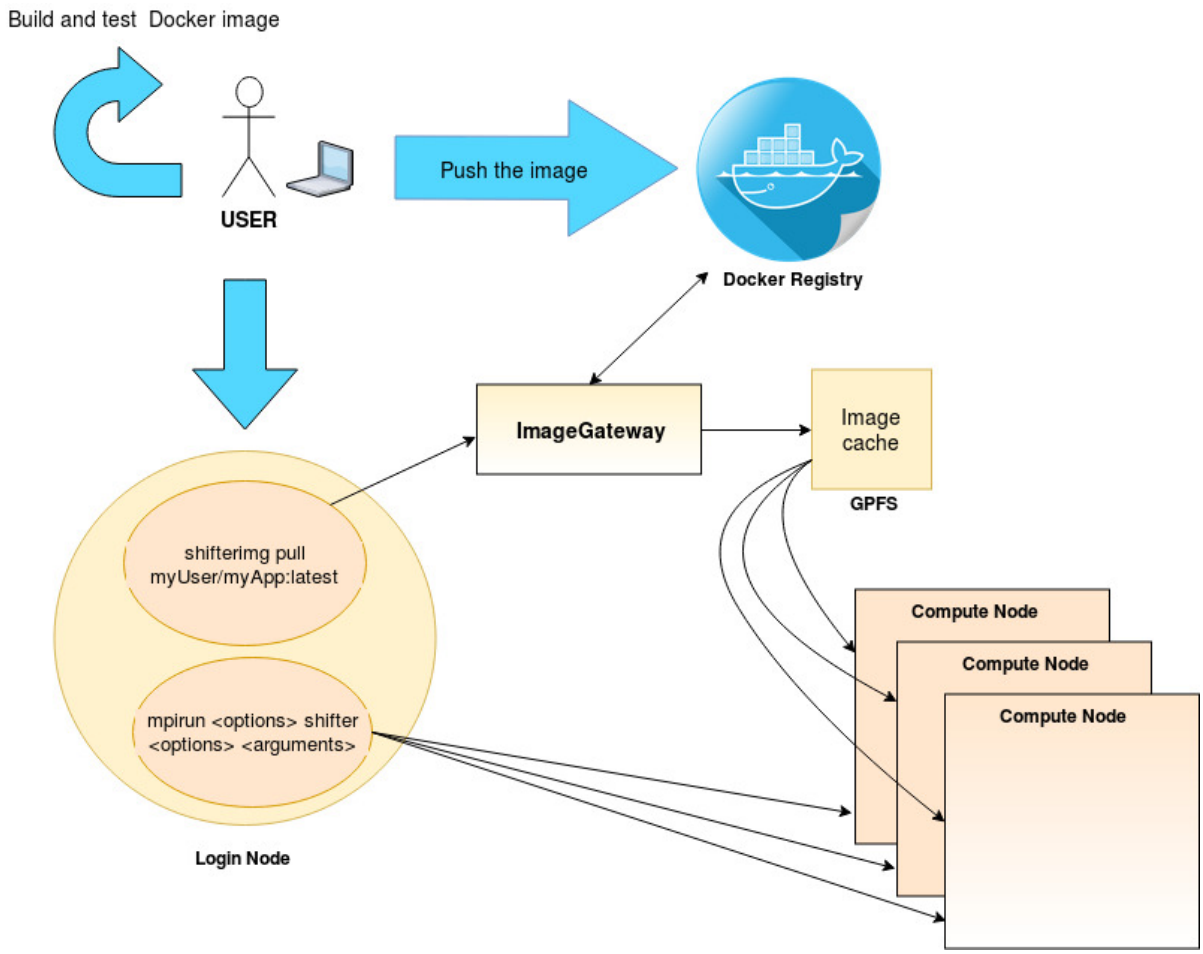

Figure 6: Shifter usage workflow without Workload Manager

3.2.3 MPI and GPU compatibility

The details involved in supporting CUDA runtime environment for GPU devices and the numerous MPI implementations can add some complexity to using containers. For instance, Singularity containers need to have ABI compatible versions of MPI inside and outside the container, a requirement that limits portability among different HPC centres.

In order to make containers full portable, Shifter also uses the ABI compatibility between different MPI implementations. With a proper configuration, Shifter is able to switch container's MPI libraries with the host ones at runtime taking advantage of specific hardware acceleration. In the case of CUDA, Shifter leverages its version compatibility and, as with MPI, provides the capacity to bind-mount host's specific libraries granting to the container transparent access to special hardware.

Nevertheless, to exploit this feature a site-specific Shifter configuration must be provided by the administrator, who needs to modify the udiRoot.conf file with the paths and environment variables of all the site-specific resources. If not defined, Shifter containers use the host's MPI and thus the application inside should use the same version.

3.2.4 Shifter early experience and observations

Despite the fact that Shifter works well once it is installed and configured, it presents some considerable problems during its set up that have to be kept in mind.

Firstly, we noticed Shifter's documentation is scarce and very focused on Cray systems. Centres like NERSC and CSCS have published papers and tutorials about Shifter in HPC where they show how they use it and describe its functionality, which are very useful, but they do not document the installation and configuration process to get a suitable Shifter deployment on different environments. Moreover, it is difficult to find which are all the commands and options of Shifter's command line tool even in Shifter's main Github repository, where we found some sections pending to be finished.

Secondly, the installation was not easy due to problems with the ImageGateway configuration and some dependencies. In particular with *json-c* and *libcurl*, which were not detected by Shifter binaries and we had to install them manually from their sources as well as reconfigure Shifter as follows:

./configure --prefix=/opt/shifter/udiRoot --sysconfdir=/etc/shifter --with-json-c=/usr/local --with-libcurl --with-munge LDFLAGS=-Wl,-rpath,/usr/local/lib

Then, in order to leverage Docker functionalities Shifter requires Mongo and Redis database with Celery job queue. These extra software packages entail a more involved installation procedure. Other installation problems we found are:

- After downloading the repository, when doing "make distclean" it deletes vital files for the compilation.
- The compilation implies downloading some source files from the internet, so it cannot be done natively on nodes which are not connected to the internet.
- Shifter does not allow any user with *sudo* permissions to execute it. We were suggested to modify the source code of the project to bypass this restriction.

Finally, we noticed a pair of runtime issues: *shifterimg* command fails sometimes when trying to pull an image from DockerHub without reporting the cause; Shifter does not provide commands to erase nor rename downloaded images, thus it has a lack of functionalities that should be solved.

3.3 Singularity

Initiated in 2015 by Gregory M. Kurtzer from Lawrence Berkeley Lab, Singularity[9] aims to be a container technology specifically designed to run containerized applications in High Performance Computing environments.

Within Singularity, three specific goals are achieved:

1. "Mobility of compute", defined as the ability to define, create and maintain a workflow, being confident that such workflow could be executed on different platforms.
2. "Reproducibility". Once a contained workflow has been defined, the container image can be snapshotted, archived and locked down such that it can be used later, being confident that the code within the container has not changed.
3. "User freedom". A user can freely choose what operating system, libraries, tools, applications, versions and so on to put into the container to create their workflow without altering the host system in any way.

To reach such goals, Singularity developers have designed the container itself as a single image file easily transferable among different machines. The entire runtime environment (an application, plus all its dependencies, libraries and other binaries, and configuration files needed to run it) are packaged together in a

single file image. In addition, the Singularity containers can interact with files local to the container in the directory where the container is executed.

Another important feature of Singularity that made it a container technology suitable for an HPC environment, is described by "A user inside a Singularity container is the same user outside the container". This means that Singularity limits the ability of a user to escalate permission inside a container, and thus makes it possible to run user supplied containers on an HPC system.

Singularity can convert existing Docker images to Singularity images or can build a native singularity image from scratch using a Singularityfile (a text file containing sequential build instructions similar to a Dockerfile).

3.3.1 Architecture overview

To understand the architecture of Singularity, a comparative image can be used (Figure 7) where a Virtual Machine (VM), a Docker container and a Singularity container are compared, showing the architectural design of each of them.

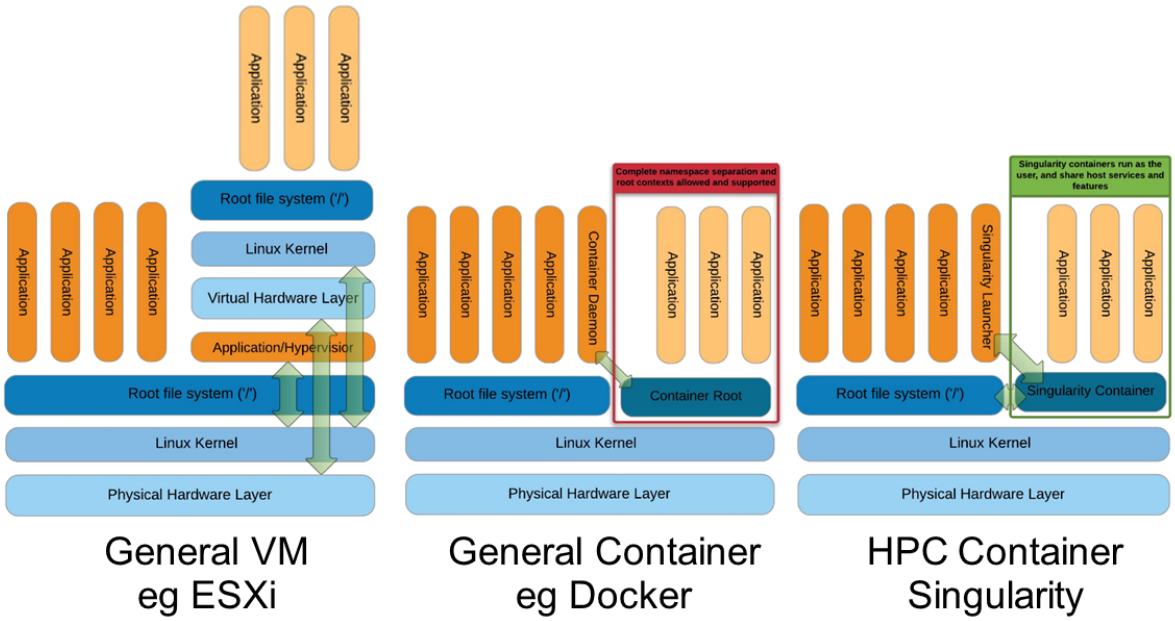

Figure 7: Comparison between a virtual machine, a docker container and a singularity container. Source: Greg Kurtzer keynote at HPC Advisory Council 2017 @ Stanford

Different from a VM, a Singularity container is an example of OS-virtualization and, different from a Docker container, there is no root daemon process running on the host that can be allowed an escalation of privileges within the container. A singularity container runs as the calling user in the appropriate process context. The "Singularity Launcher" is run by the user, loads the Singularity container and executes applications as the user.

3.3.2 Singularity usage workflow

The Singularity workflow derives directly from the Singularity security model in which a user has the same UID inside the container as outside and hence has the exact same permissions. Generally, the main actions that can be performed on a container are the building of the container image and the running of the container. Since the building operations require, generally, root administrative privileges, the user has to made these actions on a system (server, workstation, laptop, and so on) on which they have root privileges. Once the container has been built with the necessary applications, libraries and data inside, it can be easily shared to other hosts and executed without requiring root access.

This workflow is described in Figure 8 - on the left side, the build environment is shown, while on the right side we can see the runtime environment.

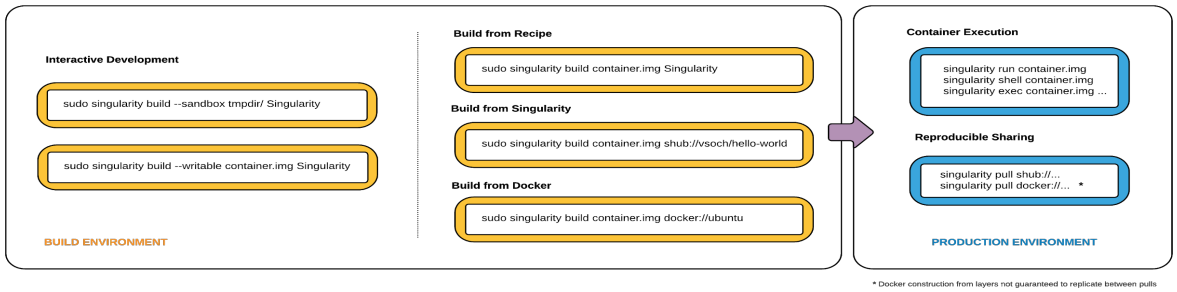

Figure 8: Singularity usage workflow. Source: http://singularity.lbl.gov/

A production container should be an immutable object, so if you need to make changes to your container you should go back to your build system with root privileges, rebuild the container with the necessary changes, and then re-upload the container to the production system where you wish to run it.

3.3.3 MPI and GPU compatibility

Since Singularity containers are designed for HPC platforms, it is possible in a very simple way to run pure MPI or hybrid (MPI + OpenMP) applications inside a Singularity container. Also, GPU support is available in these containers.

3.3.3.1 Integration with OpenMPI

Singularity containers can be created for any application and are not just limited to HPC applications. However, a very important feature of Singularity is the possibility to create simple containers for parallel and hybrid applications. Singularity treats the applications like an executable, so the "mpirun" command treats it like any other executable.

The OpenMPI/Singularity workflow works as follows:

1. mpirun is called by the resource manager or the user directly from a shell.
2. OpenMPI then calls the process management daemon (ORTED).
3. The ORTED process launches the Singularity container requested by the mpirun command.
4. Singularity builds the container and namespace environment.
5. Singularity then launches the MPI application within the container.
6. The MPI application launches and loads the OpenMPI libraries.
7. The OpenMPI libraries connect back to the ORTED process via the Process Management Interface (PMI).
8. At this point, the processes within the container run as they would normally directly on the host.

This entire process happens behind the scenes, and from the user's perspective running via MPI is as simple as just calling mpirun on the host as they would normally do.

3.3.3.2 Integration with GPU

Since Singularity 2.4.x version, it is possible to run a GPU application by adding the option "--nv" in the Singularity command. This option automatically binds host system driver libraries into the container at runtime, avoiding the installation of a GPU driver and libraries inside the container.

3.4 Charliecloud

Charliecloud [10] is a recent (2017) project started in Los Alamos National Lab which aims to enable standard Docker containers to be executed on systems without requiring any privileged daemons or operations. It aims at simplicity and providing the minimum required to run containers as a non-privileged user on typical batch oriented HPC systems. Due to this simplicity (the source code is just over 800 lines of C and shell), Charliecloud provides a useful insight to how containers are implemented in Linux.

3.4.1 Namespaces

The Linux kernel provides six namespaces which provide isolation of kernel resources between processes. Using these, a process and its children can be isolated and will not be able to see or access resources and attributes of other processes on the system and thus provides the fundamental mechanism which enables containers. Namespaces are always active and can be nested. There are five privileged namespaces and a single one unprivileged. The privileged namespaces can only be created by root and are:

1. Mount: filesystem mountpoints.
2. PID: process ID. A PID inside a namespace will have a different value when viewed globally from outside the namespace.
3. UTS: (Unix Time-sharing System) - host and domain names.
4. Network: network interfaces, ports, routing tables, etc.
5. IPC: inter-process communication resources such as pipes, shared memory, etc.

The unprivileged namespace is:

1. User: grant unprivileged processes access to privileged functionality in specific contexts where safe to do so. This namespace is the most recent and was added in kernel 3.8.

The first process in a user namespace has all capabilities in that namespace but none in the parent namespace. Additionally, a UID and GID mapping exists between the namespaces such that for example, the regular unprivileged user who created the namespace can be mapped to UID 0 (root) within the namespace. Charliecloud uses this user namespace plus the mount namespace as the minimal set necessary to implement containers by bind mounting a directory into a user namespace where the unprivileged owner effectively has root permission within that container but any system calls which are not isolated to the container are mapped back to the caller UID so that the container cannot perform privileged operations on the host.

In contrast to some other container implementations, the other namespaces are not used either because the features are not required or because there are functionality and performance advantages to not using them in the context of HPC:

- The PID namespace is not used because there is no need to hide container PIDs in a typical HPC environment.
- The UTS and network namespaces are not used because there is a distinct advantage in terms of convenience and performance for containers to be able to directly use the host network stack rather than configuring another layer of indirection.
- The IPC namespace is not used which allows the host and different containers on that host to use various IPC methods to communicate which can improve performance and simplify workflows.

3.4.2 cgroups

Control groups (cgroups) are a Linux kernel featureset which account for, limit usage and control access to resources (CPU, memory, network, etc) for a set of processes. Originally developed by engineers at Google in 2006, they have been in the main Linux kernel since version 2.6.24 (2008). Cgroups enable processes (and

hence containers) to be pinned to specific cores and NUMA memory locations which is a useful and necessary feature for enabling maximum performance in HPC environments. Some container runtime technologies incorporate cgroup functionality for added control of launching containers but Charliecloud made a conscious decision not to do so and to instead rely on external, orthogonal toolsets to perform this role. The flexibility and control of this approach can be a distinct advantage for HPC applications which require fine grained control over process placement and can be incorporated into launching containers via batch scheduling systems.

3.4.3 Workflow

Because Charliecloud uses Docker images, users can use the Docker toolset to create their application images on their own laptop, or they can use the ch-build wrapper script to build an image using an installed docker engine with a specified Dockerfile. This image is then exported to a single tar file which can be copied to an HPC cluster. At runtime, the Charliecloud provided ch-tar2dir wrapper script untars this file to a specified location and the ch-run command then creates a container with this directory as the root mountpoint.

3.5 HPC Specific Factors

In the context of executing applications via containers, a number of factors unique to HPC need to be addressed. These include the ubiquitous use of the Message Passing Interface (MPI) standard as a means of parallelising codes, the use of specialised hardware such as GPUs and the widespread use of parallel filesystems. Here we will examine the challenges each of these factors presents as well as how the alternative technologies address them where relevant.

3.5.1 Message Passing Interface (MPI)

MPI is a standard API commonly used in parallel HPC applications to pass data between processes and control the concurrent execution of many independent tasks. Numerous implementations of the published standard exist such as Intel MPI, Cray MPI, MVAPICH, OpenMPI, each providing a different set of runtime libraries and configuration options. Also, the MPI standard doesn't specify how processes are launched and so many alternative ways exist to initiate an MPI application ranging from simple shell invocation via ssh or the batch system to the use of dedicated daemons. Thus, building and running an MPI application can be quite complex and vary from one system to the next. When containerising an MPI application we first need to consider how it can be made to work in terms of the correct startup and execution of all containers involved and secondly how to make it as portable as possible so that it can run on many systems with minimal (if any) changes.

There are generally two ways to implement MPI from within a container:

1. The entire MPI stack is contained within the container. This is the model which is most commonly used and can usually be made to work with all technologies. At first glance it appears to offer the advantage of fully encapsulating the environment, however critical information such as how network addressing is set up on the host cluster and specific information on features and tunings of the HPC network is dependent on the system where the containers are going to be run. This significantly impacts the portability and potential performance of the container.
2. The MPI is split partially between the host and the container. This is the preferable approach in that the MPI within the container does not have to be built specifically for a target host or resource but simply requires a compatible version to be present on the host cluster. It also alleviates much of the networking complexities as the MPI processes in the containers can be started through the batch system or local MPI startup method. In this scenario for example, the batch system could call the regular system installed mpiexec which will then launch the containers using the set of nodes allocated to it.

Recent versions of Singularity and OpenMPI have been modified to work well together (as does Shifter and Cray MPI). Singularity detects when an OpenMPI application is being built and automatically adds all library dependencies into the image.

Additional factors to consider are how MPI processes communicate between containers within the same node. Depending on which namespaces are in use (i.e. the IPC namespace), the optimal use of shared memory for communications may not be available in which case the network must be used which will increase message passing latency.

3.5.2 GPUs and CUDA

Driven by the insatiable market demand for real-time, high-definition 3D graphics, the programmable Graphic Processor Unit or GPU has evolved into a highly parallel, multithreaded, manycore processor with tremendous computational horsepower and very high memory bandwidth. More specifically, the GPU is especially well-suited to address problems that can be expressed as data-parallel computations - the same program is executed on many data elements in parallel - with high arithmetic intensity - the ratio of arithmetic operations to memory operations. In November 2006, NVIDIA introduced CUDA[11], a general purpose parallel computing platform and programming model that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. NVIDIA CUDA is the dominant programming framework, but any language that allows the code running on the CPU to poll a GPU shader for return values, can create a GPGPU (General-purpose computing on graphics) framework. Other common programming standards for GPU parallel computing include OpenCL[12] (vendor-independent) and OpenACC[13].

To enable a container to run a CUDA application using the host GPU, there are two approaches which can be used:

1. Install GPU drivers and CUDA libraries within the container. The disadvantage here is that driver versions must match on the host and container.
2. Use a GPU enabled container technology. The nvidia-docker wrapper mentioned in an earlier section creates Docker containers with a CUDA runtime which can interoperate with the host GPU drivers thereby eliminating the need to explicitly match versions on the host and image. Singularity in version 2.4 and above enables the use of the host GPU drivers and CUDA libraries to be automatically bind mounted in a container at runtime thus eliminating the need to install them in the container image.

3.5.3 Parallel Filesystems

Parallel cluster file systems such as Lustre, BeeGFS and IBM Spectrum Scale (GPFS) are a key component of almost all HPC systems and enable a consistent view of a shared persistent file system on all nodes with high performance being a primary design principle. All of the container technologies presented above allow containers to bind mount a directory structure from the host into the runtime container and so applications can use this feature to read and write data to this filesystem without requiring specific software to be installed. Thus, for example, a user's home directory on Lustre can be mounted at runtime in a container and the application can read its input data from there and write output as the job proceeds.

Differences in the use of the Network Namespace between Docker and the other container implementations suggest that there may be a performance overhead on I/O for Docker and this is one area which will be investigated in the Benchmarking subtask of the JRA.

It has already been noted that the layered UnionFS image format of Docker is not compatible with parallel file systems and so native Docker images must be stored on a separate ext4 filesystem, possibly duplicated on each node. The other technologies all use flat images which are essentially a single large file in the case of Singularity or a tar file of a directory structure which is subsequently extracted in the case of Charliecloud. The single large image file used by Singularity is particularly well suited to parallel file systems which are typically optimised for this type of file. Also, for applications such as Python workflows which involve accessing a very large number of small module files, having the container image as a single file enables much higher performance in terms of startup time because the metadata servers are not overwhelmed by many nodes simultaneously performing metadata lookups on hundreds of small files. Benchmarks[14] have shown that as

the number of concurrent Python interpreters increases with the size of a job, the startup time increases due to file system contention but that this can be offset by using Singularity containers to reduce the number of file accesses.

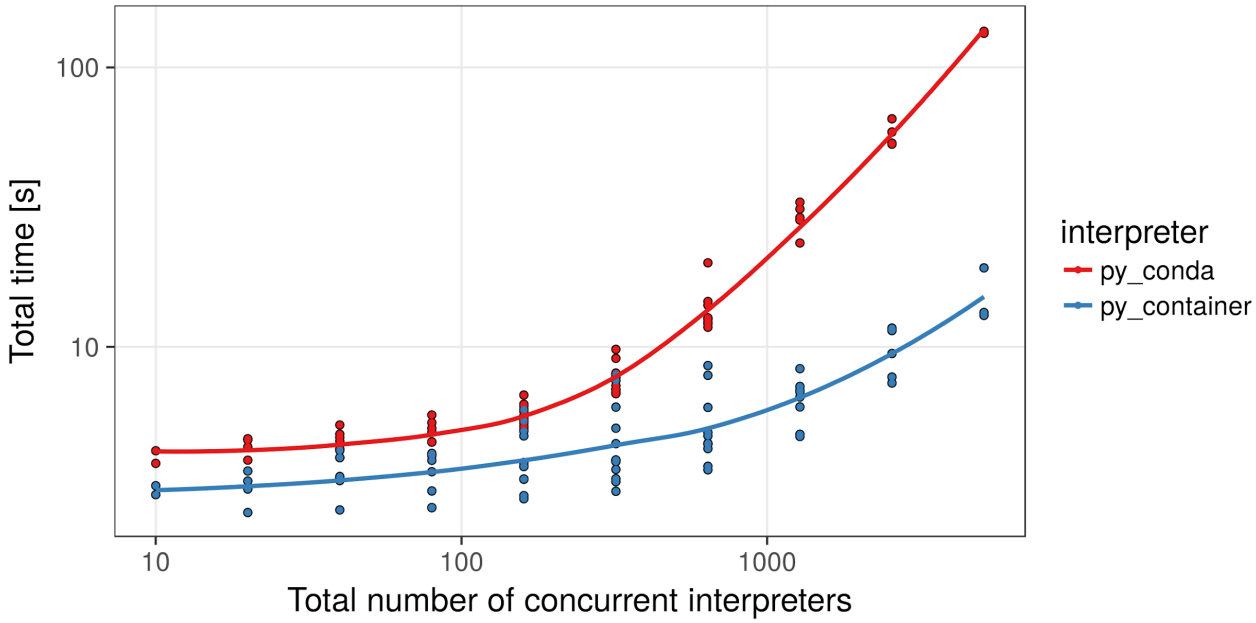

Figure 9: Startup time of python script on container vs regular filesystem install

3.6 Security Factors

The vast majority of HPC platforms are centralised shared resources where many disparate users are granted access to execute applications and store data. Some of these users may be working together on common projects and so share datasets and applications. Most HPC systems must also be accessible over the Internet to enable remote connections and so this relatively open access model combined with potentially large numbers of users of the system means that security is a big concern for most HPC centres (this environment differs from the typical cloud environment in which Docker was developed as a means of deploying web applications). Measures must be taken to ensure that regular users cannot gain unauthorised access to resources or data or cause disruption to other users. In practice this requires that any software vulnerabilities involving privilege escalation be eliminated as much as possible to ensure that regular user accounts cannot gain root access.

A comprehensive overview of container security is beyond the scope of this document but the interested reader can refer to an excellent whitepaper[20] by NCC Group for an in-depth treatment of the topic. However, at a high level, the technical risks to be evaluated include:

1. **Guest breakout**, in which vulnerabilities in the implementation of the container runtime results in the unintended ability of local users or processes within the container to access privilege levels or resources on the underlying host.
2. **Container management**, whereby the tools and methods involved to start, stop and otherwise manage the running of containers can result in privilege escalation.
3. **Cross-container attacks** are most relevant where multiple users will have containers running on the same physical host. Weak networking configurations in particular could allow malicious attacks on containers from other co-hosted containers involving e.g. eavesdropping, address spoofing.
4. **Container images** themselves could host insecure versions of software with known vulnerabilities. Here, the valid desire to have a static, self-contained application container is in conflict with the equal

desire to patch security bugs. Tools exist which can scan container images for known CVEs and it is best practice to build images using the minimal software stack necessary.

Because containers are a kernel feature, the major risk of guest breakout largely comes down to the security of the Linux kernel and there have been a lot of security bugs in the kernel over the years and continuing up to the present. For example, Docker, Singularity and Charliecloud all rely on the User Namespace feature (depending on version and configuration) to prevent root access to the host but any errors in the implementation or use of this feature can and have resulted in privilege escalation bugs. If container technology is to be widely adopted within HPC centres, then an active approach of ensuring kernel security patches are always up to date is required. This requirement is complicated by the fact that other vital HPC components such as RDMA interconnects, parallel filesystems and GPU devices also require kernel modules and these may not all be supported on the latest releases of the Linux kernel.

As was mentioned in the previous specific sections on Docker, Singularity, Shifter and Charliecloud, the latter three projects were started in large part due to the unacceptable security risks of using Docker on HPC platforms. The client-server architecture of Docker plus the fact that the Docker engine must run as root effectively gives any user with client access to the Docker daemon root access to that node. With parallel file systems ubiquitous on HPC systems this would in turn grant any user full read/write access to all data on the system. Docker was not designed for the environment and workflow common to HPC systems and so its security model is not fit for purpose. Some projects[21] seek to modify Docker to make it more secure and better integrated with batch scheduling systems, however the momentum is firmly behind projects such as Singularity which take the approach of a bespoke and simpler implementation of containers which focus on a security model and workflow more fitting with a HPC environment.

3.7 Feature Comparison

The table below summarises some of the differences and similarities between the container technologies discussed above. It is a high-level view and many aspects have not been included, however it does illustrate a significant difference between the HPC focused tools and the more enterprise focused Docker.

	Docker	Singularity	Shifter	Charliecloud
Target audience	Developers, DevOps, Cloud Native	HPC / Scientific Application end Users		
Use case	Microservices, independent web services	Encapsulation of simulation/analysis applications with difficult software stacks. Application packaging for portability and archival.		
Execution model	Persistent daemon controls container execution, etc. Managed by local or remote client	Simple executable which launches container specified by user	Simple executable which launches container specified by user but also requires a central ImageGateway	Simple executable which launches container specified by user
Image format	Layered Docker images	Single, flat (non-layered) Singularity Image. Can export Docker images to Singularity images	ImageGateway pulls images from Docker Hub and converts to Shifter images (squashfs)	Uses flattened (tar format) Docker images
Namespace isolation	By Default, share	Option to use PID and	Optional. Default is to	User Namespace isolation

	little. Process, Network, User space are isolated by default, but can be shared via CLI arg	User Namespace but mostly share Namespace with host. This enables easy and performant access to network and filesystems.	share Namespace.	only.
cgroup resource restrictions	inbuilt via CLI args: docker run [--cgroup-parent, --cpuset-cpus, --cpuset-mems, etc]	Must be manually specified or managed via the batch scheduler.		
Network access	Uses network namespace. NAT via bridge interface default.	Transparent access to host network interfaces, routing tables, etc		
InfiniBand/OmniPath	Works via IPoIB. Direct RDMA requires heavily customising Docker network settings and installing special software in container image.	Transparent access to RDMA devices on host.		

3.8 Initial Tests and Comparisons

During the course of this task, we put a lot of effort into installing the different container technologies on various clusters for the purposes of familiarisation and evaluation of how they fit into a HPC environment. The first test and comparison documented below is a proof of concept test to run an MPI benchmark across multiple nodes. We then look a little bit deeper and examine how workload colocation could be accommodated. These tests are intended only to examine and demonstrate the feasibility of running HPC applications using containers, with performance considerations coming later in subsequent tasks (Task 12.5 - Benchmarking).

3.8.1 Running Basic MPI Applications

For the purpose of testing the behaviour plus limitations an app may have when executed within a container we performed a simple test using Docker, Singularity and Shifter implementations. We selected a very simple hybrid microbenchmark which can be run with multiple configurations (OpenMP, OmpSs, MPI+OpenMP and MPI+OmpSs) to study how parallel applications may behave with containers and verify that BSC's DLB (Dynamic Load Balancing) library is able to handle load imbalances from inside.

Testbed used

Lenox cluster (courtesy of Lenovo)

- MPI Version: OpenMPI 1.10.4
- 4 compute nodes with: 2 sockets Intel® Xeon® Processor E5-2697 v3 CPU with 14 cores and 28 Hw Threads each @ 2.60GHz, for a total of 56 threads per node

The versions of Docker, Singularity and Shifter we used are 1.11.1, 2.4.5 and 16.08.3 respectively.

Firstly, we built a Docker container image with all the needed software stack to run and analyse properly the execution flow of our application. Inside the image we ran Ubuntu 14.04 with the following software:

- Open MPI 1.10.4
- OpenSSH server
- DLB
- BSC's Nanos++ runtime
- BSC's source to source compiler Mercurium
- BSC's tracing tool Extrae
- The benchmark to be tested.

Once our image was functional we pushed it to DockerHub because we wanted to use the same image with the three container implementations.

We decided to launch our tests 2 times with Docker, Singularity and Shifter, one without enabling DLB and one enabling it, using 2 nodes with 2 MPI processes each node and 4 threads per MPI process, so in total we were running 8 threads of our benchmark on each node.

In the Singularity and Shifter cases, when submitting the jobs were just invoked from the login node *mpirun* specifying a hostfile and desired bindings, treating containers as wrappers of our program. With Docker, however, we deployed 2 containers as virtual nodes connected through an overlay network and ran our tests from inside since Docker's aim is to make containers as isolated and autonomous as possible.

In Figure 10 we can see the execution flow of the same executable with Docker, Singularity and Shifter respectively. In red it is represented each OmpSs' task execution (the parallel code) and in pink all the sequential code (the code outside parallel regions) that each MPI process performs, which consists of MPI calls and the sequential code of our microbenchmark. We can also observe that the execution presents a considerable load imbalance between MPI processes 1.1, 1.2 and 1.3, 1.4 because there are threads doing nothing useful, which is represented in black since they are idle, between pink or red regions.

In essence the executable behaves the same with all three implementations, nevertheless it is evident that Docker takes some more time to end than Singularity or Shifter. Considering the fact that Docker containers operate using an overlay network it makes sense there exists extra overhead because of MPI communications.

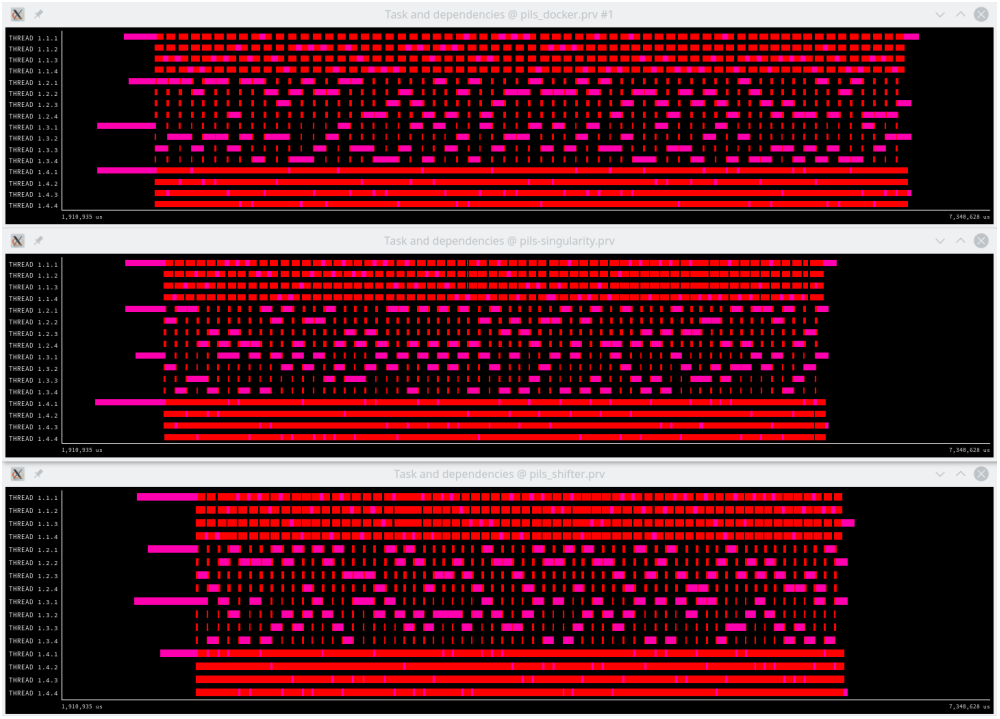

Figure 10: Paraver trace without DLB using Docker (top), Singularity (middle), Shifter (bottom)

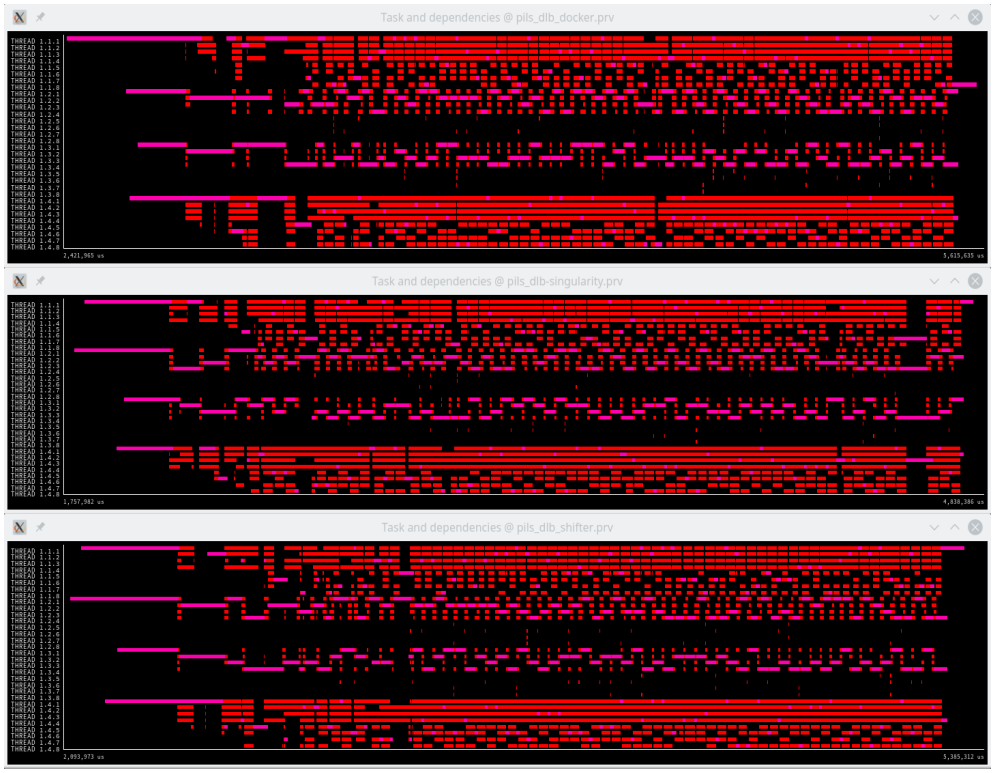

Figure 11: Paraver trace with DLB using Docker (top), Singularity (middle), Shifter (bottom)

In Figure 11 we expose the flow our application with dynamic load balancing. The three traces (Docker, Singularity and Shifter) are very similar and we can clearly appreciate the effect of enabling DLB library. Besides, each MPI process now has 8 threads instead of two. The application possesses the same amount of

hardware resources as in Figure 10, but with DLB each process is able to borrow the assigned CPUs of the other MPI process in the same node, so at the end one MPI process is able to potentially run on 8 CPUs and therefore it needs 8 threads now. If we take a closer look (Figure 12) it seems clearer how when the less loaded MPI processes 1.2 and 1.3 finish their work, instead of blocking themselves and going idle their CPUs DLB assigns that resources to the other two processes which still have computations to do. As a result of a more efficient hardware usage because DLB avoids having CPUs in idle state, we were able to get a speedup of around 30% with respect to the same execution but without DLB enabled presented above.

In short, with these tests we could answer our initial questions about how an application may work within containers. Our final conclusions are:

1. It is viable to launch MPI applications with Docker, Singularity and Shifter.
2. The application presents very similar behaviour with Docker, Singularity and Shifter, thus it is logical to believe that these three implementations do not interact significantly with the application's execution.
3. Docker brings some overhead which may be due to the network implementation.
4. Applications seem to be easily scalable with Singularity and Shifter, with Docker however we would need a more complex deployment involving explicit network and MPI setup and configuration.

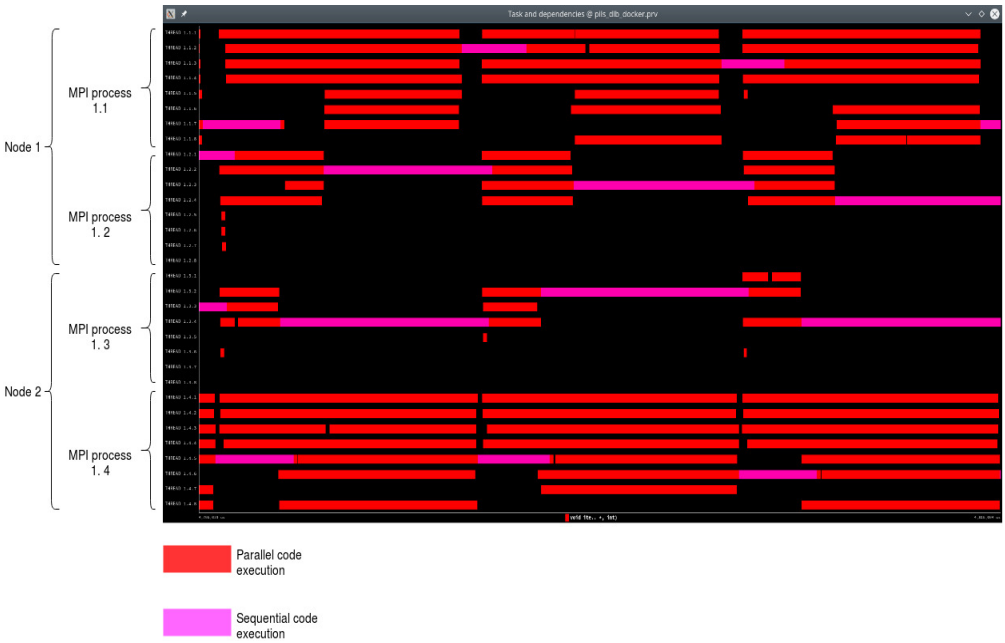

Figure 12: DLB effect

5. It is possible to take advantage of DLB library without special integration.

3.8.2 Workload co-location

The objective of Task 12.4 in the JRA will be to explore node co-location of applications. Having more than one application executed in a single node is very useful from the system administrator point of view, since it can easily increase the utilisation of the system. It also makes a lot of sense that applications are able to share their resources within a node if we consider current trends in computer architecture, where single nodes continue to increase in core count (either CPUs only or mixed with GPUs), a hierarchy of memories available (e.g. Non-volatile Memories, NVM), different network interfaces, and so on. With so many resources available in a single node, it is possible that a single application is unable to exploit them all, which leaves a lot of unused resources in the supercomputer. Therefore, sharing resources between applications becomes desirable.

However, if two applications run in the same node without any kind of coordination or control, they can find themselves fighting for some of the available resources, which may end up resulting in severe performance degradations for both. This means that we need to ensure the isolation between applications to avoid these interferences, and container technologies today look promising to achieve such a goal.

In order to be able to optimise resource utilisation in a node, applications must be malleable, i.e. able to change the number of resources they use during runtime. Examples of malleable applications include those developed with OpenMP or OmpSs[15] programming models (to exploit intra-node parallelism), but also with MPI (inter-node parallelism). In particular, we are interested in what are called hybrid applications, that exploit both intra and inter-node levels of parallelism at the same time.

Having all this in mind, one of the technical requirements we need from the container technology we have to select is that it enables to adapt resources for each container depending on their demands. As a first approach, we will focus on the CPU utilisation metric to determine how many CPU resources are assigned to each container running in the same node.

BSC provides a tool called Dynamic Load Balancing library (DLB)[16] to enable applications share resources in the same node. We need to see if the same ideas implemented in DLB can be also directly applied to containers, or adaptations are needed. DLB is able to share resources between different applications running in the same node, or even in the same application to speed up the processing. It is a library linked with the application that is able to lend the unused CPUs from an application to another one running in the same node, therefore speeding up the execution of the second one. DLB keeps a fair control of resources, returning them to the original owner when they are needed. Besides, DLB is able to interface with the job scheduler with its DROM API, to exchange knowledge about resource needs and resources available.

We consider two container technologies as candidates to deal with the isolation problem mentioned before: Singularity and Docker. They are the most commonly used frameworks nowadays. At this initial stage, we have been running compatibility tests of containers with hybrid applications (MPI+OpenMP or MPI+OmpSs) using DLB to share resources between them. So, for each container technology, we have installed it in a testing environment and in the MareNostrum 4 supercomputer (whenever possible) to see if there are any incompatibilities between containers, the programming models used, the DLB library, the benchmarks used, and the machines. Last but not least, we also considered and tested Shifter.

We need to distinguish compatibility tests (look for incompatibilities between the different technologies), with benchmark tests. The formers are useful for discarding incompatible technologies, and are specifically useful for Task 12.1, while the latter are the ones related to Task 12.5 and their goal is to numerically evaluate the performance of the different solutions proposed.

The next sections give more insights on the compatibility tests, while providing some details about pros and cons for each container technology when trying to run co-location of applications. We hope our experience can help data centre or supercomputers administrators to determine what is the best solution if they plan to enable co-location of workloads in their systems.

3.8.2.1 Workload co-location in Docker

Docker is the most common containerization technology, even though it was not designed for HPC. Its design of being a system container, and thus, having each instance of an image acting as a complete virtual node of a cluster, allows workload co-location of any application executed to be trivially managed inside the container as it would be on bare metal. But it's not as easily orchestrated when trying to do so amongst different containers on the same node.

Base container image construction

In order to be able to execute and measure our experiments, we created a base container with several BSC tools, that container would be used afterwards as the base for the containerization of each applications. Those tools are:

- OpenMPI: Needed in order to compile the binaries inside the container. Its version changed in order to match each testbed OpenMPI installation.
- Nanos++[17]: Runtime of OmpSs, needed for compilation.
- Mercurium[18]: BSC's source to source compiler used for a pre-compilation of Ompss applications.
- Extrae[17,19]: BSC's tool used to track events from application executions.
- DLB: BSC's library for Dynamic Load Balancing.

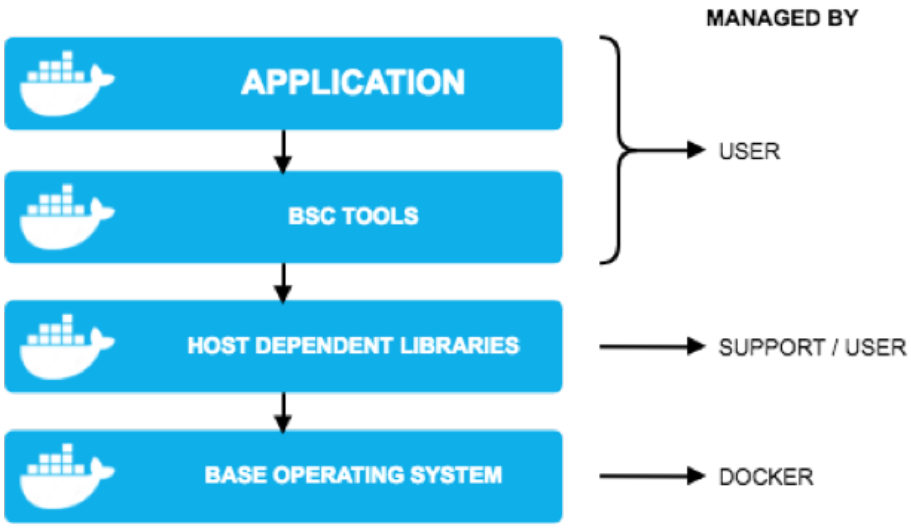

Figure 13: Container image software stack used for Docker tests

The idea is to structure the containers to take advantage of Docker's stacked structure in order to be able to reuse the already built image for all future applications.

Experiment 1: MPI tests

As Docker is not designed for HPC, we first needed to check the feasibility of executing MPI applications among Docker containers and its capabilities on managing workload co-location. To test this, we used a simple "hello world" program written in MPI just to verify the connectivity between containers. T his is executed using two nodes and one container per node, each one running one instance of MPI. The isolation and resource management were verified using the tool nanox-bindings from the Nanox runtime.

As already stated, isolation is trivially satisfied inside a single container. On the other hand, when managing isolation between containers running in the same node we are able to run them with well-defined resource allocation thanks to different Docker options, for instance *--cpuset-cpus* which bind our container to a set of CPUs. However, enabling these kind of options limits our container to use only its assigned resources and no more during execution. Thus, some dynamic workload managers would not be able to work properly among multiples containers.

Experiment 2: MPI + OmpSs tests

Our next experiment was testing the execution and performance of hybrid applications. The selected testing applications were the PILS and BT-MZ benchmarks, and they were executed with different configurations of granularity, from 1 MPI per node and 56 OmpSs threads to 56 MPIs per node and 1 OmpSs thread. As with MPI alone, isolation was trivially satisfied with a single container, and can be easily managed between two containers.

Experiment 3: MPI + OmpSs + DLB test

The last step was to perform experiments with hybrid applications and DLB in order to ensure the correct behaviour of the library and the capabilities of Docker on dynamically managing workload co-location. The selected applications for this test were the same benchmarks as for Experiment 2, recompiled in order to make them work with the DLB library. In this case, we could verify DLB's behaviour inside a single container but not between different containers, which still has to be tested more in depth in Task 12.4.

Conclusions on Docker test

Even though the technology is widely used in cloud computing, and workload co-location is easily managed inside a single Docker container, we still have not been able to use DLB for the management of multiple containers. This can be a huge drawback for it, as it impacts directly on the execution management of different applications on the same node. More research will need to be carried out during Task 12.4.

3.8.2.2 Workload co-location in Singularity

Singularity is one of the most promising containerization technologies on this aspect. Its nature of being an application container delegates the isolation and resource management of each container to the host operating system, allows us to easily manage it as if it was a bare-metal application. In the specific case of DLB library, once a Singularity container is executed it shares its host shared memory, so it works out-of-the box, since DLB makes usage of the shared memory for internal functionality.

Base container image construction

Given Singularity's feature of being able to build containers from Docker images, we reused the base image created for Docker as a base for our Singularity containers. We did that in order to guarantee coherence between both installations and minimise the necessary modifications to update any of the included tools on both technologies. In conclusion, all images have the following structure:

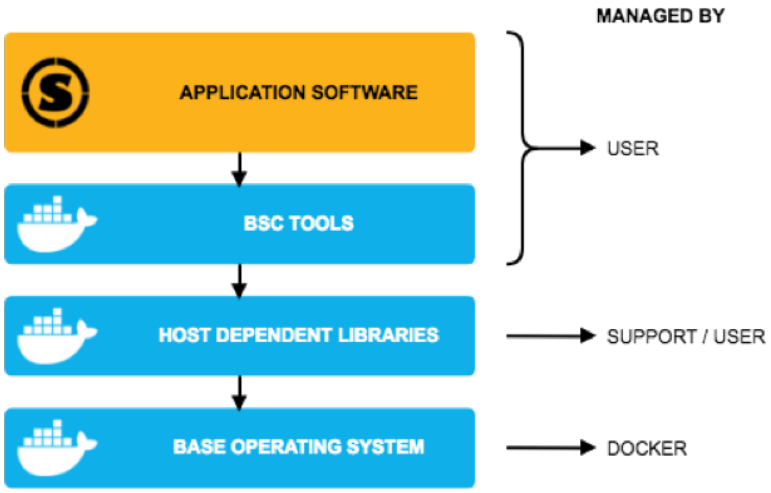

Figure 14: Container image software stack used for Singularity tests

Experiment 1: MPI tests

We started with basic MPI functionality tests in order to ensure we were able to distribute the container execution resources externally. This experiment mimics the first done with Docker, using a basic "hello world" written in MPI and with the only objective of being able to test the connectivity among containers using MPI and the possibility to manage the execution resources externally. This experiment was executed on all testbeds with positive results. Isolation between applications is easily achieved.

Experiment 2: MPI+OmpSs tests

After the basic functionality was checked, our next experiment was testing the execution of hybrid applications. The selected testing applications were PILS and BT-MZ benchmarks, and they were executed with different configurations of granularity. On both benchmarks we were able to manage each process resource access without any serious issue, and confirmed the results of the basic MPI tests.

Experiment 3: MPI+OmpSs+DLB tests

Finally, we checked the behaviour of DLB when working with Singularity containers. The selected applications for this test where the same benchmarks as for Experiment 2, recompiled in order to make them work with the DLB library. We confirmed that DLB was perfectly capable of dynamically managing resource distribution among the different containers depending on their threads status, independently of their internal application.

Conclusions from Singularity tests

Singularity's simplicity is key for easily managing workload co-location, being able to handle containers by treating them as if they were bare-metal applications and working out-of-the-box. The minimal isolation of resources allows for potentially higher performance and dynamic load balancing which will be investigated in later Tasks. Some of the built-in workflow conveniences such as how it integrates with the host's OpenMPI or CUDA installation have the downside of decreasing a container images portability however.

3.8.2.3 Workload co-location in Shifter

Shifter workflow usage is almost identical to Singularity. We treat Shifter as a wrapper of our application so it is enough to run one *mpirun* to start it. As with Singularity, the host's Operating System is responsible for managing Shifter's container resources. DLB is also compatible with this implementation and able to use the shared memory of the host.

Base container image construction

As one of Shifter's aim is to leverage Docker images we just reused our Docker images with zero modifications. The *shifterimg pull* command automatically pulls the image from DockerHub and converts it to Shifter's format.

Experiment 1: MPI tests

As with Docker and Singularity we checked that with Shifter we are able to manage externally container's resources. First, we tested it with a basic MPI "Hello World" and later with a simple shared memory semaphore in order to see whether different containers have connectivity and are able to share host's resources. Shifter passed both tests successfully.

Experiment 2: MPI+OmpSs tests

After our first experiment showed no workload co-location problems, we tested Shifter with PILS using different inputs and allocations, from one node and one thread to multiple nodes with various MPI processes. No problems were found.

Experiment 3: MPI+OmpSs+DLB tests

At last we executed the same PILS benchmark as in experiment 2 and confirmed DLB capacity to manage application's resources as in bare-metal.

Conclusions from Shifter tests

As stated with Singularity, we can deal with Shifter as with a common application, but we have to keep in mind its MPI dependence either taking advantage of Shifter's specific MPI compatibility or directly using the same implementation of the host. Potentially it has higher performance capabilities than Docker.

3.8.2.4 Recommended architecture for workload co-location

Comparing the strengths and weaknesses of each technology, first we can see that Docker is the most portable technology, since it allows us to work on any machine with the same image without any host dependencies. However, it requires a complex setup in order to execute and run containers, once it's configured on a system any application can be easily run without any problems. On the other hand, Singularity and Shifter are easier to use in an HPC environment but this convenience has a drawback in portability (which is not insurmountable but merely the consequence of the ease of use).

Both Singularity and Shifter excel at workload co-location, allowing us to easily manage its assigned resources as if it was a simple binary executing on host. Whereas for Docker it is slightly more complex to manage this due to its goal to virtualize a complete isolated system. This full isolation may also have performance implications yet to be investigated.

4 Overview of Supporting Tools

The focus of the preceding sections was concerned with the runtime of containers. While this is a primary consideration for HPC, it is also critical that application containers can be built, stored and executed efficiently and effectively. Below, we discuss some of the ecosystem and supporting infrastructure necessary to make best use of containers.

4.1 Creating container images

The easy and transparent creation of container images was arguably the most significant advantage Docker introduced over older container technologies such as LXC. Dockerfiles are simple ASCII textfiles which enable the step by step creation of an image starting with a small base image of a Linux distribution and adding package installs, shell commands and environment settings.

4.1.1 General Steps

The image creation workflow can be broken into the following tasks:

1. Selecting the base image to build the application image on
2. Creating and gathering all software artifacts and their dependencies
3. Creating an image build description (a Dockerfile or similar)
4. Building the image, either locally or on a build system
5. Publishing the image in a repository, from where it can be fetched for execution
6. Testing the image

To be able to reproduce the image creation process, the image source files including build description, application source code and binaries have to be versioned. In practice this means that all artefacts should be placed under a version control system such as Git, and care must be taken to have all external dependencies as well as the base image referred to with a version tag.

The build process may be automated by using a continuous integration (CI) process, where new image source code triggers a build and automatic tests against it. The publishing step can be gated so that only image versions that successfully pass testing become available. Popular CI systems include Jenkins, GitLab Runner and Travis.

4.1.2 Singularity

Building a Singularity container requires root privileges on the build system, so on a typical HPC system the users do not have the option of building their own containers. They can import containers they have created e.g. on their own system or on a suitable virtual machine. It is also possible to build Singularity containers in the cloud through Singularity Hub using Singularity recipes.

Singularity images can be built starting from:

- Existing container repository in Singularity Hub
- Existing container repository in Docker Hub
- Existing container on your local machine
- Sandbox directory
- File in .tar or compressed .tar.gz format
- Singularity recipe file

It should be noted that there may be problems when starting with Docker images that were not created read-only because singularity defaults to read-only containers plus is normally executed with non-root privileges and so filesystem permissions for certain directories may not be writable by the user executing the container.

Singularity can produce containers in three different formats:

- Compressed read-only squashfs file system suitable for production
- Writable ext3 file system suitable for interactive development
- Writable (ch)root directory called a sandbox for interactive development

It can also convert containers from one format to another.

It is a best practice to build your immutable production containers directly from a Singularity recipe file. If changes were made to the writable container (ext3 or sandbox) before conversion, there is no record of those changes in the Singularity recipe file rendering your container non-reproducible.

4.2 Storing container images

Container images can be stored in an image registry, which is a type of file server. It allows end users to store images (push) and to retrieve them (pull). There can also be additional features such as the ability to add metadata to the images, access control or security scanning of images.

Image registries can either be public or private. Notable public repositories include the Docker Hub[22], Quay.io[23] and the Red Hat Container Catalog[24]. At a minimum, a public repository allows anyone to pull images. Some allow any user to also share their images with others by pushing them to the registry. Docker Hub is a cloud-based registry service which can store manually uploaded images and link to code repositories, build images and test them after new commits. It is a key component of the Docker ecosystem and provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.

Docker Hub provides the following major features:

1. Image Repositories: find and pull images from community and official libraries, and manage, push to, and pull from private image libraries to which you have access.
2. Automated Builds: automatically create new images when you make changes to a source code repository.
3. Webhooks: a feature of Automated Builds, Webhooks let you trigger actions after a successful push to a source code repository.
4. Organizations: create work groups to manage access to image repositories.

Some image registries specialize in images for specific fields. One example of such a registry is the Singularity Hub[25]. They provide images with scientific software. It is also possible to build a service that provides additional features on top of existing registries: Dockstore[26] gathers images with scientific software from Quay.io and Docker Hub and adds additional metadata and features for the Common Workflow Language (CWL)[27] that is used to describe analysis workflows for various fields of science.

A private registry is a registry that is only accessible to a limited set of users and is typically hosted by an organization for its own use, though there are also options hosted by a third party, such as the Google Container Registry[28]. Docker Inc. distributes an implementation of a Docker registry as a Docker image that can be self-hosted. There are also products such as OpenShift[29], GitLab[30] and Artifactory[31] that include an integrated Docker registry.

4.3 Workload management

Application containers are rarely executed in isolation as a single instance. Mostly, a set of containers must be started in a coordinated fashion because they are all part of a single MPI job or there may be a workflow involving the output data of an application being used as the input for another. These tasks can be managed via specialist workflow managers which are a well developed area of tools.

4.3.1 Batch scheduling systems

Most HPC clusters are managed using batch scheduling systems whereby a user submits a job to a "queue" requesting a specified number and type of node for a certain amount of time. If and when these nodes become available, the batch system takes the queued jobs and starts it on a set of nodes. Containers can add some additional complications to this process such as:

1. Does the container image need to be extracted or moved prior to startup?
2. How is the container started - are daemons involved as is the case with Docker?
3. Are there any considerations required for networking? Again, Docker by default creates and uses its own private network.
4. How are parallel file systems attached to the container?
5. How is resource usage by the container managed? i.e. how are cgroup or cpuset configurations managed?

The difference in target audience of the various technologies shows up strongly here - Docker is not well suited to batch systems but the other more HPC focused platforms are more easily integrated.

Singularity can execute containers like they are native programs or scripts on a host computer and thus integration with schedulers (e.g. PBS Pro, SGE, SLURM) is simple. All standard input, output, error, pipes, IPC, and other communication pathways that locally running programs employ are synchronized with the applications running locally within the container.

Additionally, because Singularity is not emulating a full hardware level virtualization paradigm, there is no need to separate out any sandboxed networks or file systems because there is no concept of user-escalation within a container. Users can run Singularity containers just as they run any other program on the HPC resource.

Singularity can also integrate with host MPI when containers are built to support this. From the user point of view the integration is invisible and Singularity MPI jobs can be run as any other MPI job on the system.

Much of the planned research work for Task 12.2 ("Using containers technologies to improve portability of applications in HPC") will involve investigating the above questions and evaluating the ease of use and effectiveness of the alternative technologies.

4.3.2 Container orchestration systems

Batch scheduling systems schedule jobs with a limited lifetime on a cluster. Each job is put into a queue from which it is scheduled onto one or more servers. The time limit of a job is enforced by the scheduler. With the availability of container technology, the jobs that are scheduled can be in the form of containers that run on the batch scheduling system just like any other binary. This model is good for computations that have a clearly defined beginning and an end.

The container orchestration systems available today such as Kubernetes[32], Docker Swarm[33] and Mesos[34] take a different approach compared to batch scheduling systems for running software. Their purpose is to ensure that an application runs reliably, can scale up if needed and can be accessed by its end users. They are thus more geared towards running long running software with no specific end time, like interactive web

applications. Within scientific computing this means, for example, software like Apache Spark[35], Jupyter Notebooks[36] or APIs that provide some functionality like image recognition or classification.

While it is possible to run a containerized distributed application on multiple hosts without the help of a container orchestration system, this quickly becomes impractical as you have to manage both the application and the lifecycle of the containers. You have to ensure that when a server fails, the containers that were on it are automatically transferred to healthy hosts. You have to ensure that new container instances can be easily created when load on the application increases to a level that can't be covered by its current resources. You need to ensure that data is stored securely and that it can be transferred securely to and from the application. The purpose of container orchestration systems is to provide a common abstraction for these types of problems so that they don't have to be solved by each individual container user separately.

While the interfaces and abstractions provided by different container orchestration systems are different, they share common aspects. For example:

- Connecting multiple hosts to form a cluster on which containers can be scheduled;
- Providing a way to create multiple replicas of a container for horizontal scaling and fault tolerance;
- Some way to connect persistent storage to containers for storing databases and other data;
- Routing of traffic from the Internet to container instances;
- Pipelining the analysis of a dataset through different application containers.

These and other topics will be investigated during the course of Task 12.3 – Orchestration of applications packaged as containers.

5 Architecture Proposal

Container as a service centres around the requirements of users and the applications they wish to use for the purpose of conducting research work. Hence the architecture is primarily one of designing a recommended workflow and suggesting a suite of tools which can work within the constraints of typical HPC centres. A guiding principle of this architecture proposal is that it must be easy to understand and simple to use for researchers and must be straightforward for HPC centres to implement in parallel with their traditional service. Only if these aims are met will such a service have a good chance of being adopted and widely used within the HPC community. Section 5.1 describes this proposed high-level architecture.

Beyond the basic service architecture, we have also considered how it could be enhanced further towards the particular goal of enabling Open Science. If container as a service can make computer applications a more re-usable and reproducible aspect of research work, then a vital next step is to ensure that these application images are findable and sufficiently documented. Section 5.2 describes the considerations and existing models around digital preservation which we propose should be incorporated into the next stage of the service.

5.1 Recommended general architecture & workflow

The main high-level components of the container as a service architecture are as follows:

1. **Application Container Images**

   The enabling feature of application portability is being also to encapsulate all required software into a single portable image.

   This component includes tools to help package applications into images as well as a central repository to enable users to store and retrieve images. Basic authentication and search capabilities are also required. Technology choice here involves Docker and Singularity image formats and it is possible to convert Docker images to Singularity images.

2. **Container runtime technology**

   Different software implementations exist which enable the execution of containers on a Linux system. Each will have different requirements, advantages and disadvantages. The proposed architecture will allow each site or system to provide as many of these as they wish. Local security policies and requirements as well as software dependencies will determine which container runtime software is exposed to the users. Examples of requirements include the ability of the container runtime to interact with tools for workload co-location. Docker is the first technology with widespread adoption but other mature technologies like Singularity may be more suitable for HPC applications and environments, especially due to Docker's strong requirements on running with system privileges, which can jeopardize security in big infrastructures. Other HPC specific variants which can be made available include Shifter and Charliecloud.

3. **Workflow Manager**

   Different applications are frequently part of larger workflows and so containerised applications must be able to integrate with workflow managers which can chain together numerous applications and their corresponding input and output datasets. HPC systems remain predominantly batch oriented. As such, the execution of containerised applications will need to conform to the constraints of existing batch workload managers (eg Slurm, Torque, PBSPro) in a majority of sites. But more container-native orchestration tools such as Kubernetes are also of interest where cloud-like platforms are in use and should be included.

From a user perspective the workflow is as follows (Figure 15):

1. User optionally creates an application container image and pushes it to either a public image repository or directly to a HPC system.
2. User logs onto HPC system and optionally pulls application container image from an image repository if image isn't already present on system. An image repository isn't mandatory - images can be manually copied or created on the system itself also.
3. User writes workflow job which could be as simple as a single application requiring just 1 node or multiple applications, chaining together input and output datasets run across multiple nodes. This job is then submitted to the workload manager.
4. The workflow manager is responsible for allocating the required hardware nodes and initiating the startup of the containerised applications. Data staging between network file systems and local node file systems may be performed here also.

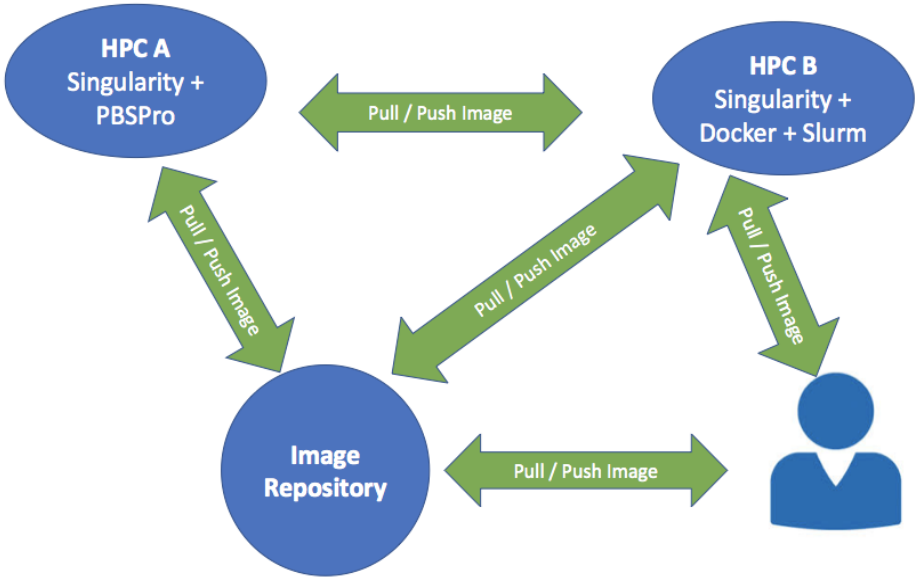

Figure 15: Workflow example from user perspective

Each HPC site or cluster within a particular site is free to support any one or combination of the above technologies. So, a single cluster could support both Docker and Singularity as well as bare-metal jobs which is ideal for benchmarking purposes. Each cluster may also have a different workload manager but it is highly unlikely that multiple options would exist on the same cluster. The requirements and characteristics of each container runtime technology will necessitate different approaches to running applications. For example, Singularity can natively run images from parallel file systems such as Lustre which are common on HPC clusters but this is not true of Docker, and so additional work will be required to stage images into and out of compute node local storage.

So, the proposed architecture to be implemented is:

1. An image repository to contain Docker and/or Singularity images accessible for read/write. This could also be a set of complementary repositories including Docker Hub, Singularity Hub and a dedicated HPC Europa 3 repository. Central to the concept though is that these repositories are available over the Internet to use at any HPC centre.
2. A number of clusters with at least one (and preferably more) container runtime implementations installed with a workload manager, multiple nodes and a parallel file system available to run both native bare-metal and containerised applications on the same hardware.

3. The ability for a user to create an application image locally on their laptop or home HPC system and publish it for use by others or themselves on different remote HPC systems.
4. To run an application, the user logs into a specific HPC system, pulls the application image from the repository and submits a batch job to the workload manager to execute the set of containers required.

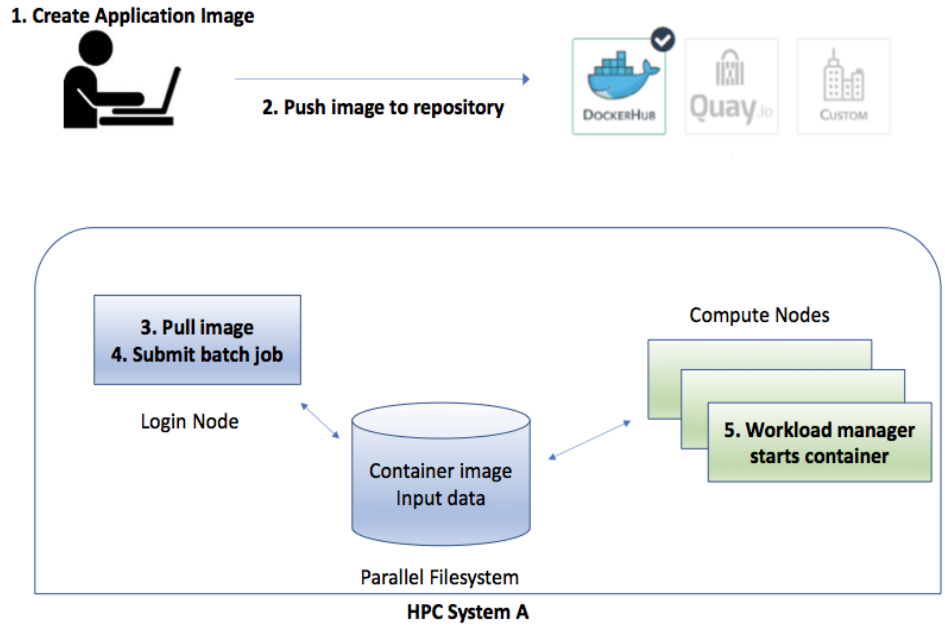

Figure 16: Architecture proposal diagram

5.2 Digital Preservation System

As discussed throughout this document, containerised computer applications fulfil a number of desirable requirements which enable Open Science:

- keeping a history and description of software installation and stack configuration;
- sharing previous work and configurations with other researchers;
- enable easy portability of end-user applications to different HPC systems;
- enable reproducibility of results.

However, creating a container image of an application only solves part of the problem. To fully fulfil the larger aims, it must be possible to publish the images and all associated metadata in a way which makes it easy for other researchers to search for, access and re-use these images.

These requirements can be fulfilled using well established, standard digital preservation procedures to manage information that describes the content of container images. The image registries discussed in Section 4 do have some search facilities and allow for metadata tags but they are quite limited in scope and not designed for supporting scientific research. Implementing a purpose specific OAIS compliant container image repository/catalogue with standards and best practices could greatly enhance the visibility and re-use of application images. At this stage it is not clear how such a system could best be implemented using existing software and specifically if existing container image repository software be extended to support this role. It will remain an open area of investigation for the remaining implementation work of this task as well as Task 12.2 to more fully examine these topics.

Digital preservation

Digital preservation is defined as the series of managed activities necessary to ensure continued access to digital information for as long as necessary[37]. The Open Archival Information System (OAIS) is a reference model[38] used by a wide variety of organizations and institutions with digital preservation needs.

A digital preservation repository seeks to preserve information for access and use by a designated community (e.g. HPC community) which is an identified group of potential consumers who are interested in a particular set of information.

A digital repository/catalogue is a computer system that ingests, stores, manages, preserves, and provides access to digital content for the long-term.

Container images can be managed as digital content in a digital preservation repository/catalogue which ensures the following advantages:

- container images are within the physical control of the repository/catalogue;
- container images can be uniquely and persistently identified and retrieved in the future;
- all information including extended and specialised metadata attributes are available so that the digital content can be understood by a designated community (i.e. HPC community);
- significant characteristics of the container environments are preserved.

Open Archival Information System (OAIS)

A sample digital preservation platform which could be implemented is the OAIS reference model[38]. In the OAIS model, content to be preserved is ingested into the system as Submission Information Packages (SIPs), is stored and maintained over time as Archival Information Packages (AIPs), and is made accessible to users as Dissemination Information Packages (DIPs).

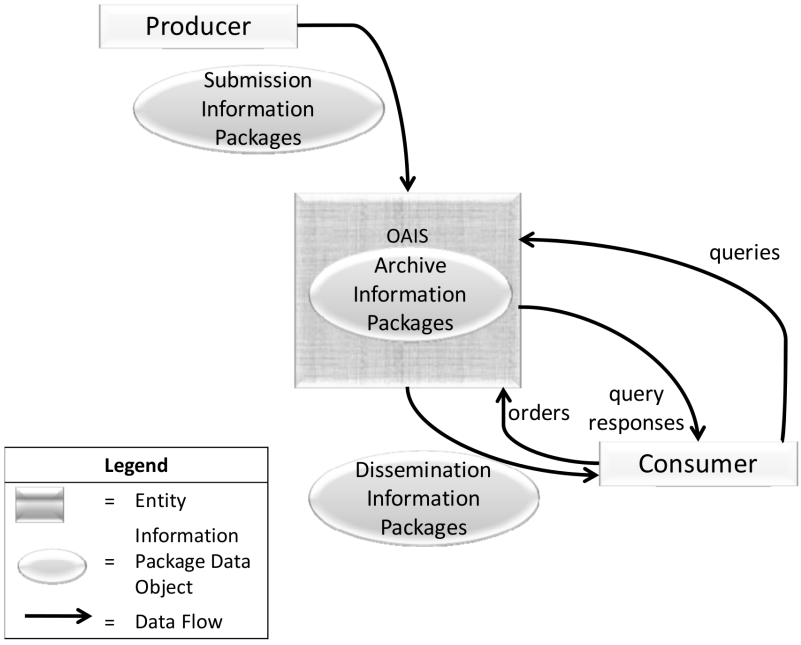

Figure 17: OAIS High Level External Interactions

There are six functional entities in an OAIS[38,39]:

- Ingest function: receives information from producers and packages it for storage. It accepts a SIP, verifies it, creates an AIP from the SIP, and transfers the newly created AIP to archival storage
- Archival Storage function: stores, maintains, and retrieves AIPs. It accepts AIPs submitted from the Ingest function, assigns them to long term storage, migrates AIPs as needed, checks for errors, and provides requested AIPs to the Access function
- Data Management function: coordinates the Descriptive Information of the AIPs and the system information that supports the archive. It maintains the database that contains the archive's information by executing query requests and generating results; generates reports in support of other functions; and updates the database.
- Administration function: manages the daily operations of the archive. This function attains submission agreements from information producers, performs system engineering, audits SIPs to ensure compliance with submission agreements, develops policies and standards. It handles customer service and acts as the interface between Management and the Designated Community in the OAIS environment.
- Preservation Planning function: supports all tasks to keep the archive material accessible and understandable over long terms even if the original computing system becomes obsolete, e.g. development of detailed preservation/migration plans, technology watch, evaluation and risk analysis of content and recommendation of update and migration.
- Access function: This function includes the user interface that allows users to retrieve information from the archive. It generates a DIP from the relevant AIP and delivers it to the customer who has requested the information.

Digital Preservation Metadata

The preservation functions depend on the availability of preservation metadata that describes the digital content in the repository to ensure its long-term accessibility.

Metadata contains records of the context of the digital material. Sharing and preserving the context of digital materials can be just as important as sharing and preserving the actual container applications. The specific metadata needed for long-term preservation falls into four categories based on basic preservation functional groups:

1. **Descriptive metadata** - Describes the intellectual entity through properties, such as author and title, and supports discovery and delivery of digital content. It may also provide an historic context, by, for example, specifying which scientific instrument was the original data (source provenance). An example of a standard schema for descriptive metadata is MODS (Metadata Object Description Schema)[40].
2. **Structural metadata** - Information about internal physical structural relationships, such as which raw data files as well as logical structural relationships, such as which directories. The METS (Metadata Encoding and Transmission Standard)[41] standard and schema includes such metadata.
3. **Technical metadata** - Includes technical information that applies to any file type, such as information about the software and hardware on which the digital object can be rendered or executed, or checksums and digital signatures to ensure fixity and authenticity. It also includes content type-specific technical information, such as software dependencies.
4. **Administrative metadata** - Information about rights management: The information necessary to restrict access to the digital object. It includes provenance information on the creation and subsequent treatment of the digital object, including details of responsibilities for each event in its lifespan. There

are different standard schemas that can specify these metadata. Examples are METS and PREMIS (Preservation Metadata: Implementation Strategies)[42].

5.3 Conclusions

The intervening period between the submission of this Joint Research Activity proposal (March 2016) and the start of work saw rapid change and new developments on the topic. At the time, Docker was the only viable tool for managing containerised applications, but since then a number of specialised HPC centric technologies have emerged and gained rapid adoption. In particular, Singularity has since become the de-facto standard tool at many HPC centres where it is now a standard maintained package available for all users of the system and has significant community support from HPC vendors, application developers and users. As we have discovered throughout this review document, Docker's focus on enterprise web services results in some incompatibilities with typical HPC requirements. The ease of use of Singularity with respect to network, file system and batch system integration as well as its security model and MPI integration make it much easier to deploy alongside traditional application deployments on current HPC systems. Shifter is similar in its approach but is more tightly linked to Cray systems and the requirements of the US national labs it was developed in. Charliecloud is a new minimalist implementation which looks promising as a Singularity alternative albeit with a much reduced featureset. Also, the default configuration of Docker is not optimised for performance and particularly in cases where network communications are important, initial investigations suggest that Singularity, etc will perform better.

The ease-of-use of Singularity does however introduce a slight loss of portability of images in cases where host libraries are introduced as dependencies (e.g. OpenMPI and CUDA integration). This can be offset through the use of standard, neutral base images and published Dockerfiles or Singularityfiles which enable images to be rebuilt from scratch on the target system. Initial tests performed during the course of this analysis exercise demonstrated the ease with which applications can be imported and run on an existing HPC system. For example, we have run a GPU version of Tensorflow (which is notoriously difficult to build) inside an Ubuntu based container on an old SLES11 cluster and packaged this up for end users of the system to run production jobs. In short, our initial recommendations would be that for the majority of cases, Singularity is the best technology to use unless portability is the primary concern and security and performance are secondary, in which case Docker should be considered.

In order to fully realise the potential of creating mobile applications however, it is vital that applications can be published, archived and searched by the community at large. Thus, an important consideration for future work in this area will be the features provided for by the image registries in terms of metadata management and other digital preservation factors.

The second year of this subtask will involve the implementation of a container as a service and will deploy a cluster with Singularity, a local image repository and workload manager. The technical details involved with implementing this will be documented and provided along with user documentation in the final report (Deliverable D12.2).

6 References

1. The Vital Importance of High-Performance Computing to U.S. Competitiveness. Available at: https://itif.org/publications/2016/04/28/vital-importance-high-performance-computing-us-competitiveness. (Accessed: 6th March 2018)
2. Open Science - Research and Innovation - European Commission. (2018). Available at: http://ec.europa.eu/research/openscience/index.cfm?pg=home. (Accessed: 6th March 2018)
3. Hardware virtualization - Wikipedia. Available at: https://en.wikipedia.org/wiki/Hardware_virtualization. (Accessed: 6th March 2018)
4. rkt. rkt/rkt. *GitHub* Available at: https://github.com/rkt/rkt. (Accessed: 19th April 2018)
5. Docker. *Docker* Available at: https://www.docker.com/. (Accessed: 6th March 2018)
6. NVIDIA. NVIDIA/nvidia-docker. *GitHub* Available at: https://github.com/NVIDIA/nvidia-docker. (Accessed: 4th April 2018)
7. Shifter: User Defined Images. Available at: http://www.nersc.gov/research-and-development/user-defined-images/. (Accessed: 21st March 2018)
8. Website. Available at: D. M. Jacobsen and R. S. Canon. Contain this, unleashing docker for HPC. Proceedings of the Cray User Group, 2015. https://www.nersc.gov/assets/Uploads/cug2015udi.pdf. (Accessed: 21st March 2018)
9. Kurtzer, G. M., Sochat, V. & Bauer, M. W. Singularity: Scientific containers for mobility of compute. *PLoS One* **12,** e0177459 (2017).
10. Priedhorsky, R. & Randles, T. Charliecloud: unprivileged containers for user-defined software stacks in HPC. in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis* 36 (ACM, 2017). doi:10.1145/3126908.3126925
11. CUDA Zone. *NVIDIA Developer* (2017). Available at: https://developer.nvidia.com/cuda-zone. (Accessed: 19th April 2018)
12. OpenCL - The open standard for parallel programming of heterogeneous systems. *The Khronos Group* (2013). Available at: https://www.khronos.org/opencl/. (Accessed: 19th April 2018)
13. OpenACC. Available at: https://www.openacc.org/. (Accessed: 19th April 2018)
14. wresch. wresch/python_import_problem. *GitHub* Available at: https://github.com/wresch/python_import_problem. (Accessed: 4th April 2018)
15. The OmpSs Programming Model | Programming Models @ BSC. Available at: https://pm.bsc.es/ompss. (Accessed: 4th April 2018)
16. Dynamic Load Balancing | Programming Models @ BSC. Available at: https://pm.bsc.es/dlb. (Accessed: 4th April 2018)
17. Nanos ++ | Programming Models @ BSC. Available at: https://pm.bsc.es/nanox. (Accessed: 4th April 2018)
18. Mercurium | Programming Models @ BSC. Available at: https://pm.bsc.es/mcxx. (Accessed: 4th April 2018)
19. Extrae | BSC-Tools. Available at: https://tools.bsc.es/extrae. (Accessed: 4th April 2018)
20. NCC Container Security Whitepaper. Available at: https://www.nccgroup.trust/globalassets/our-research/us/whitepapers/2016/april/ncc_group_understanding_hardening_linux_containers-1-1.pdf. (Accessed: 26th March 2018)

21. unioslo. unioslo/socker. *GitHub* Available at: https://github.com/unioslo/socker. (Accessed: 27th March 2018)
22. DockerHub. Available at: https://hub.docker.com/. (Accessed: 4th April 2018)
23. Quay. Available at: https:quay.io. (Accessed: 4th April 2018)
24. Container Catalog - Red Hat Customer Portal. Available at: https://access.redhat.com/containers/. (Accessed: 4th April 2018)
25. vsoch. singularity-hub. *Singularity Hub* Available at: https://singularity-hub.org/. (Accessed: 4th April 2018)
26. Dockstore. Available at: https://dockstore.org/. (Accessed: 4th April 2018)
27. common-workflow-language. common-workflow-language/common-workflow-language. *GitHub* Available at: https://github.com/common-workflow-language/common-workflow-language. (Accessed: 4th April 2018)
28. Container Registry - Private Docker Registry | Google Cloud. *Google Cloud* Available at: https://cloud.google.com/container-registry/. (Accessed: 4th April 2018)
29. OpenShift: Container Application Platform by Red Hat, Built on Docker and Kubernetes. Available at: https://www.openshift.com/. (Accessed: 4th April 2018)
30. The only single product for the complete DevOps lifecycle - GitLab. *GitLab* Available at: https://about.gitlab.com/. (Accessed: 4th April 2018)
31. Open Source | JFrog. *JFrog* Available at: https://jfrog.com/open-source/. (Accessed: 4th April 2018)
32. Kubernetes. *Kubernetes* Available at: https://kubernetes.io/. (Accessed: 4th April 2018)
33. Swarm mode overview. *Docker Documentation* (2018). Available at: https://docs.docker.com/engine/swarm/. (Accessed: 4th April 2018)
34. Apache Mesos. *Apache Mesos* Available at: http://mesos.apache.org/. (Accessed: 4th April 2018)
35. Apache Spark™ - Lightning-Fast Cluster Computing. Available at: https://spark.apache.org/. (Accessed: 4th April 2018)
36. Project Jupyter. Available at: http://www.jupyter.org. (Accessed: 4th April 2018)
37. Jantz, R. & Giarlo, M. J. Digital Preservation: Architecture and Technology for Trusted Digital Repositories. *Microform & Imaging Review* **34,** (2005).
38. Website. Available at: Consultative Committee for Space Data Systems Secretariat 2012, Reference model for an open archival information system (OAIS): Recommended practice (CCSDS 650.0-M-2: Magenta Book), CCSDS, Washington, DC. https://public.ccsds.org/pubs/650x0m2.pdf. (Accessed: 3rd April 2018)
39. Open Archival Information System - Wikipedia. Available at: https://en.wikipedia.org/wiki/Open_Archival_Information_System. (Accessed: 3rd April 2018)
40. Metadata Object Description Schema: MODS (Library of Congress Standards). Available at: http://www.loc.gov/standards/mods. (Accessed: 3rd April 2018)
41. Metadata Encoding and Transmission Standard (METS) Official Web Site | Library of Congress. Available at: http://www.loc.gov/standards/mets. (Accessed: 3rd April 2018)
42. PREMIS. Available at: http://www.loc.gov/standards/premis/v2/premis-2-0.pdf. (Accessed: 3rd April 2018)