EPJ Web of Conferences **251**, 02033 (2021) https://doi.org/10.1051/epjconf/202125102033
CHEP 2021

CernVM-FS powered container hub

Enrico Bocchi[1*], *Jakob* Blomer[1], *Simone* Mosciatti[1], and *Andrea* Valenzuela[1]

[1]CERN, Esplanade des Particules 1, 1211 Geneva 23, Switzerland

Abstract. Containers became the de-facto standard to package and distribute modern applications and their dependencies. The HEP community demonstrates an increasing interest in such technology, with scientists encapsulating their analysis workflow and code inside a container image. The analysis is first validated on a small dataset and minimal hardware resources to then run at scale on the massive computing capacity provided by the grid. The typical approach for distributing containers consists of pulling their image from a remote registry and extracting it on the node where the container runtime (e.g., Docker, Singularity) runs. This approach, however, does not easily scale to large images and thousands of nodes. CVMFS has long been used for the efficient distribution of software directory trees at a global scale. In order to extend its optimized caching and network utilization to the distribution of containers, CVMFS recently implemented a dedicated container image ingestion service together with container runtime integrations. CVMFS ingestion is based on per-file deduplication, instead of the per-layer deduplication adopted by traditional container registries. On the client-side, CVMFS implements on-demand fetching of the chunks required for the execution of the container instead of the whole image.

1 Introduction

In recent years, container technologies have seen wide adoption by software developers, system administrators, and IT practitioners to the point of becoming the preferred way to package, distribute, and deploy applications. Containers provide a lightweight isolation layer that decouples the application running inside from the specifics of the host environment and simplifies the deployment of software and services on diverse infrastructures, from personal laptops to distributed clusters in the cloud.

A crucial role for the deployment of applications in containers is the successful storage and distribution of container images. These tasks are typically delegated to *container registries*, specialized repositories meant to store container images. Registries can be public, the most prominent example being Docker Hub [4], private, offered as turn-key services by major commercial cloud providers, or self-hosted. The distribution of images is based on a push–pull model where the maintainers of containers build new images with bug fixes releases and/or new software releases and push (i.e., upload) them to container registries. Interested users can then pull (i.e., download) the images on their node to run the contained software.

The High Energy Physics (HEP) community shows an increasing interest in container technologies, with scientists and researchers making use of containers to encapsulate their

*e-mail: enrico.bocchi@cern.ch

analysis workflow and code into an immutable and distributable object. Containers, indeed, well meet the needs of big science environments: i) Reproducibility, with containers providing the ability to freeze software and tools used for analysis so to reproduce it in the future; and ii) large-scale execution, with containers being easily deployable on a heterogeneous set of computing resources to take full advantage of the available computational power.

At the scale of big science, however, the distribution of containers based on the traditional push–pull model shows performance and scalability limitations. Container images built and used in the HEP environment can easily reach tens of gigabytes in size and, even if pushed only once to the registry, they can potentially be pulled by thousands of computing nodes part of the Worldwide LHC Computing Grid (WLCG) [1]. This puts additional load on both the network infrastructure from the container registry to the computing nodes and the storage capacity of each computing node, given that container images must be downloaded and unpacked into the local filesystem.

The CernVM File System (CVMFS) has recently seen the implementation of additional capabilities to support the ingestion of existing container images and their distribution at scale, as well as to integrate with widely-adopted container runtimes. This paper describes HEP container images and the CVMFS role for their distribution (Sec. 2), explains newly implemented functionalities for container support both server- and client-side (Sec. 3.1 and Sec. 3.3, respectively), and evaluates the performance ingestion and the improved efficiency in storage and distribution (Sec. 4). Finally, Sec. 5 draws the conclusions.

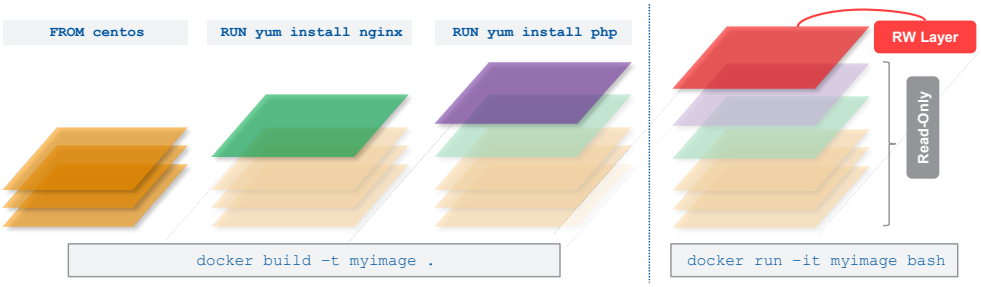
2 The containers ecosystem and CVMFS role for HEP images

Container technologies rely on three components to successfully distribute and run images: i) Container runtimes, which are installed on the host machine and manage components and resources needed to run containers; ii) container images, immutable self-standing packages that encapsulate binaries, dependencies, configuration files, etc., and allow for portability of the containerized applications; and iii) container registries, repositories where images are pushed to by publishers or pulled from by interested users.

Taking the storage perspective, container images are made of a collection of layers that represent different filesystem snapshots in `tar` format. A json file called *manifest* specifies how to stack all the layers to obtain the final filesystem required to run the container. The Open Container Initiative (OCI) [3] standard defines the filesystem snapshots, i.e., the layers, as *chains*. Fig. 1 exemplifies the process of building and running a sample image.

This approach suffers from several limitations caused by both the format chosen for storing images and the distribution mechanism. Firstly, the breakdown of container images into layers severely impacts the ability to deduplicate content (e.g., if the same file exists in different layers, it will be stored multiple times). Deduplication can happen only with a per-layer granularity, which is rather coarse and, worse, depends on how maintainers build their images. Secondly, container images can be big in size (also due to the limitations with deduplication), leading to an increased demand of network bandwidth when pulling the image and to a longer waiting time before being able to run the container. Lastly, big images will require additional storage space on the hosts where the container runtime executes. Local storage can be a scarce resource in some computing environments, e.g., on High Performance Computing (HPC) systems, and automatic eviction of unused images is typically not implemented by the container runtime to reclaim storage space.

CVMFS addresses all the limitations described above and greatly improves the efficiency in the distribution and storage of container images. When ingesting a container image in CVMFS, its content is subject to file-level deduplication, which is more efficient than layer-based deduplication. Statistics collected from a production scenario (better detailed in Sec. 4).




```
# docker history myimage
IMAGE               CREATED             CREATED BY                                      SIZE       CUMULATIVE
75cc2375258a        4 seconds ago       /bin/sh -c yum -y install php                   66.9MB     382MB
e779b8a4024f        9 seconds ago       /bin/sh -c yum -y install nginx                 77.8MB     315MB
470671670cac        4 days ago          /bin/sh -c #(nop)  CMD ["/bin/bash"]            0B         237MB
<missing>           4 days ago          /bin/sh -c #(nop)  LABEL org.label-schema.sc…   0B         237MB
<missing>           7 days ago          /bin/sh -c #(nop) ADD file:aa54047c80ba30064…   237MB      237MB
```

Figure 1. The process of building and running a sample image with tag `myimage`. The container runtime builds on top of an existing image tagged as `centos` and installs the `nginx` and `php` packages in two different steps. Each step corresponds to an additional read-only layer overlaying the previous image. When the build process is complete, users can run the resulting image and the container runtime will add a read-write layer on top, used as scratch space during the container execution.

In addition, computing nodes no longer need to download the full images locally but rather take advantage of the lazy-loading feature provided by CVMFS. This allows downloading only the files that are strictly needed for the execution of the container (previous studies [2] confirm our own findings that only a small percentage of the total image volume is actually used), ultimately saving network bandwidth and local storage space. CVMFS implements an advanced cache area that can make use of different storage types (local, external, tiered) and is configurable in size. Such cache is self-managed and files are automatically purged according to the least recently used policy.

2.1 Use of Containers in the HEP community

Containers are more and more popular in the HEP community. In HEP, containers are convenient for users to get a working setup quickly, for software librarians to distribute and store releases, and for scientists to run computations and reconstruction analysis.

HEP images are equally impacted by the problems of storage and distribution efficiency described above. In order to use CVMFS in place of traditional registries, we study which container images are of interest for our users, identifying three main classes:

- **Base images** containing the bare operating system. They change rarely, are relatively small in size, and are used as the basic building blocks to build other images on top.
- **Experiments' images** containing the software stack of a specific experiment. They are rather big due to the number of dependencies installed at the build phase, see new releases on a weekly basis, and are used by the users as building blocks for their custom images.
- **Users' images**, tailored to perform a specific task or analysis on data. These images change very frequently, multiple times per day during the development of an analysis, and require the experiment software stack, resulting in being very large in size.

Out of the three, users' images are the most challenging ones to support due to their large size and because users expect low latency of ingestion, i.e., they want to have the images available for running them at scale as soon as possible.

2.2 Use of CVMFS in the HEP community

CVMFS has long provided an efficient, globally-shared software area for HEP experiments. Large and frequently changing software directory trees are centrally maintained by software librarians. Clients load and cache only the tiny fraction of files that are required at any given point in time to run a particular compute job. There is widespread knowledge available in the community on how to operate CVMFS clients and the web-cache-based content delivery network with regional replica servers and site-local squid caches. Therefore, CVMFS is a natural match for the distribution of container image contents. Ideally, containerized applications should benefit from the host isolation and platform independence provided by the container runtime in concert with the distribution efficiency of CVMFS.

3 Container Support in CVMFS

This section explains how container images are stored in CVMFS to overcome the limitations of the push-pull model and to let scientists build and use their own images. The Daemon that Unpacks Container Images into CVMFS (DUCC) service has been developed to *ingest* images into a CVMFS directory tree. In the course of ingestion, DUCC unpacks the container image tarballs to exploit file-based deduplication and creates the required directory structures to let container runtimes readily pick up ingested images for execution.

3.1 Server capabilities for container images ingestion

Among the available container runtimes, three are particularly popular in the HEP community: i) Singularity [5] has its roots in the scientific environment and is the most used for containerized jobs on the WLCG; ii) containerd [6] implements the cri-o [7] interface used by Kubernetes [8] and hence integrates well with container orchestration tools; iii) Podman [9] has the ability to run rootless, is well integrated with the CentOS ecosystem, and provides an interface identical to the one offered by Docker.

Runtimes can be classified according to their support for the container image filesystem. Singularity expects images to be presented as a root filesystem to `chroot` to. Users can indeed run a Singularity container just by invoking the command against a directory path. containerd and Podman, instead, expect images to be structured as a collection of layers. Such runtimes employ a union filesystem to generate the final filesystem and run the container.

What follows describes how existing images are ingested and how their content is organized in the CVMFS directory structure to support runtimes with different requirements:

- **Layer-based runtimes** require that all the layers composing an image are available in CVMFS. A hidden top-level directory named `.layers` stores subfolders whose names match the hash of the layers stored inside. This allows for immediate identification of any layer and for verifying no duplicated layers exist. During ingestion, the image manifest is parsed to identify every layer that composes the image and their hash is computed. It is then checked whether the name of any sub-directories in `.layers` matches it. If so, the layer already exists (i.e., it was previously ingested) and DUCC continues by processing the following layer. Instead, if no sub-directory matches the computed hash, the layer is downloaded and ingested. Layers stored in CVMFS must be an exact copy of the layers

provided by the container registry. This might cause problems during the ingestion process as the user (typically unprivileged) running the CVMFS ingestion commands might not have the required permissions to create and populate directories (i.e., owned by `root`). CVMFS circumvents the problem by using the `cvmfs_server ingest` command that bypasses the default permission checks and preserves the permissions of the original image. The meta-data of every layer directory is stored in a separate CVMFS *nested catalog*.

- **Flat filesystem runtimes** require the image root filesystem. If many users' images (i.e., big images that change frequently, see Sec. 2.1) have to be ingested, downloading all the layers and building the final filesystem for each of them can easily become a bottleneck. Users' images, however, are typically based on the same experiments' images. A naive approach would force us to download all the layers (including the ones from the experiments' images) every time the user changes her own, relatively small, part. Instead, we store all the chains (i.e., the filesystems snapshots) of each image in CVMFS and re-use them. For instance, if the user updates the last layer of her container, the second-to-last chain would still be valid for both the old and the new image. In this case, instead of recreating the whole filesystem of the new image from scratch, we copy the second-to-last chain into another directory and apply the last layer on top of it. As a naïve copy of the filesystem would be inefficient, CVMFS 2.8 introduces a new feature called *template transactions*: Metadata-only operations that allow copying quickly a CVMFS directory into another one.

3.2 Manage image ingestion in CVMFS

The development of DUCC took place in close contact with experiments at CERN, which confirmed their interest in having the base and the experiment images ingested in CVMFS. Our goal, however, is to equally well support user images. For this purpose, we need a simple way to let users push their images and to CVMFS. As a result, we designed DUCC to support different methods to signal which images must be ingested.

- **Wishlist** – Users update a shared document where they specify which images should be ingested. DUCC regularly checks if the listed images are available and up-to-date on CVMFS and, if that is not the case, proceeds with the ingestion. This approach is straightforward but introduces latency between the creation/update of an image and its ingestion in CVMFS.
- **Webhook notification** – Traditional container registries offer a webhook notification mechanism that allows receiving an HTTP call when specific events take place on the registry. It is hence possible to register a webhook notification for images pushed to the registry that is intercepted by DUCC, which will in turn download the image from the registry and ingest it into CVMFS. This approach overcomes the latency limitation of the wishlist and does not require any additional action from the users other than pushing their images to the appropriate registry. The webhook integration has been developed and tested to work with Harbor [10], a community-developed container registry deployed at CERN.
- **Registry shim** – A specialized proxy intercepts the tarballs pushed by the users, ingests them into CVMFS, and forwards them to a traditional container registry. The biggest advantage of this approach is the provisioning of a synchronous API, which makes the images immediately available in CVMFS and in the registry. However, the proxy has not been implemented so far due to the development effort required and the fact that the CVMFS infrastructure anyway has, although small, an intrinsic propagation delay.

3.3 Integration with container runtimes

The container runtimes available on compute nodes must be instrumented to use CVMFS as the data source for container images. Instead of pulling images from a remote container

registry, runtimes should rather access the CVMFS repository hosting the images of interest and fetch the required files or layers from there. The way this happens depends on each container runtime and on the image format they expect:

- **Singularity** [5] has been one of the first runtimes to exploit CVMFS. The integration between the two is seamless as Singularity supports running containers directly from the CVMFS directory hosting the container image previously ingested by DUCC. Users can select which container to run by invoking Singularity against the specific directory.
- **containerd** [6] provides support for plugins to implement custom functionalities. CVMFS integration with containerd is achieved by the *snapshotter* plugin, a specialized component responsible for assembling all the layers of container images into a stacked filesystem that containerd can use. The snapshotter takes as input the list of required layers and outputs a directory containing the final filesystem. It is also responsible to clean-up the output directory when containers using it are stopped.
- **Podman** [9] supports the concept of *additional image stores*[1], a set of custom directories hosting usable images. The filesystem used by Podman for its additional stores is compatible with CVMFS, hence it is sufficient to replicate in CVMFS the expected directory structure to allow Podman to use images from CVMFS.

4 Evaluation

This section reports on DUCC ingestion performance. Measurements are taken at the CERN Data Centre on a machine equipped with 16 CPUs and 64 GB of RAM running CentOS 7. CVMFS is set up to publish to the local SSD disk, while source container images are provided by Docker Hub and by the GitLab Container Registry deployed at CERN. Performance measurements are based on logs output by DUCC.

4.1 Ingestion of layers

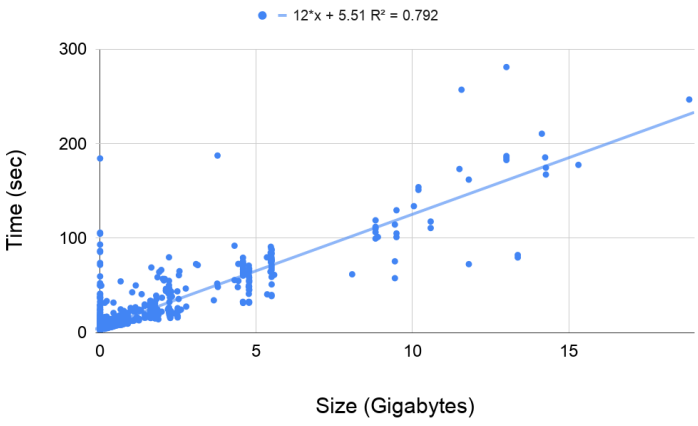


Figure 2. Time needed to to ingest layers in CVMFS.

[1]https://www.redhat.com/sysadmin/image-stores-podman

The time needed to ingest one layer can be qualitatively approximated (Fig. 2) with:

$$Time\ [s] = 1.2 \times 10^{-2} \times Size\ [MB] + 5$$

Our setup takes roughly one second every 80 MB of content to publish. The time needed to maintain the metadata information can be considered negligible. The constant part considers the time to publish a CVMFS transaction, which is independent of its size.

Before the ingestion in CVMFS, layers need to be downloaded from the remote registry. In the test setup, all the layers need to be ingested and they are downloaded in parallel. As soon as one layer is available, DUCC proceeds to ingest it. This approach allows us to wait a minimum amount of time between downloading and ingestion since when one layer is being ingested, the others are concurrently being downloaded.

4.2 Ingestion of chains

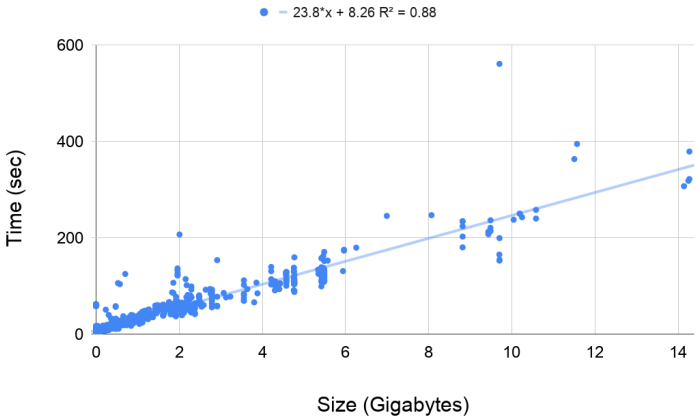


Figure 3. Time needed to ingest chains in CVMFS.

The trend line for the creation of a chain id (see Fig. 3) can be defined as:

$$Time\ [s] = 2.3 \times 10^{-2} \times Size\ [MB] + 8$$

The higher constant factor (compared to layers ingestion) owes to the fact that layers can not be downloaded in parallel when ingesting a chain. Each layer must be downloaded sequentially and, therefore, DUCC needs to wait for the download over the network of the next layer before continuing with the ingestion. In addition, creating a chain requires unpacking the tarred layers on the local filesystem. This operation requires additional time that makes the overall process slower, while the ingestion of pure layers uses a faster internal code path that does not demand interactions with the local filesystem. We are attributing the smaller coefficient factor to the less overhead that we incur in creating the incremental layer instead of ingesting the tarball in CVMFS.

4.3 Characterization of image repositories

We characterize the content of two repositories that constitute CVMFS-powered container hubs: `/cvmfs/unpacked.cern.ch` and `/cvmfs/singularity.opensciencegrid.org`, storing more than 800 and 500 images, respectively, for the HEP community.

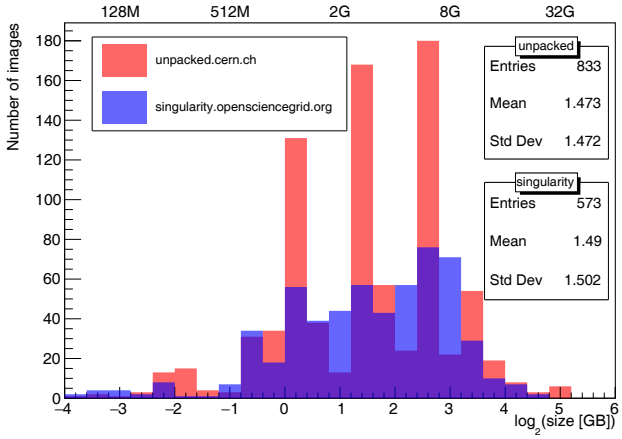


Figure 4. Size distribution (uncompressed) of container images stored in `/cvmfs/unpacked.cern.ch` and `/cvmfs/singularity.opensciencegrid.org`.

Fig. 4 shows the size distribution of images stored in the two repositories and demonstrates that HEP images tend to grow large, with a mean image size bigger than 2 GB. It is therefore likely to overflow computing node disks when many concurrent jobs use different container images. In this scenario, pulling the whole image from a traditional container registry would not scale, while the approach taken by CVMFS makes smarter use of the local disk space by fetching, caching, and automatically evicting only the required files.

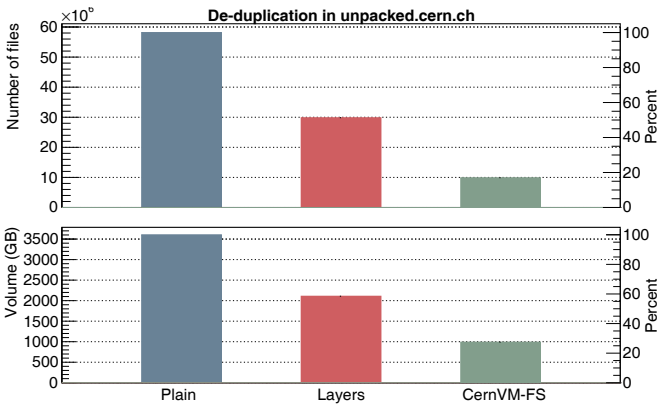


Figure 5. Comparison of layer-based and file-based deduplication efficiency.

Fig. 5 shows the deduplication efficiency for images stored in `unpacked.cern.ch`. While layer-based deduplication (used by traditional registries) saves approximately 40% of files and bytes, the file-based deduplication implemented in CVMFS is much more effective (80% of files and 70% of bytes), demonstrating to be twice as efficient.

4.4 Client-side cache efficiency

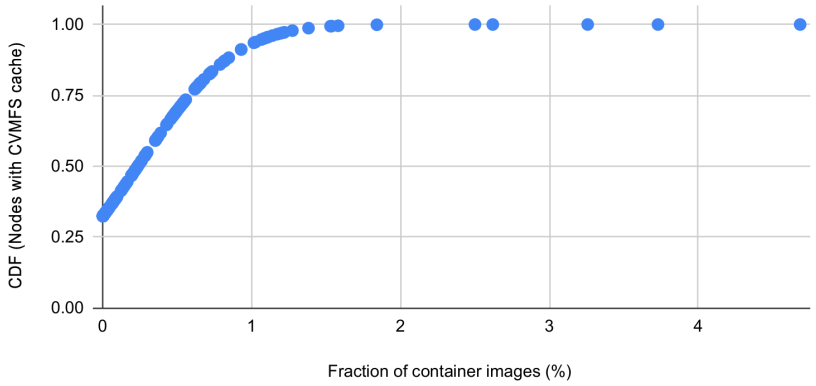


Figure 6. Cumulative distribution of the fraction of an image required to start the container.

In order to assess the cache efficiency on the client side, we analyzed the content of CVMFS client caches on the CERN batch farm with respect to container image contents. The dataset includes 1800 nodes and was captured in February 2021. The results (Fig. 6) show that a very small fraction of container images is needed to start the container and execute the contained applications. The traditional push–pull model would require the download of the entire images on the nodes, while CVMFS fetches only the files that are actually required, saving time to start the application, network bandwidth, and local disk space.

5 Conclusions

This paper describes the workflow and the tools to distribute standard OCI container images via CVMFS in an efficient and scalable way. This allows scientists and researchers to develop their software locally and encapsulate it into a container image. Once ready, the image is pushed to a container registry and automatically ingested in CVMFS to enable its execution for production jobs at the WLCG scale.

There exist several advantages in distributing standard containers with CVMFS: i) Containers maintain the same isolation properties of standard containers distributed with the traditional layer-based approach; ii) The file-based deduplication used by CVMFS shines for efficiency compared to the traditional layer-based one. This reduces storage and network capacity requirements; iii) The on-demand caching feature implemented in CVMFS clients eliminates the need of downloading the entire container image locally, making wiser use of the local disk space and reducing the time needed to start the container.

Adapting CVMFS to use standard OCI images without relying on ad-hoc formats allows for quick onboarding: Users can create their images with familiar tools while DUCC is instrumented to interpret the resulting images and make them available in CVMFS.

In addition, container distribution via CVMFS builds on top of existing software and hardware infrastructure in the WLCG and does not require additional components or configuration changes. To an extreme, the Singularity container runtime and the containerd snapshotter can themselves be distributed over CVMFS, enabling container execution on computing resources that would miss the required packages.

References

[1] Bird, I., et al. "Update of the Computing Models of the WLCG and the LHC Experiments." CERN-LHCC-2014-014. 2014. https://cds.cern.ch/record/1695401

[2] Harter, T., et al. "Slacker: Fast distribution with lazy docker containers." 14th USENIX Conference on File and Storage Technologies (FAST 16). 2016.

[3] Open Container Initiative, https://opencontainers.org/ (last accessed 15/02/2021)

[4] Docker Hub, https://hub.docker.com/ (last accessed 12/02/2021)

[5] Kurtzer G. M., Sochat V., Bauer M. W. "Singularity: Scientific containers for mobility of compute" PloS one. 2017. https://doi.org/10.1371/journal.pone.0177459

[6] containerd – An industry-standard container runtime with an emphasis on simplicity, robustness, and portability. https://containerd.io/ (last accessed 15/02/2021)

[7] CRI-O: OCI-based implementation of Kubernetes Container Runtime Interface https://cri-o.io/ (last accessed 15/02/2021)

[8] Bernstein, D. "Containers and cloud: From lxc to docker to kubernetes." IEEE Cloud Computing 1.3 (2014): 81-84.

[9] Podman. https://podman.io/ (last accessed 15/02/2021)

[10] Harbor. https://goharbor.io/ (last accessed 15/02/2021)