docker & HEP: Containerization of applications for development, distribution and preservation

S. Binet, B. Couturier

E-mail: binet@cern.ch

Abstract.

HEP software stacks are not shallow. Indeed, HEP experiments' software is usually many applications in one (reconstruction, simulation, analysis, ...) and thus require many libraries – developed in-house or by third parties – to be properly compiled and installed. Moreover, because of resource constraints, experiments' software is usually installed, tested, validated and deployed on a very narrow set of platforms, architectures, toolchains and operating systems. As a consequence, bootstrapping a software environment on a developer machine or deploying the software on production or user machines is usually perceived as tedious and iterative work, especially when one wants the native performances of bare metal.

Docker containers provide an interesting avenue for packaging applications and development environment, relying on the Linux kernel capabilities for process isolation, adding git-like capabilities to the filesystem layer and providing (close to) native CPU, memory and I/O performances.

1. Introduction

Development of High Energy and Nuclear Physics (HENP) software is following more and more best practices devised in the industry. These recipes help practitioners manage code versioning, ensure build reproducibility and tame the unbridled growth of external dependencies. The tools developed in the software industry to cope (even at scale) with these mundane issues have now percolated to some extent in HENP experiments.

Moreover, development and application isolation have been facilitated by the increased usage of virtual machines, which also greatly helped portability. However, even if production clusters are usually Linux systems running Linux virtual machines, this portability comes at a price: resources overhead.

While virtual machines provide a machine-level virtualization environment, containers provide an operating-system-level virtualization environment for running multiple isolated operating-systems. Containers seem to better match the main production use-case of typical HENP clusters. LXC [4] and OpenVZ [5] were the first to introduce containers into the Linux ecosystem, but Docker [1] is the project that really popularized and democratized them.

This paper explores the possible applications of Docker containers to typical HENP workflows. We first introduce in more details the *modus operandi* of Docker containers and then focus on the hepsw/docks containers which provide containerized software stacks for – among others – the LHCb Experiment [18] at CERN. Then, we discuss various strategies experimented with

to package software (*e.g.* `cvmfs` [6], `RPM`s, source-based) and how we applied them to optimize provisioning speed and disk usage, leveraging the caching system of `Docker`. Finally, we report on benchmarks comparing workloads on bare metal with regard to `Docker` containers setups.

2. A `Docker` primer

`Docker` is an open source project to pack, ship and run any application as a lightweight container.

`Docker` uses `Linux` namespaces, `cgroups` [7] and unioning file systems to isolate processes. Container images are rather similar to virtual machine images, but share the `Linux` kernel with the host machine. This provides a much more lightweight setup and allows to provision container images in seconds – compared to minutes for virtual machines – as well as to run hundreds of such containers on a typical desktop machine.

Thanks to `cgroups`, containers can have their own network interface: in layman's terms, containers can be seen as a super-`chroot`, with no device emulation – hence the almost bare metal performance. `Docker` also provides a layered file system, building on unioning file systems (`UnionFS`[19], `AUFS` [20]) or `device mapper` [21] for kernels without such modules. Each layer of the file system is mounted on top of prior layers. The first layer is the so-called *base image*, holding an initial collection of files and folders provided by a distribution (Ubuntu, RHEL, Fedora, etc...) which does not need to be the same as the host distribution. Each layer is read-only (only the top-layer is modifiable) and only stores on disk the *delta* with regard to the previous layer. Individual layers are indexed by hashes à la `git` and can be shared among images: this scheme enables very lightweight disk resources requirements and thus, very fast disk provisioning performance.

2.1. `Docker` images

`Docker` images are created from a base container. `Docker` ships a comprehensive set of *official* base containers (`ubuntu`, `centos`, etc...) which can be downloaded locally *via* the `docker pull` command, as shown in figure 1.

```
>> docker pull ubuntu
latest: Pulling from ubuntu
e9e06b06e14c: Pull complete
a82efea989f9: Pull complete
37bea4ee0c81: Pull complete
07f8e8c5e660: Already exists

Digest: sha256:8126991394342c2775a9ba4a843869112da8156037451fc424454db43c25d8b0
Status: Downloaded newer image for ubuntu:latest
```

Figure 1: `pull` retrieves a `docker` image from the registry (also known as `docker hub`).

The list of local images can be queried using the `docker images` command, as shown in figure 2.

```
>> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
ubuntu              latest              07f8e8c5e660        5 days ago          188.3 MB
centos              latest              fd44297e2ddb        2 weeks ago         215.7 MB
ubuntu              12.10               c5881f11ded9        10 months ago       172.1 MB
```

Figure 2: `images` prints the list of local images, retrieved from the registry or created locally.

Once an image is created or downloaded from the `Docker` registry [2], it is possible to run an executable off that image, inside a container. As shown in figure 3, it is also possible to specify an explicit version (here 12.10) for the image one wants to run.

```
>> docker run ubuntu:12.10 echo "hello world"
hello world
```

Figure 3: `docker run` runs an executable inside a container. Here, the command `echo` is run on top of the base image `ubuntu`, explicitly using the version 12.10 of that image.

It is also possible to run containers in detached mode, the typical use case for (micro)services or web servers. The exact syntax is given in figure 4.

```
>> docker run -d ubuntu sh -c 'while true; do echo "hello"; sleep 1; done;'
0ac942723c259a4963e0feff04d57e9bf8ad28e72158a231111d8f3718d960e6

>> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
0ac942723c25        ubuntu:latest       "\"sh -c 'while true   6 seconds ago    Up 3 seconds

>> docker attach 0ac942723c25
hello
hello
hello
hello
...
```

Figure 4: `docker run` in detached mode. Here, the command `sh` is run on top of the base image `ubuntu`.

As the command is run in detached mode, one needs to attach to the running container (`0ac942723c25`) to see its output. A container can also be managed via the `start/stop/restart` subcommands.

`Docker` images can be searched for, published on and retrieved from the `Docker Hub` [2], a global registry of official and user provided images. This index is available from the `docker` command line, as shown in figure 5, but also from the web: `https://hub.docker.com`.

```
>> docker search apache
NAME                STARS      OFFICIAL    AUTOMATED
tomcat              131        [OK]
tutum/apache-php    71                     [OK]
httpd               50         [OK]
maven               32         [OK]
fedora/apache       30                     [OK]
[...]

>> docker pull fedora/apache
Pulling repository fedora/apache
963668e7af33: Download complete
963668e7af33: Pulling image (latest) from fedora/apache
3d26c48a13f: Download complete
Status: Downloaded newer image for fedora/apache:latest

>> docker run -d -p 80 fedora/apache
128d9712c922aab640a68d56c1cc35f5c17a889e136bc7983804035333264d92

>> docker ps
CONTAINER ID        IMAGE                     COMMAND             PORTS
128d9712c922        fedora/apache:latest      "/run-apache.sh"    0.0.0.0:32768->80/tcp

>> curl localhost:32768
Apache
```

Figure 5: `docker search` queries the `docker` registry for images matching a given string (either in their name or description.) The `fedora/apache` exposes the `Apache` web server on port `80` which needs to be exported to the host. `Docker` can remap that port to a non-privileged one.

2.2. Creating customized images

Users can create new images interactively, launching a new container off a base image, running commands interactively and committing the resulting state of that container into a new image, as shown in figure 6.

```
>> docker run -i -t ubuntu bash
root@5ad62f3a9b4b:/>> apt-get install -y memcached
[...]
Unpacking memcached (1.4.14-0ubuntu9) ...
root@5ad62f3a9b4b:/>> exit
exit

>> docker ps -l
CONTAINER ID        IMAGE               COMMAND             CREATED
5ad62f3a9b4b        ubuntu:latest       "bash"              3 minutes ago

>> docker commit 5ad62f3a9b4b binet/memcached
220c5747349a7ec1e16297eec0df11eac277b9a9a6a149e386aff9f63bec868e

>> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
binet/memcached     latest              220c5747349a        4 minutes ago       190 MB
ubuntu              latest              07f8e8c5e660        5 days ago          188.3 MB
centos              latest              fd44297e2ddb        2 weeks ago         215.7 MB
fedora/apache       latest              963668e7af33        2 weeks ago         627.1 MB
ubuntu              12.10               c5881f11ded9        10 months ago       172.1 MB
```

Figure 6: `docker run` runs the `bash` command in a container in interactive mode (`-i`) with a pseudo-`TTY` (`-t`). Once the container is in the wanted state (needed packages installed, applications correctly configured, etc...), it can be saved into a new image, named `binet/memcached` in this example.

Interactively creating new images is very useful for development or debugging the creation process. But for scalability and reproducibility purposes, a scripting interface is a necessity. The `Docker` project introduced the `Dockerfile` file specification which can be described as a `Makefile` for creating images. The syntax resembles that of shell scripts, with a few keywords described at [3].

The `Dockerfile` equivalent to the listing of figure 6 is shown in figure 7. Actually creating the new image is done by running `"docker build"` in the directory holding the `Dockerfile`.

```
>> cat Dockerfile
## create a memcached image
FROM ubuntu
MAINTAINER me@example.com

RUN apt-get install -y memcached

>> docker build --tag=binet/memcached .
Sending build context to Docker daemon 2.048 kB
Sending build context to Docker daemon
Step 0 : FROM ubuntu
 ---> 07f8e8c5e660
Step 1 : RUN apt-get install -y memcached
 ---> Running in 7f8349773ece
[...]
 ---> aac9f626b9fb
Removing intermediate container 7f8349773ece
Successfully built aac9f626b9fb

>> docker images
REPOSITORY          TAG                 IMAGE ID            CREATED             VIRTUAL SIZE
binet/memcached     latest              aac9f626b9fb        About a minute ago  190 MB
```

Figure 7: A simple `Dockerfile` scripting the creation of a new image off `ubuntu` where `memcached` is installed.

3. Containerization of HENP applications

Docker can be useful for a number of typical HENP workflows:

- encapsulating the build of an experiment software stack, ensuring there are no hidden implicit external dependencies;
- installing an already built software stack, easily deploying it on any number of nodes and sites, and quickly productizing containers;
- distributing stable development environments.

We have created a number of base images for general use for the HENP community and collected them under the github.com/hepsw/docks repository, namely:

- Scientific Linux CERN 5 (hepsw/slc5-base),
- Scientific Linux CERN 6 (hepsw/slc-base) and,
- CERN CentOS 7 (hepsw/cc7-base.)

Committed under this repository are also the Dockerfile and other supporting files to create images holding the binary installation of the Gaudi [13] control framework (hepsw/lhcb-gaudi) and the LHCb physics analysis software, Da Vinci (hepsw/lhcb-davinci.) These images have not been published on the docker registry because of their size ($O(10GB)$): the available bandwith on the docker hub render them impractical to distribute. Having a HENP-dedicated registry on dedicated hardware, handling Virtual Organizations and Grid certificates, would lift this issue.

In the author's opinion, the most difficult task was to create the base images for the HENP customized SLC images, as the documentation on how to create an image from scratch – without any other image to base it on – is scarce. The obvious strategy of basing the SLC images off the official centos and then updating the whole system to its CERN flavour resulted in too large ($O(GB)$) disk images, as the SLC software portfolio diverged too much from CentOS. Indeed, updating the whole system from the official centos image required to modify many files across the filesystem. This operation, coupled with the snapshotting feature of docker that saves the state of the filesystem before and after a command, tremendously increased the size of the final image eventhough docker was smart enough to only record the diffs. Eventually, the hepsw/slc* images have been created with the rinse [8] tool which usefully provides a template for Scientific Linux. Using rinse allowed to sidestep the snapshotting feature of docker and import the final state of the filesystem directly into a fresh docker image. The hepsw/cc7-base image was created from the official centos:centos7 image with the CERN yum repositories tacked on. CERN CentOS 7 [1] is more closely based on CentOS-7 hence the disk image size was not an issue.

In comparison, the process of creating the hepsw/lhcb-* was a much easier task, mainly consisting in identifying the hidden (runtime) dependencies on the underlying operating system, translating them into needed RPMs to install and bulk-installing the LHCb applications' binaries with the *ad hoc* tool, lbpkr [9].

Finally, we packaged cvmfs for the ATLAS, CMS, LHCb and LSST software stacks. cvmfs is a network filesystem optimized for read access and can deliver experiment software in a fast, scalable, and reliable way. The hepsw/cvmfs-* images contain a correctly configured cvmfs daemon, ready to fetch (lazily, on demand) and cache binaries and other assets for each of the four experiments. The resulting image, relatively small by today's standards ($650MB$) needs to be run with additional privileges (with the --privileged command line option) for FUSE's

[1] CERN CentOS 7 is expected to be the next production platform. But work is still on going to fully validate LHC experiments' software stack on this operating system, thus we will only consider SLC for the remainder of this article.

```
## install gaudi from RPMs
FROM hepsw/slc-base
MAINTAINER binet@cern.ch

ENV MYSITEROOT /opt/lhcb-sw
ENV CMTCONFIG x86_64-slc6-gcc48-opt

RUN mkdir -p ${MYSITEROOT} && \
    echo ":: installing software under [${MYSITEROOT}]..."

## install some system dependencies
RUN yum install -y bzip2 freetype glibc-headers tar which

## retrieve install
RUN curl -O -L  http://cern.ch/lhcbproject/dist/rpm/lbpkr && \
    chmod +x ./lbpkr

## install (source+binaries)
RUN ./lbpkr install-project GAUDI v26r1
```

Figure 8: `Dockerfile` scripting the creation of the `hepsw/lhcb-gaudi` image.

benefit. Installing `cvmfs` can be difficult on non-standard or exotic `Linux` distributions: the `hepsw/cvmfs-base` can thus be a quick and easy way to install and test it, even on `MacOSX` [2]. Note that while a `cvmfs`-based `docker` image will fetch the needed binaries from a *ad hoc* repository, the cache of binaries is currently not shared between running containers [3].

4. Benchmarks

Since the rise of `docker` on the DevOps [22] scene, numerous benchmarks have been published [11, 12], testing provisioning, measuring *CPU* and memory usage, etc... In this paper, we tested a typical LHCb application (`gaudirun TupleEx.py`), measuring the image disk sizes and container resources, compared with bare metal, using the complete software distribution over `AFS` [23], a very popular (at CERN) distributed read/write file system.

4.1. Disk size

The image disk sizes are reported in Table 1. The `hepsw/lhcb-base` image is based on `hepsw/slc-base` and thus only adds a couple hundreds of megabytes. Adding the complete `Gaudi` framework on top of `hepsw/lhcb-base` amounts to almost four gigabytes, and installing the whole physics analysis suite on top of `hepsw/lhcb-base` amounts to almost eight gigabytes. `hepsw/cvmfs-base` is based on `hepsw/`

Image	Tag	Size
hepsw/slc-base	6.6	135.6 MB
hepsw/lhcb-base	20150331	336.6 MB
hepsw/lhcb-gaudi	v26r1	3.911 GB
hepsw/lhcb-davinci	v36r5	7.790 GB
hepsw/cvmfs-base	20150331	629.4 MB
hepsw/cvmfs-lhcb	20150331	629.4 MB

Table 1: Image disk sizes as reported by `docker images`.

Neither sharing the operating system with the host nor leveraging the copy-on-write mechanism from the unioning filesystems help reducing the disk storage requirements compared

[2] Running `docker` on `MacOSX` requires to run a thin `Linux` virtual machine where `docker` is installed. Everything has been packaged and streamlined in the `boot2docker` [10] project.

[3] see issue `https://github.com/hepsw/docks/issues/11`

to a pure virtual machine based approach as the amount of binaries needed by LHC experiments software completely dwarfs that of a full operating system. The disk sizes of the installed software as shown by du(1) is reported in Table 2.

Image	Directory	Size
hepsw/lhcb-base	/opt/lhcb-sw	67 MB
hepsw/lhcb-gaudi	/opt/lhcb-sw	3.600 GB
hepsw/lhcb-davinci	/opt/lhcb-sw	7.300 GB

Table 2: Disk sizes of the installed software as reported by du(1).

The image sizes for hepsw/cvmfs-* weigh less than $650MB$ and are much more easily distributable: once cvmfs and its dependencies are installed (*i.e.:* hepsw/cvmfs-base), adding the configuration for LHCb is lost in the noise. These rather lean images have not yet any software installed: they will need network access on first usage.

4.2. CPU *and* VMem

For this benchmark, we ran repeatedly a test application packaged with the Gaudi installation which involves creating histograms and n-tuples, and saving them on disk. For each of the three configurations tested – AFS, hepsw/lhcb-gaudi and hepsw/cvmfs-lhcb – we ran that application, measured *CPU* with time(1) and *VMem* with top(1). No difference in *VMem* usage was noticed. The results for the *CPU* usage are reported in Table 3. The three setups show similar performance except for the very first they are run: on a cold cache, AFS and hepsw/cvmfs-lhcb need to retrieve (and then cache) the needed binaries over the network. Once this one-time overhead was dodged, performance was stable and similar across the board.

usr (s)	sys (s)	CPU	real
56.87	14.26	66%	1:46.50
57.62	13.07	99%	1:11.17
57.69	13.46	99%	1:11.58
57.93	13.26	99%	1:11.66

(a) AFS timings.

usr (s)	sys (s)	CPU	real
55.93	12.34	98%	1:09.54
55.43	12.88	98%	1:09.12
55.54	12.16	98%	1:08.83
55.39	11.60	98%	1:07.81

(b) hepsw/lhcb-gaudi timings.

usr (s)	sys (s)	CPU	real
55.53	14.01	88%	1:18.75
54.95	12.83	97%	1:09.36
55.42	12.86	98%	1:09.35
55.42	13.01	98%	1:09.63

(c) hepsw/cvmfs-lhcb timings.

Table 3: *CPU* timings for AFS (left), hepsw/lhcb-gaudi (right) and hepsw/cvmfs-lhcb (middle). Notice how the outliers (first row) for AFS and hepsw/cvmfs-lhcb: the needed binaries are being fetched over the network.

5. Conclusions and Outlook

We presented the various use cases where containerization technologies like docker could make a beneficial impact in HENP workflows. *Containerizing* a HENP software stack is easily doable

once base images tailored to HENP software environments are created and published. Containers do not show degraded performances compared to running the same executable on bare metal in a *CPU* and *I/O* intensive setup. Thus, docker-based workflows could potentially improve the resources usage efficiency of our Linux clusters, addressing the issue of non-homogenous use of Linux distributions on the Grid without requiring virtual machines.

Another interesting use of containers would be packaging and data preservation. Once an image containing a software stack has been prepared, it can be easily shared and deployed. While the on-disk representation of such a packaging is just tar(1) – a reliable and serviceable file format – for such a setup to be viable in the long term, an on-disk specification needs to exist to prevent lock-in. The appc [14] App Container format, started by rkt [15] a docker competitor, is an attempt to address this issue. Developments in that area need to be followed up. Another issue to monitor is the security model provided by containers. The monolithic approach of docker (a single daemon with root privileges) renders it prone to privileges escalation and thus hinders its use on interactive clusters like lxplus [24]. Its rkt competitor is more modular and touted to be more security conscious.

A tighter integration with nightly build systems (*e.g.* Jenkins [16]) and build clusters (*e.g.* Mesos [17]) might be the subject of a future investigation. Indeed, this would pave the way towards having a universal development and testing environment which could be easily shared with other fellow developers (for debugging a thorny build or runtime problem) or easily deployed on HENP controlled resources but as seamlessly migrated to other commercial cloud platforms.

References

[1] Docker, http://docker.io
[2] Docker Hub, https://hub.docker.com
[3] Dockerfile syntax reference, https://docs.docker.com/reference/builder
[4] LXC, https://linuxcontainers.org
[5] OpenVZ, https://openvz.org
[6] CernVM File System, http://cernvm.cern.ch/portal/filesystem
[7] Linux control groups, http://en.wikipedia.org/wiki/Cgroups
[8] rinse, RPM installation entity, http://collab-maint.alioth.debian.org/rinse
[9] lbpkr, https://github.com/lhcb-org/lbpkr
[10] boot2docker, a lightweight Linux distribution to run docker containers, http://boot2docker.io/
[11] KVM and docker benchmarking, http://bodenr.blogspot.ch/2014/05/kvm-and-docker-lxc-benchmarking-with.html
[12] Felter W. et al., "An Updated Performance Comparison of Virtual Machines and Linux Containers", RC25482, 2014
[13] Barrand G. et al., "GAUDI – A software architecture and framework for building LHCb data processing applications", CHEP 2000
[14] App Container Specification and Tooling, https://github.com/appc/spec
[15] rkt, an App Container runtime for Linux, https://github.com/coreos/rkt
[16] Jenkins, an extensible open source continuous integration server, https://jenkins-ci.org/
[17] Mesos, a distributed systems kernel, http://mesos.apache.org/
[18] LHCb Technical Proposal, CERN/LHCC 98-4, LHCC/P4 (1998)
[19] UnionFS, A Stackable Unification File System, http://unionfs.filesystems.org/
[20] AuFS, advanced multi layered unification filesystem, http://aufs.sourceforge.net/
[21] device mapper, http://www.sourceware.org/dm/
[22] DevOps, http://en.wikipedia.org/wiki/DevOps
[23] Andrew Filesystem, http://en.wikipedia.org/wiki/Andrew_File_System
[24] LXPLUS Service, http://information-technology.web.cern.ch/services/lxplus-service