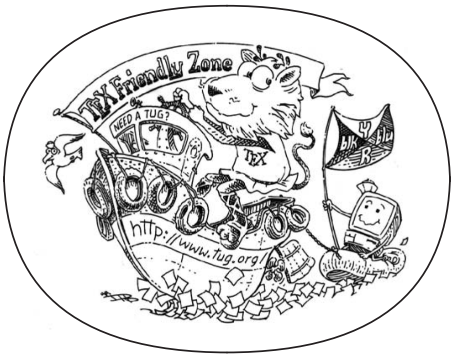
andré miede & ivo pletikosić

A CLASSIC THESIS STYLE

A CLASSIC THESIS STYLE

ANDRÉ MIEDE & IVO PLETIKOSIĆ



An Homage to The Elements of Typographic Style

June 2018 – classicthesis v4.6

Politecnico di Milano
V Facoltà di Ingegneria
Corso di laurea in Ingegneria delle Telecomunicazioni
Dipartimento di Elettronica e Informazione

TITOLO DELLA TESI

Relatore:	Prof. Nome COGNOME
Correlatore:	Ing. Nome COGNOME

Tesi di Laurea di:
Simone Mosciatti   Matr. 878758

Anno Accademico 2017-2018

Ohana means family.
Family means nobody gets left behind, or forgotten.

— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.

1939 – 2005

ABSTRACT

Short summary of the contents in English... a great guide by Kent Beck how to write good abstracts can be found here:

https://plg.uwaterloo.ca/~migod/research/beckOOPSLA.html

ZUSAMMENFASSUNG

Kurze Zusammenfassung des Inhaltes in deutscher Sprache...

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth [16]

ACKNOWLEDGMENTS

Put your acknowledgments here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio[1], Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, Jörg Weber, Henri Menke, Claus Lahiri, Clemens Niederberger, Stefano Bragaglia, Jörn Hees, Scott Lowe, Dave Howcroft, José M. Alcaide, David Carlisle, Ulrike Fischer, Hugues de Lassus, Csaba Hajdu, Dave Howcroft, and the whole LaTeX-community for support, ideas and some great software.

Regarding LyX: The LyX port was intially done by *Nicholas Mariette* in March 2009 and continued by *Ivo Pletikosić* in 2011. Thank you very much for your work and for the contributions to the original style.

1 Members of GuIT (Gruppo Italiano Utilizzatori di TeX e LaTeX)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

CERN, European Organization for Nuclear Research, (French: Conseil européen pour la recherche nucléaire) is an European research organization that operates the largest particle physics laboratory in the world.

Its mission is to:

- Provide a unique range of particle accelerator facilities that enable research at the forefront of human knowledge
- Perform world-class research in fundamental physics
- Unite people from all over the world to push the frontiers of sciences and technology, for the benefits of all.

It host the instrument of the 4 biggest physic collaboration:

- ALICE
- ATLAS
- CMS
- LHCb

CERN host also a plethora of smaller physical collaborations that benefits from the instruments, know how, network effect and services availables.

Between the services offered to its users the computing service is one of the most interesting. Indeed CERN host and manage one of the biggest computing data center used for public research.

An issues that affect the operations inside the data center is the provisioning of software on the computing servers.

A specialization of the same problem is the provisioning of containers images.

The general problem of software provisioning is been solved by the use of CernVM-FileSystem [2], a read only file-system that provides a scalable, reliable and low maintenance software distribution system.

This thesis will explore the problem of creating a suitable read-only file-system structure to provision containers images on computing nodes. We will provide a general read-only file-system structure and we will implement the proposed methodology on top of CVMFS.

This thesis is composed by several parts: The background will provide the necessary information on the CERN computing architecture (WLCG), then we will explore CVMFS and why it is a good fit for the

CERN computing architecture, the last part of the background will cover the integration between CVMFS and containers technologies. The state of the art will explore what alternatives are available for software distribution in general and for distribution of images. We will then define the problem that this thesis is trying to solve and few metrics of interest in our specific case. The methodology part will explain the details of the solution we propose for this specific problem. The implementation chapter will focus on how the proposed methodology is been put in practise. We will evaluate the result of the proposed methodology and implementation on the result part following the metrics that were previously proposed. Finally we will propose future work and enhancement to the implementation

2 BACKGROUND

In this chapter we are going to introduce the concept and the technologies that made this work possible. We will start introducing the Worldwide LHC Computing Grid (WLCG), which is the *collaboration* that provide the computing power necessary to the CERN mission. The dimension of the WLCG and its specific workload required and allowed a specific software distribution system, CernVM-FileSystem which is used to provision the machines on the WLCG.

Then we will introduce the concept of containers, a different way to solve the software distribution problem that is widely adopted in the industry. In particular, we will focus on Docker containers and the Docker images format. We will then explore Singularity, a container runtime capable of running containers images stored as simple folder in the filesystem. Finally we will explore the Docker `cvmfs/graphdriver`, a Docker plugin that allow to run Docker images whose content is available on a read-only file-system without the need of downloading or accessing the Docker standard images.

WLCG

The Worldwide LHC Computing Grid is an global collaboration of more than 170 data centers in 42 countries. The mission of the WLCG is to provide the computing resources to store, distribute and analyze the data generated by the operations of the LHC [3],[4].

The organization of the WLCG follows a hierarchical model, where each level of the hierarchy is called "Tier". The most central Tier is the Tier-0, which is hosted by CERN in the Geneva Area and in Budapest. There are 13 Tier-1 data centers with enough storage and computing capabilities to support the Grids operation around the clock. Tiers-1 are geographically distributed: 8 of them are in Europe, 3 in North America and the rest is in Asia. Finally, the Tier-2 data centers do not have strict requirements and are generally operated by research centers and universities [4]. A schematic representation of the architecture of the WLCG is provided on Figure 2.1.

The work of the WLCG is mostly divided in two big classes: analysis of the data from the LHC detectors and Monte Carlo simulations. The WLCG assume and support a batch computing paradigm. Analysis and simulations are split in smaller jobs that are distributed to different computing node that can work in parallel [4], [1].

Before to start each job it is necessary to install the software on the server. Unfortunately the amount of software potentially needed in

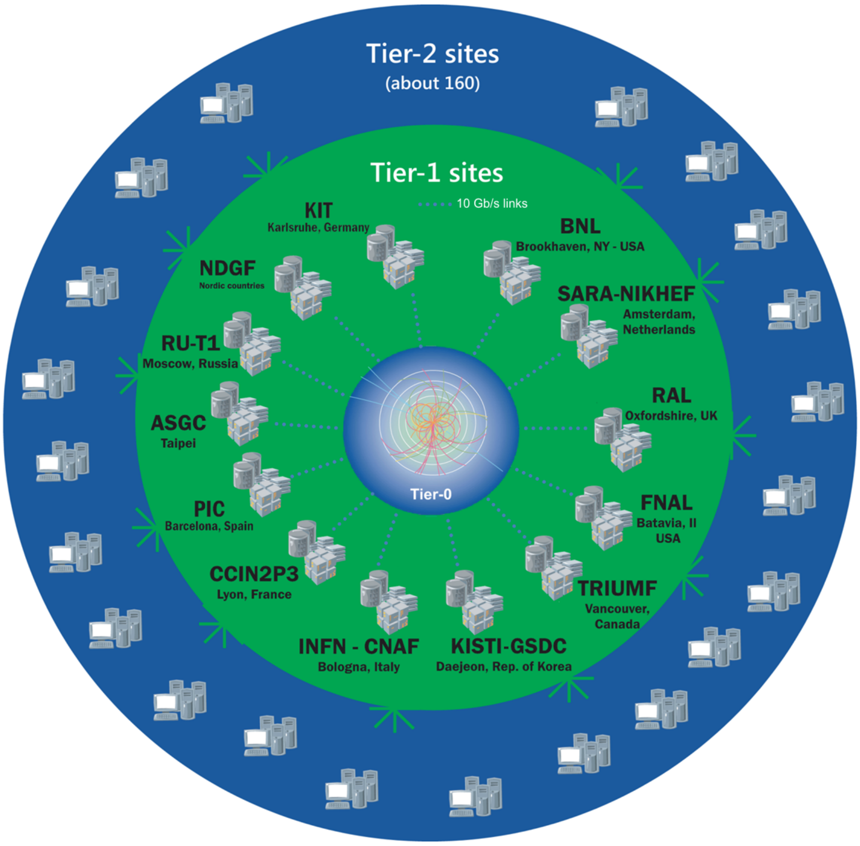


Figure 2.1: Schematic representation of the WLCG

each computing node and the velocity at which the software is updated can make the installation challenging. Moreover, simpler installation techniques that rely on packages managers are not applicable since they would put the centralized package managers themselves under too much load. Several solution have been proposed and used in the past, eventually it settled for the use of CernVM-FileSystem (CVMFS) [2].

CVMFS

This section will explore CernVM-FileSystem, we will start with an high level overview of CVMFS, then we will explore what happens when a file is requested.

CVMFS High Level Overview

CernVM-FileSystem [2] provides a scalable, reliable and low-maintenance software distribution system. It is implemented as a read-only POSIX file-system in user space exploiting FUSE (File-system in USErspace) [18] and standard web server technologies such as Apache or NGNIX.

Each running instance of CVMFS provides a read-only file-system that is denominated *repository*. At CERN different collaborations maintains different repositories, but all of them can be mounted from all the computing node in the WLCG. CVMFS is engineered to support repository of size on the order of the Terabyte with billions of files.

To save storage space files are addressed by their content (Content Addressable Storage), hence duplicated files will be stored only once.

In order to distribute software to geographically distant data centers and keep a low latency, CVMFS allows to cache content in different machines. This allow to host a cache server in each Tier of the WLCG. The use of caches fits perfectly with the Tiers models of the WLCG presented above. The Tier-0 host the main repository (Stratum-0), and the Tier-1 host the first level of cache (Stratum-1) and so on.

The content of the files are served using the HTTP protocol by a standard web server. The files are lazily downloaded only on the machine that need them and only when necessary.

In order to locate and request files from CVMFS the clients download the catalog, a simple SQLite database which describes a subtree of the whole file-system. The catalog contains all the metadata of files and directories, including owner, group, permission, and size. Moreover the catalog contains also the URL where to download the files.

A root catalog is available in a know path, and, if the file-system grows too large, the root catalog links to other sub-catalogs. The use of sub-catalogs allows to keep each catalog small improving the query time.

CVMFS Details

CVMFS is implemented using the Client-Server architecture. The server is responsible to manage the content of the repository and to expose it via HTTP API. The client is installed in the host machine and is responsible to expose the content of the repository to the users and it is implemented as a FUSE daemon which implements all the system calls necessary for a read-only file-system.

When a CVMFS file-system is mounted, it starts by reading a configuration file which describes each repository. The client then downloads a simple text file which points to the catalog of the repository. Once the catalog is downloaded the client has all the information necessary to start responding to the system calls performed by the user.

As an example, when the user requires a `stat` system call against a file, the client reads from the catalog all the information about the file like, size, permission, mode, etc.. and replies with them. Instead when the user requires to read from a file, the client first downloads the file from the server, stores it into a local cache and passes through each read operation to the local copy.

This approach allows to download only the file really required, since all the other system calls can be served by just reading from the catalog. However this implies that the reading latency from a file depends on the network latency. This strategy works very well if the reading latency of a file is not a major concern and if reading from the catalog is fast. However, if the catalog grows too big, then the queries become too slow.

To overcome this limitation the sub-catalogs were introduced. A sub-catalog is exactly like the normal catalog, but while the catalog refers to the whole file-system tree, a sub-catalog refer to a smaller sub tree of the file-system. In order to avoid confusion, we will refer to the root-catalog as the catalog that includes the root of the file-system and to sub-catalog to all the other catalogs in the file-system. The root-catalog, of course, embeds several sub-catalogs, and each sub-catalog can, recursively, embed another sub-catalog.

When is required to read information about a file, the client starts by looking into the root catalog and then it follows the sub-catalogs structure until it finds the required file.

CONTAINERS

While CERN solved its problems of software distribution with CVMFS the industry opted for a different approach: containers. Containers are a standard units of software that package up code and all its dependencies so that computer applications run quickly and reliably from one computing environment to another [15].1

In order to standardize containers and make the technology interoperable in 2015 the *Open Container Initiative (OCI)* was founded [7]. The OCI defined a standard format to use to pack containers into *images* [**oci-image-spec**], this format have been adopted by Docker and that is used in the *Docker Images*. A Docker image is an immutable set of tar files, where each tar file is called layer. Prior to run the container, each layer gets mounted one on top of the other to re-create the original environment where to run the application [5]. The content of each image is codified in a `json` file, the manifest, which provides the unique name of the image itself and which refers to each layer that compose the image by their unique identifiers. The unique identifier of both images and layers is the result of the function `hash256` of their content [6].

Docker images are distributed through Docker registries, simple HTTP servers that given the unique identifiers of a layer provide the layer itself, similarly, given the identifier of an image the registries provides its manifest [12].

Docker allow to associate a human readable identifier to each image, this name is composed by a namespace, which identifies the user or organization that created the image, a name, which identifies the image itself and a tag, which identifies the version of the image. These names are not immutable and are meant to be used just by humans to recognize and use the images [14]. The repository where the image is hosted, its namespace, its name and its tag create a hierarchical structure between the several images that is easy to navigate for humans.

Docker and the cvmfs/graphdriver plugin

Docker is a thin CLI layer on top of a Linux daemon, `dockerd` which is responsible to obtain the Docker images from the registries, mount the layers, manage the runtime of the image and allow communication of a Docker runtime with the host system [13].

Docker layers can be shared between different images, so the Docker daemon download them once and store them in the local machine for future use [10]. On average less than 7% of the content inside a Docker image is used during the runtime of the image [9]. The layers are distributed as a single tar file [5], hence a naive distribution of layers with CVMFS would not provide any advantage. The whole tar files would need to be downloaded erasing the advantages of using CVMFS.

A solution to this problem was introduced with the `cvmfs/graphdriver` plugin and the concept of "thin-images" [8]. In this work we are not interested in the internal of the `cvmfs/graphdriver` plugin, it will be sufficient to know that the Docker daemon can be enhance with the use of plugins [11] and that the plugin allows us to run what are called "thin-images".

Thin-images are Docker images that are created starting from standar (*fat*) Docker images. The content of a *thin-image* is only a simple `json` file whose content is the list of layers needed by the original *fat-image* and where to find them in the host file-system, we call this file the *recipe*. *Thin-images* can be executed only if the Docker daemon have enabled the `cvmfs/graphdriver` plugin, indeed the plugin is able to read the content of the thin-image, mount the original layers inside the container, and finally execute the application [8].

If the files necessary to run the *thin-images* are distributed by CVMFS, it is possible to run docker containers without downloading any unnecessary content.

Moreover the `cvmfs/graphdriver` plugin is able to run standard (*fat*) Docker images, hence conventional Docker images are supported seamlessly [8].

On figure 2.2 we show how the plugin decides how to mount the container file system. If it detects that the image is a standard *fat* Docker image it run it as a standard image, if it detect that is a *thin* image it mounts the layers from the CVMFS read-only file system.

The downside of this approach is that a "thin-image" can be downloaded, store in the local storage and successfully run using the content provide by CVMFS. On a second moment the images can be update, uploaded newly on the registry and the old content in CVMFS is deleted to host the new content of the image. If now the user try to run the same images, it will not download it again from the registry but it will use the image already cached, referring to content not available anymore on CVMFS, hence it will fail to run.

Singularity

Singularity [21] is another container runtime, it provides its own image format but is capable to run standard Docker images [20] as well. Moreover it is capable of running containers, also Docker containers, directly from a directory containing the unpacked container file-system itself [19].

Given the Singularity capability to run Docker containers directly from a directory containing the unpacked Docker image file-system its integration with CVMFS is simpler that the one with Docker itself. Indeed it is sufficient to host the directory containing the unpacked file-system of the Docker image in CVMFS.

PROBLEM DEFINITION

We have introduce how CERN has overcome the challenges of software distribution using CVMFS. Unfortunately this is not enough

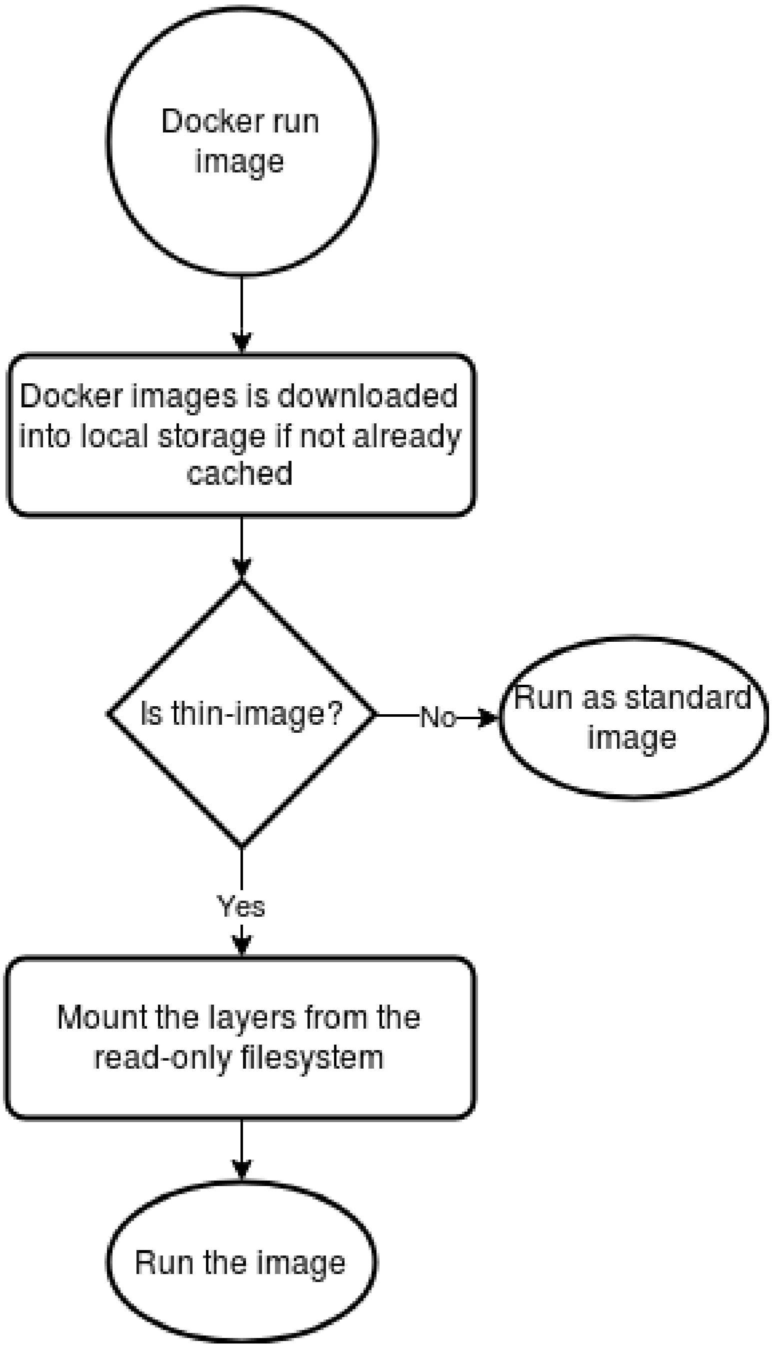


Figure 2.2: Decision proces for running docker thin images

since run-time dependencies between software components can break application that instead are running perfectly fine in a different environment.

Since containers pack up all their runtime dependencies in a standard environment they are a suitable solution for this problem. However containers are distributed as big tar files which is against the design and working principle of CVMFS.

Both CVMFS and containers aims to solve the similar problem of server provisioning, however they made different trade-offs. CVMFS opted for an efficient distribution of the content, however this makes difficult and inconvenient to pack all the runtime dependencies of a particular application. Containers opted to make each application self-contained so that the application can run reliably also on different computer environment, however they loose efficiency in the distribution of their content.

We don't believe that efficient content distribution and application containerization are contrasting goals, hence in this thesis will attack the problem of how to efficiently distribute the content of containers.

In Chapter 5 we will present a generic read-only file system structure suitable to host container content. In Chapter 6 we will show how we implemented such read-only file system structure on CVMFS in order to distribute containers content. Finally we will show the result of such work on Chapter 7. Possible enhancement to this work are finally exposed on Chapter 8.

3 STATE OF THE ART

In this chapter we are going to explore what tools and methodologies are already available and know to provision software on a computing cluster. We will explore what is used in the industry as well as what was developed specifically inside the CERN environment We will also focus on how those system works with container technology, in this particular chapter, we will focus just on Docker since Singularity is not widely used in the industry and none of the following tools support it.

ANSIBLE

Ansible is a provisioning system based on the concept of playbooks and inventory.

Playbooks are files that describe the desired final status of a server. It is include software to be installed, configuration files to be created and setted, security policies, status of services and it may also include docker images to be downloaded.

The inventory is a list of the nodes where we wish to reach the final status described in the playbook.

The Ansible approach is declarative, indeed, we describe what we want and where we want it and then we leave it to an internal engine to reach the desired state.

Ansible can also be categorized as a non-intrusive provisioning system, it works using an a remote connection (SSH) to the nodes it is managing without relieing on any software been installed on the node itself.

PUPPET

Puppet is another declarative provisioning system.

Similarly to Ansible, also in Puppet, we describe the desired status of the servers. The big difference with Ansible is that Puppet needs a daemon, called "agent", running on the provisioned servers.

The agent ensure that the desired configuration is keep in spite of manual changes on the machine.

The agent is as well responsible to makes all the necessary changes to the system.

KUBERNETES

Kubernetes (K8S) is a container orchestration system based on docker for computing clusters.

A central master is responsible for managing the cluster. It coordinates all the activities such as scheduling containers or rolling out updates. The worker nodes simply execute the commands from the master.

PACKAGE MANAGERS

Packages managers are the standard way to install software on most linux distribution.

A package is conceptually composed by an archive that contains the actual files to install on the system and by a configuration that describe where to install each file.

Moreover a package describe its dependencies, dependencies that are recursively installed by the package manager.

ALICE SOFTWARE INSTALLATION SYSTEM

In order to ease the load on the network the Alice collaboration opted to distribute packages using the BitTorrent protocol. The installation is still based on standard linux packages, but the use of a peer to peer distribution mechanism avoided to overload the central package manager. However, it still relies on a central authoritative BitTorrent tracker hosted at CERN itself.

DIFFERENCE WITH CVMFS

All the above tools used for installing software relies on installing the software on the machine before that it is actually request, doing so, it normally moves also files that are not strictly necessary for every computation, wasting

CVMFS on the other side avoid to move any software on the machine if it is not necessary. The downloading of a file is defer to when it is necessary (on the OPEN syscall) hence only the file strictly necessary are downloaded.

It results in an increase of latency when starting an application but the bandwidth consumption is keep to the minimun.

4 PROBLEM DEFINITION

In the Background on chapter 2 we addressed how CERN has overcome the challenge of software distributions using CVMFS. Unfortunately this is not enough since run-time dependencies between software components can break application that instead are running perfectly fine in a different environment.

A possible solution to this problem is the use of container technology and we explore how both Singularity and Docker with the `cvmfs/graphdriver` plugin are capable to run Docker images hosted in CVMFS avoiding to download unnecessary from remote host saving bandwidth.

In this work we aim to introduce a read-only file-system structure implementable in CVMFS that will allow us to bridge together the efficient content distribution provide by CVMFS with the encapsulation of runtime dependencies provide by container technologies.

This work will focus on a specific container technology, Docker. Indeed we focus on describe a file-system structure suitable to run Docker images using Singularity and the thin-images Docker graphdriver plugin.

The proposed file-system structure aims to the following goals:

- Minimize the amount of space required
- Minimize the start up time of containers
- Minimize the time necessary to add a new docker image into the file-system
- Minimize the number of files in the CVMFS sub-catalogs
- Minimize the complexity of managing the files-ystem itself
- Provide a great usability for the users.

Some of those goal can be measured to asses if we were successful or not, in particular we are going to:

- Compare the amount of space required by the CVMFS repository versus the amount of space required for each layer.
- Compare the start-up time of containers hosted in CVMFS versus the start-up time of containers not hosted on CVMFS
- Provide the distribution of number of files in each sub-catalog

- Measure the complexity of managing the file-system using as a proxy the cyclomatic complexity of the software that actually create and manage the file-system.

Unfortunately we didn't find usable proxy for measuring the usability of the file-system structure nor it make sense to measure the time necessary to add a new Docker image into the file-system since it depends on too many variable to provide a useful measure.

5 METHODOLOGY

In this chapter we are going to proposed a read-only file-system structure for running Docker images using both Singularity and the docker thin-images plugin. We focus on Singularity because is a widely deployed system in HPC and on docker thin-images because it allow the user to use the Docker infrastructure to run images whose content is distributed with a read only file-system.

The section 5.1 will provide a high level overview of the proposed file-system structure.

The section 5.2 will analyze the file-system structure to host the Docker images to run with Singularity. We will explain how we created a hierarchical structure similar to the one of the docker registries while keeping the repository maintainable and without putting too much pressure in the sub-catalog system.

The section 5.3 will analyze how we store the content used by the thin-images Docker plugin. Structural similarities between Docker images and Docker layers will drive us to adopt a very similar solution to avoid stressing the sub-catalog system. The sharing of layers between different docker images will allow us to avoid repeating work, but it will introduce difficulties during the deletion of the image itself that we will solve using a reference count system.

Finally the section 5.4 will show how we keep track of the images already present in the file-system.

HIGH LEVEL OVERVIEW OF THE PROPOSED FILE SYSTEM

In this section we are providing an high level view of the proposed file-system structure, the details for the Singularity runtime will follow on section 5.2 while the details for the `cvmfs/graphdriver` plugin will follow on section 5.3.

The propose file-system provide two main structure. A non-hidden set of directories that host the unpacked images to run with Singularity and a hidden folder to host the layers used by the `cvmfs/graphdriver` Docker plugin. Along with these structure another hidden directory will contains metadata information about the Docker images already in the repository.

The directories that host the unpacked Docker images have the same hierarchical structure of the Docker structure as mentioned in 2.3, hence the first directory is the name of the registry that host the image, it follow the namespace which refers to the user of the

```
/cvmfs/unpacked.cern.ch
 ├── registry.hub.docker.com
 │    └── library
 │         └── centos:centos7
 ├── .layers/
 └── .metadata/
```

Figure 5.1: High level visualization of the proposed file-system

organization responsible for the image and the last level is the image name along with the tag.

The directory that host the layers of the Docker images can conceptually be a flat structure simply containing the layers each identified by its digest. However a simple flat structure will put too much pressure in the catalog system so we will aggregate layers that share the same digest prefix and create a sub-catalog for each aggregation.

The hidden metadata folder will follow the same hierarchical structure of Docker images that allow to quickly locate the metadata information of an image given just the name of the image itself.

SINGULARITY IMAGES

To run Docker images using Singularity is sufficient to start the singularity executable providing as input the directory where the image is been unpacked. In this section we are going to show how we structure the file-system in a way that allow users to easily discover and run unpacked docker images using Singularity while keeping the file-system easy to maintain.

As mentioned in 2.3 docker images have a hierarchical structure. The first level of the hierarchy is the docker registry where the image is hosted. The most common registries in our case are the official docker hub [1] and the CERN internal registry [2].

The second level in the hierarchical structure is the namespace of the docker image. If the image is one of the official docker images it will be the standard namespace: "library". In all the other cases, the namespace will be the same as the original docker image. For example for the images belonging to the ATLAS collaboration we use the namespace `atlas`.

The last level is the name of the image itself together with the tag of such image, separated by a colon (`:`). We decided to avoid yet another level containing just the tags. Indeed there are relatively few tags for each image and adding another level of indirection would have made it harder to explore the file-system. Moreover, we decided to use the colon because it is the same character used in the docker registries

1 `registry.hub.docker.com`
2 `gitlab-registry.cern.ch`

```
/cvmfs/unpacked.cern.ch
 └─ registry.hub.docker.com
     ├─ library
     │   ├─ centos:centos7 -> /cvmfs/unpacked.cern.ch/.flat/75/75835...0c4ab6d
     │   ├─ centos:latest -> /cvmfs/unpacked.cern.ch/.flat/75/75835...0c4ab6d
     │   ├─ debian:stable -> /cvmfs/unpacked.cern.ch/.flat/a4/a4274...ba594cb
     │   ├─ gcc:latest -> /cvmfs/unpacked.cern.ch/.flat/ce/ceccd...a75dd28
     │   ├─ openjdk:9 -> /cvmfs/unpacked.cern.ch/.flat/5a/5adaf...5344d70
     │   ├─ python:2.7 -> /cvmfs/unpacked.cern.ch/.flat/3c/3c43a...0e7c9d4
     │   └─ python:3.4 -> /cvmfs/unpacked.cern.ch/.flat/43/43953...ed73435
     └─ efajardo
         └─ docker-cms:tensorflow ->
            /cvmfs/unpacked.cern.ch/.flat/2d/2d5b4...97d44fc
```

Figure 5.2: Visualization of the Filesystem structure, the arrows indicate symbolic links

between the images and the tag and it is immediately recognizable by users.

While this structure is user friendly, it makes the maintenance of the repository complex.

The tags used in each image are not immutable, hence, without continuous maintenance, it may happen that the images stored inside the file-system are not up to date making difficult for the user to know what version of the software is being run. However with the described structure, it would be extremely complex to detect if an image is up to date or if it needs further updates.

To work around this issues we exploited the fact that each image is uniquely identified by its digest. Indeed we decided to store the real content of the images in an hidden folder that embed the digest itself while preserving the structure presented above using symbolic links.

We show the directory structure of the file-system on figure 5.2.

The folder that contains the real content of a Singularity images are all below the standard subdirectory `.flat/`. The name `.flat/` was chosen to make it clear that only flatted file systems are stored in there.

Embedding the digest in the name of the folder allows to immediately find the location of an image, which is useful when an image become obsolete and need to be deleted from the file-system.

From a theoretical point of view it would be sufficient to store the whole content of the Singularity images in the folder `.flat/$image_digest`. However, from a practical point of view this would create too much content in a single folder putting too much pressure in the CVMFS sub-catalog system.

To overcome this issue we decided to create a fixed number of "super-directories" where we placed the unpacked folder of the images. To easily locate each unpacked folder in the super-directories we decided

```
/cvmfs/unpacked.cern.ch/.flat
├── 0c
│   └── 0cbf37812bff083eb2325468c10aaf82011527c049d66106c3c74298ed239aaf
├── 2c
│   └── 2cc378c061f7b3e8d9096728eb75722a89f31fb3f3117ed10c66cc2f4b8ab281
├── 5a
│   └── 5adaf00da2a3cf6b611e7c850778fad3dc62c548864706b822b5f3ce65344d70
└── ea
    ├── ea4c82dcd15a33e3e9c4c37050def20476856a08e59526fbe533cc4e98387e39
    └── eadfca9546a132104b8bdb6b76952c6e5d412301704b7bc94e9176bcc5dda0fe
```

Figure 5.3: Visualization of the "super directories" in the ".flat" subdirectory

to call each super-directory as the prefix of the digest of the images it is containing. Since the digest is an hexadecimal string this approach provides us with $16 \times 16 = 256$ fixed super-directories inside the `.flat/` directory, each of which will contain only the content of the images whose digest start with those 2 specific bytes.

On figure 5.3 we can see that "0c", "2c", ..., "ea" are all "super-directories" and each one contains only the file-systems that start with "0c", "2c", ..., "ea" respectively. Note the case of "ea" that contains file-systems of multiple images whose digest start with "ea".

However, to relieve pressure from the catalog system is not sufficient to simply aggregate the images into "super-directories", we also need to create a sub-catalog for each "super-directory." Moreover, since each image can contains itself a lot of files we decide to create a sub-catalog also for each unpacked image.

Another positive side-effect of the use of symbolic links is that symbolic links manipulation is defined as atomic in the POSIX standard.

The use of "super-directories" is necessary for limits in the implementation of CVMFS and they are not necessary on an abstract read-only file-system.

DOCKER THIN IMAGES

While for running docker images using Singularity it is sufficient to have the image unpacked in a simple directory running docker containers using the thin-images plugin requires a more complex set up. As explained in 2.3.1 the recipe of the docker thin-image contains the path of the directories where each layer of the original docker image is hosted, those directories will be mounted by the docker plugin.

All the docker layers are stored under a common subdirectory of the file-system, the `.layers/` directory.

Since the sub-tree of the file-system used by the Docker thin-images is used only by the Docker plugin we don't need to create a human-friendly structure like we did for the Singularity sub-tree.

Like docker images also the docker layers are identified by an unique digest, and similarly to the docker images, store all of them in a single directory will put too much pressure in the CVMFS sub-catalog system, hence we follow the exact same model used for storing the unpacked images also for the layers creating 216 super-directories.

A big advantage of the use of layers over flat images, is that layers can be shared by multiple images.

The sharing of layers allow us to avoid re-doing work that is already been done, in particular if a later is already in the file-system it will not be added again. On the other hand it makes more complex removing an image since it is necessary to remove each layer that compose the image, but some layer may be shared between images.

Removing layers has the important implication that once the layer is removed every thin image that relies on it won't work anymore. However those thin-images could be stored on the client side where we don't have any access. Please refer to the figure **??** on page **??**

To do not disrupt the user workflow while keeping the repository to a manageable size we consider several option:

1. Never remove layers
2. Remove layers as soon as possible
3. Provide a grace period before finally removing the layer

The option to never remove layers is impractical since the size of the file-system will grow unbounded.

Remove layers as soon as possible is not desiderable, even running computation could be broken by this policy and the users have no way to deal with this possibility but retrying the whole computation.

The last option is the most sensible and better suited for our use case, and so it is the one that we implement, this gave users the possibility to:

1. Complete their computation
2. Update the local images in order to always run stable containers

In order to know which layer to delete from the file-system we store a reference that map each layer to the images that use the layer itself.

These references are stored as metadata in a simple `.json` file. We store one of these reference file for each layer in the file-system.

Anytime a new image is added to the file-system we update the several references files, adding for each layer in the image, a reference to the image itself.

When we decide to remove an image, for any layer we check that it is used only by the image we want to remove, if this is the case, we remove the layer, if it is not the case we just remove the reference of the image.

Listing 5.1: Algorithm to add an image reference to the layer metadata

```
Function AddReferenceToImage
        Pass In: LayerReference, ImageReference
        ReferenceFile := FindReferenceFile(LayerReference)
        if ReferenceFile exist
                References := LoadReferenceFromFile RefereceFile
                Add ImageReference to References
                Overwrite References to ReferenceFile
        else
                References = ImageReferences
                Write References to ReferenceFile
        endif
EndFunction
```

Listing 5.2: Algorithm to remove an image from the file-system

```
Function RemoveLayer
        Pass In: LayerReference, ImageReference
        ReferenceFile := FindReferenceFile(LayerReference)
        References := LoadReferenceFromFile RefereceFile
        Remove ImageReference from References
        if size References == 0
                Remove Layer
        else
                Overwrite References to ReferenceFile
        endif
EndFunction
```

```
/cvmfs/unpacked.cern.ch/.layers
└─ 21
   ├─ 2100d...d7b7dbf
   │  ├─ .metadata
   │  │  └─ origin.json ..........................the reference file
   │  └─ layerfs/ ... This directory contains
   │                  the file-system of the
   │                  layer itself and is the
   │                  one that appears in the
   │                  recipe of the thin-image
   └─ 217f7...601e9e7
      ├─ .metadata
      │  └─ origin.json
      └─ layerfs/
└─ c3
   ├─ c300b...4190f83
   │  ├─ .metadata
   │  │  └─ origin.json
   │  └─ layerfs/
   └─ c3683...53a1c45
      ├─ .metadata
      │  └─ origin.json
      └─ layerfs/
```

Figure 5.4: Complete visualization of the `.flat` directory

In order to store both the metadata information about the layers (in particular the "reference" file mentioned above) and the actual file-system of the layer an additional directory structure is used. Below the directory called as the digest of the layer there are two more directories:

1. `layerfs/` directory that actually store the content of the layer
2. `.metadata/` directory that stores the references to the image in a simple JSON encoded file, "origin.json"

Of course, the recipe of the thin images is not concerned at all with the content of the `.metadata/` directory. Hence the recipe files points directly to the `layerfs/` directory.

The complete structure for storing docker images is the one showed in 5.4

KEEPING TRACK OF THE WORK ALREADY DONE

To avoid to perform duplicated work is necessary to keep track of which image is already been added to the file-system. The same information may be used by the users to know exactly what images is hosted in the file-system.

```
/cvmfs/unpacked.cern.ch/.metadata
└── registry.hub.docker.com
    ├── library
    │   ├── python:latest
    │   │   └── manifest.json ........................the manifest file
    │   ├── r-base:latest
    │   │   └── manifest.json
    │   └── julia:latest
    │       └── manifest.json
    └── atlas
        └── athena:latest
            └── manifest.json
```

Figure 5.5: Structure of the `.metadata/` directory

In order to know which image is already been converted we need to uniquely identify each image, as already mention, using the combination of image name and tag is not enough, since the tag are mutable. Hence we rely on the digest of the image.

The information about each image is stored into another top-level hidden directory, `.metadata/`.

Inside the `.metadata/` folder we have others directories, one for each hosted image. Inside those directories there is a single file, `manifest.json` that store the manifest of the image itself.

As already mentioned in **??** the manifest contains the digest of the image itself. Comparing the manifest stored in the file-system with the manifest downloaded from the docker registries is possible to understand if the image should be updated or nor.

The structure of the `.metadata/` folder is show in figure 5.5.

CLOSING REMARKS

In this chapter we have introduced a file-system structure suitable to host docker images that can be run using both Singularity and the docker thin-images plugin.

We started by storing the unpacked images used by Singularity in a hierarchical structure that recall the one of the Docker registries to enhance the discoverability of the images itself. This approach however would have make difficult to maintain the repository since we would not know the version of each image unpacked. We overcome this issue storing the real unpacked images in a hidden directory that embed the digest of the image itself and using symbolic links to preserve the hierarchical structure. Too many unpacked images stored under the same directories however would have put too much pressure on the catalog system of CVMFS, hence we adopted the use of super-directories.

On the second part we analyzed how to store the layers used by the docker thin-image plugin. A single layer can be used by multiple images this allowed us to avoid repeating work but at the same time it makes more complex removing an image from the file-system. We decided to keep a reference count to know when is safe to actually delete a layer. The same issue of too many files under the same sub-catalog arose also for storing the layers, we used the same approach used for the unpacked images based on the use of super-directories.

The last section explored how we keep track of exactly which image is already store in the file-system storing the image catalog in a hidden subdirectory.

6 IMPLEMENTATION

In this chapter we are going to explore how the ingestion inside CVMFS is been implemented. At first we will describe the write interface of CVMFS, then we are going to talk how we unpack Docker images to use them with Singularity. We will move on to describe how we ingest Docker *fat images* while transforming them into Docker *thin-images* suitable to be used with the *cvmfs/graphdriver* plugin, moreover we will explore the custom modification done to CVMFS in order to accommodate the needs of layer ingestion. We will then explore how all the bookkeeping of images and layers is manages. Then we are going to understand how the images are removed from the file system. Finally we will show the interface given to the administrator.

All the work presented in this chapter is implemented in the *repository-manager*, a command line utility written in the Go(lang) language [17].

CVMFS WRITE INTERFACE

As mentioned in 2.2.2 CVMFS is implemented with a Client-Server architecture, while the client is strictly read-only, the server does provide a write interface. What we write in the server is what the client is then able to read. The write interface of CVMFS is a transactional interface, hence is possible to open a transaction, modify the file system and then either publish the modification or abort the transaction. Those actions are carried out respectively by the command 'cvmfs_server transaction', 'cvmfs_server publish' and 'cvmfs_server abort'. When a transaction is open the server file system is a standard writable linux file system, hence is possible to modify it using the standard POSIX API, Linux commands or even graphical file explores. Moreover is possible to test locally the new file system without actually commit the changes. Of course these actions are not available to the clients that have access only to a read-only interface.

Finally is important to keep in mind that only a single transaction can be open at any given time, CVMFS will refuse to open a second transaction.

SINGULARITY INGESTION

The step to ingest a singularity images are pretty straightforward. Initially the image is downloaded from the remote registry and stored in a temporary area. The download is carried out by Singularity itself, Listing 6.1: Algorithm to unpack a Docker image with Singularity and ingest it into CVMFS

```
Function UnpackAndIngestDockerImage
        Pass In: DockerImageName

        Digest := RetrieveDigestFromImageName DockerImageName
        TemporaryDirectory := CreateTempDirectory
        UnpackDockerImageWithSingularity TemporaryDirectory

        StartCVMFSTransaction
            FlatDir := CreateFlatDirectory Digest
            MoveFrom TemporaryDirectory Into FlatDir
        CommitCVMFSTransaction

        HumanReadableName := GetDirectoryFromDockerImageName
            DockerImageName

        StartCVMFSTransaction
            CreateDirectory HumanReadableName
            CreatSymlinkFrom HumanReadableName To FlatDir
        CommitCVMFSTransaction

EndFunction
```

in order to minimize the possibilities of inconsistency or of error. Once the download complete successfully and the unpacked container file-system is in the local file-system we start the real ingestion phase.

The first step of the ingestion is to open a transaction in CVMFS. Once we open the transaction we copy the temporary directory into the CVMFS filesystem under the ".flat/" directory. Then we commit the first transaction. A second transaction takes care of creating the symbolic link as describe in 5.2.

This few step are sufficient to make the Singularity images available through CVMFS. The listing 6.1 show the details of the algorithm.

DOCKER INGESTION

Converting Docker *fat images* into Docker *thin images* is a more complex project than simply make the unpacked image available to use with Singularity. We will explore all the details of this process in this section.

In the first part we will introduce how CVMFS is able to directly ingest tar file which is format used to distribute Docker images as mention in **??**. The we will explore how starting from the Docker manifest we add the layers to the CVMFS file system, then we show how we create the *thin image* itself.

Moreover we also upload the Docker *thin image* to a Docker registry.

CVMFS Ingestion of Tarball

As mention in 2.3 Docker layers are distributed as tar files. In order to support the use case of ingesting Docker layer we decide to add a new command to CVMFS ‘cvmfs_server ingest‘. The ingest command takes as input a tar file and a directory inside the CVMFS file system and extracts all the files and directory in the tar file into the directory provide as input. This command implicitly open and commit a CVMFS transaction, hence is possible to have only a single concurrent ingestion.

Docker Ingestion Algorithm

The first step of the algorithm is to retrieve the manifest of the Docker images from the Docker registry, as soon as we have the manifest the algorithm checks if the specific images is already been successfully ingested into the file system. This check happens using the metadata stored in the *.metadata* directory as mention in 5.4. The check consist in a simple comparison between the digest of the manifest of Docker images just downloaded and the digest of the images already stored in the *.metadata* folder. If the images is already in the file system the algorithm terminate.

The next step is to ingest every single layer of the Docker images into the CVMFS file system. As previously we check if the layers already exists in the file system itself. Since the layers are stored under a path that embeds theirs own digest 5.3, checking if a layer is already in the file system consist is simply checking if the folder where we would ingest the layer already exists or now. If the layer already exists we move to the next layer of the image.

The ingestion of a layer follows a similar procedure of the ingestion of an unpacked images, the layer is first downloaded into a temporary directory and then is ingested using the ‘cvmfs_server ingest‘ command. Another option could have been to avoid storing the layer in the temporary directory and simply let the ‘cvmfs_serve ingest‘ command read the content of tar file from `STDIN`, we decided against this approach since a no-negligible amount of times the download of the layer fails in the middle wasting all the work already done by the ’ingest‘ command.

If the ingestion of any of the layers fails we stop the whole algorithm and we rely on retries from the administrator in order to have the Docker image served on CVMFS.

After all the layers have been successfully ingested the next step of the algorithm is the creation of the Docker *thin image*. The Docker *thin image* is a standard Docker image which content is a single file `thin.json` that contains the set of layers to mount before to start the container itself 2.3.1. To create this image is sufficient to encode the location of layers into a `.json` file and then pack this file into a standard Docker image. The Docker *thin image* is then uploaded into the Docker registry.

The last step of the algorithm is to store the metadata information about the image just ingested in order to avoid to repeat work already done. This is done simply storing the manifest of the docker image in the *.metadata* folder following the schema presented in 5.4.

GARBAGE COLLECTION OF IMAGES

We have only described how we add new images to the CVMFS file system, however updating an image is quite common, especially if the images are referred by mutable tag such as "latest" which actually represent the latest version of a particular application.

During the update of an image we avoid to immediately delete the files from the CVMFS repository, as mentioned in 5.3 this could cause disruption of service for users.

Instead we keep track of all the images that are not necessary anymore in a specific file, the `remove-schedule.json` file which is stored in the hidden directory *.metadata/* just below the main root of the CVMFS file system. The `remove-schedule.json` files contains a collections of the manifest of all the images we are not interested in anymore.

When is time to actually delete all the old images we scan the `remove-schedule.json` file and we carry out the actual removal of the images.

The removal of a singularity image is quite simple, indeed, is sufficient to remove the whole directory.

Removing the layers of the docker images is more complex. At first we need to identify all the layers that we need to check. This is simple since this information is stored in the manifest itself which is stored in the `remove-schedule.json` file.

Then, for each layer we obtain the list of images that need the layer itself. From that list we remove the image we are eliminating from the file system. If the list is now empty we proceed to remove also that specific layer from the file system.

ADMINISTRATOR INTERFACE

In order to store images in the CVMFS file system is necessary to know what image store, in which repository store it and how to call the respective thin-image. We decide to call the triplet `<Input image, CVMFS Repository, Output image>` a *wish*.

To express a list of those wishes we opted for a simple YAML file that store a specialization of a generic "wish list". In the YAML file we specify a list of Input images, only a single CVMFS repository and a

`$(scheme)`	`https`
`$(registry)`	`registry.hub.docker.com`
`$(repository)`	`library/centos`
`$(reference)`	`6`
`$(image)`	`librari/centos:6`

Table 6.1: Available placeholders and their application to the image `https://registry.hub.docker.com/library/centos:centos6`

single syntactical transformation for the Output images. An example of this specific *wish list* is show on Listing 6.2.

Listing 6.2: Example of a small *wish-list*

```
version: 1
user: smosciat
cvmfs_repo: 'thin.osg.cern.ch'
output_format: '$(scheme)://gitlab-registry.cern.ch/smosciat/thin
    -osg/$(image)'
input:
    - 'https://registry.hub.docker.com/library/centos:latest'
    - 'https://registry.hub.docker.com/library/centos:centos6'
    - 'https://registry.hub.docker.com/library/centos:centos7'
    - 'https://registry.hub.docker.com/library/debian:latest'
    - 'https://registry.hub.docker.com/library/debian:stable'
    - 'https://registry.hub.docker.com/library/debian:testing'
    - 'https://registry.hub.docker.com/library/debian:unstable'
    - 'https://registry.hub.docker.com/library/ubuntu:latest'
    - 'https://registry.hub.docker.com/library/fedora:latest'
    - 'https://registry.hub.docker.com/library/python:latest'
    - 'https://registry.hub.docker.com/library/python:2.7'
    - 'https://registry.hub.docker.com/library/python:3.4'
    - 'https://registry.hub.docker.com/library/openjdk:latest'
    - 'https://registry.hub.docker.com/library/openjdk:8'
    - 'https://registry.hub.docker.com/library/openjdk:9'
    - 'https://registry.hub.docker.com/library/gcc:latest'
    - 'https://registry.hub.docker.com/library/r-base:latest'
    - 'https://registry.hub.docker.com/continuumio/anaconda:
      latest'
    - 'https://registry.hub.docker.com/bbockelm/cms:rhel6'
    - 'https://registry.hub.docker.com/bbockelm/cms:rhel7'
    - 'https://registry.hub.docker.com/efajardo/docker-cms:
      tensorflow'
    - 'https://registry.hub.docker.com/lincolnbryant/atlas-wn:
      latest'
```

The syntactical transformation depends on the input image and is applied to obtain the final name of the output image. It simply replace the placeholder `$(PLACEHOLDER)` on the `Output Image` with respective item of the `Input Image`. We show a reference table on Table 6.1.

The use of a simple YAML file to express the desired content of the file system brings several benefits. Since the wish list can be hosted on a version-control system like Github or Gitlab can be used to host and keep track of the several version of the *wish-list*. Moreover it enable a Pull-Request based approach to change the wish list itself. Users who wish a new image to be added to the repository can simply make a pull request adding their image to the wish list. The administrator of the system act as a gatekeeper, inspect the image that is been required to be add and decide if add the image or not.

THE *repository-manager* COMMAND LINE TOOL

All the work presented in this chapter is been implemented in the *repository-manager* [17] command line tool. The tool provide to the administrator of the repository just two main command. The *convert* command and the *garbage-collect* command.

The *convert* command takes as input a *wish list* as show in 6.2 and follow all the procedure to add the unpacked Docker images to be used with Singularity to the repository as well as creating the several Docker *thin images* and push them into the Docker registries. Moreover, whenever it need to delete an image it add it to the `remove-schedule.json` file.

The actual removing of the images is carried out by the *garbage-collect* command which reads the `remove-schedule.json` files and carried out the removal as describe in 6.4.

Finally another command of the utility is wort to mention, the *loop* command simply execute the *convert* command in an infinite loop, each time reading again the *wish list* file. This allow the administrator to run the conversion in the back ground and set up a periodic job that update the *wish list* file downloading it from a revision control system.

7 RESULTS

In this section we explore the result of this works.

We mentioned 4 metrics that we were considering for this work:

- Minimize the amount of space required
- Minimize the start up time of containers
- Limit the number of files in each CVMFS catalogs
- Minimize the complexity of managing the filesystem itself

All the measurement are going to be done against a CVMFS repository containing images necessary for standard HEP work.

In order to quantify the amount of space required we are simply going to measure the amount of space that the repository uses, we will compare this figure with the amount of data that the repository provides and with a simple summation over the size of the layers stored in the repository.

The startup time of a container is greatly influenced by the cache layer in all cases, either if we serve the content with CVMFS or if the content is already cached in the hosting machine.

We will measure the startup time of all kind of technologies with and without cache a significant number of times, the measurement are made inside the CERN data center where we assume a stable and reliable internet connection.

The amount of files in each CVMFS catalog is a simple measurement, since the amount of catalogs is rather big we will synthetize this measurement.

To measure the complexity of managing the filesystem we are going to measure the cyclomatic complexity of the software that we use to manage it.

SPACE REQUIREMENT

By default CVMFS stores all its content in `/srv`, hence the most reliable way to obtain the size of the repository is to analyze the storage used under `/srv`. The storage required for the whole repository is of 27G calculate with the `df` unix utility as show in 7.1.

	.flat	.layers
Size	28	27

Table 7.1: Apparent size in GB of the two folder *.layers* and *.flat*

		Cache		No Cache	
		Avg	STD	Avg	STD
Thin-Image on CVMFS	Singularity	10.31	4.38	29.13	4.02
	Docker	96.74	5.40	180.02	35.13
Naive	Singularity	7.38	2.07	1884.76	366.84
	Docker	95.26	5.29	1279.21	168.99

Table 7.2: Benchmark of startup time of a containers, the first number is the average while the second is the standard deviation. The units are in hundredths of seconds. $n = 100$

Listing 7.1: Storage require to store the whole repository

```
>> df -h /srv
Filesystem      Size  Used Avail Use% Mounted on
/dev/vdb        197G   27G  161G  15% /srv # <-- I believe is too
     much, most likely an error
```

We than obtain the size of the two hidden folder *.layers* and *.flat* using the 'cvmfs_server list-catalogs' command which provides us with the number of files in each catalog and the number of bytes that each catalog manage.

We can see that the Content-Addressable-Storage used by CVMFS helps a lot by reducing the amount of space required to host the repository by half, which make sense since the *.layers* and the *.flat* contains the exact same files.

CONTAINER STARTUP TIME

This Section explores the startup time of the containers host in the CVMFS file-system.

We will show the average time of starting a container, start the python shell inside the container and executing the exit command inside the shell.

We decide for this simulation because it requires to have access to the python interpreter in the container, which is a executable of not negligible size.

We propose 8 scenarios that we synthesize in the Table 7.2 below.

We ran the benchmark 100 times, and we collect the startup time with the unix utility `time`. The code that we used to obtain the number is available on the listing on Section 7.5

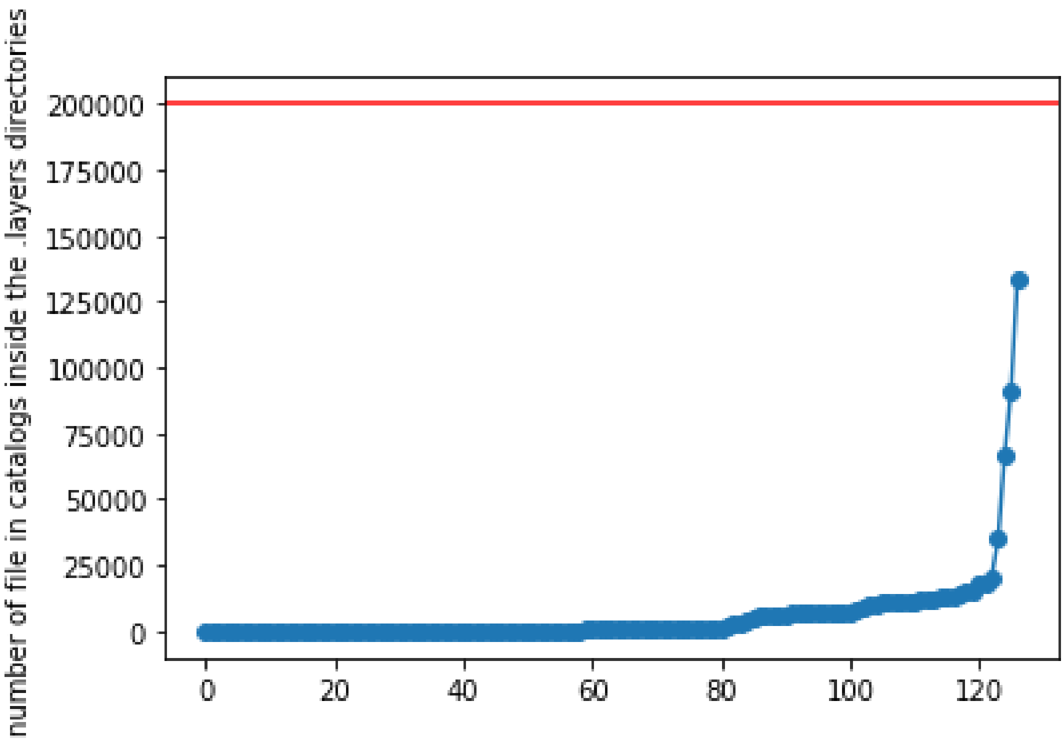


Figure 7.1: Number of files inside the catalogs in the *.layer* directory.

We can see that the use of CVMFS is a huge help when the cache is not available, moreover its overhead is almost negligiable when the cache is present. Is interesting to know that the containers hosted in CVMFS without cache starts in an amount of time comparable to the containers hosted naively but using cache (4 times slower in the case of Singularity and 2 times slower in the case of Docker.)

FILE IN THE SUB-CATALOGS

CVMFS provides a rule of thumb about the number of files that should be hosted in each catalog to avoid stressing the sub-catalog too much. It suggest to limit the amount of file to less than 200000.

Using again the CVMFS command 'cvmfs_server list-catalogs' we obtain the number of files in each catalog.

We show the result in the graphs 7.1, 7.2, 7.3 and 7.4.

We can see how all the catalogs have less than 200000 entries respecting the suggestion of CVMFS.

Moreover, we can notice how the amount of file in a Docker image follows a power distribution.

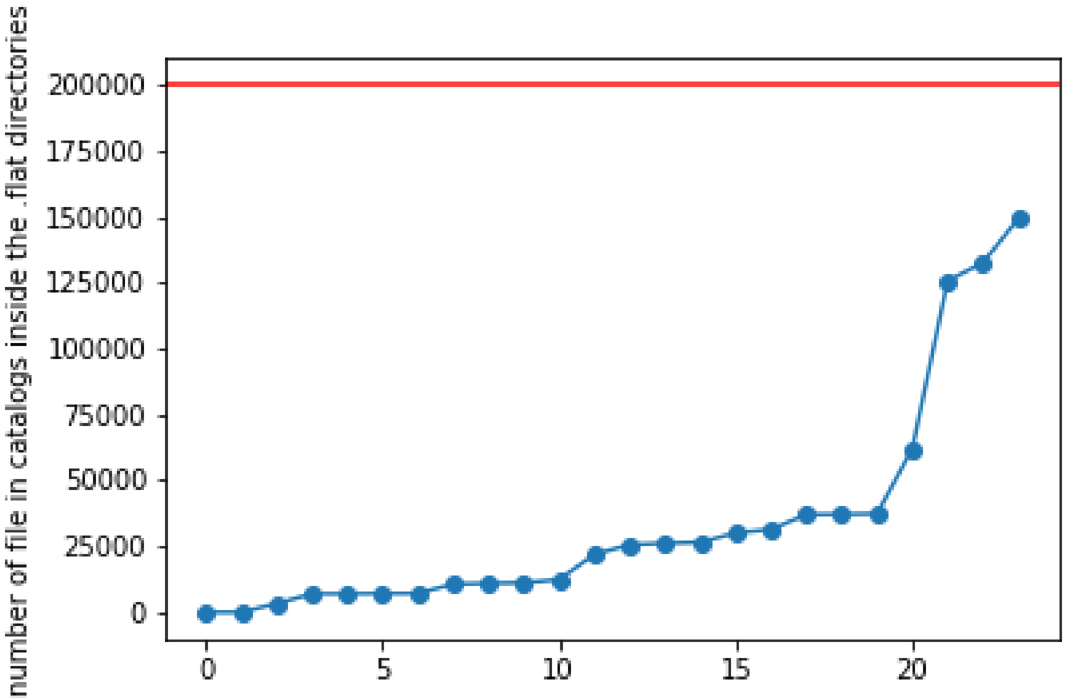


Figure 7.2: Number of files inside the catalogs in the *.flat* directory.

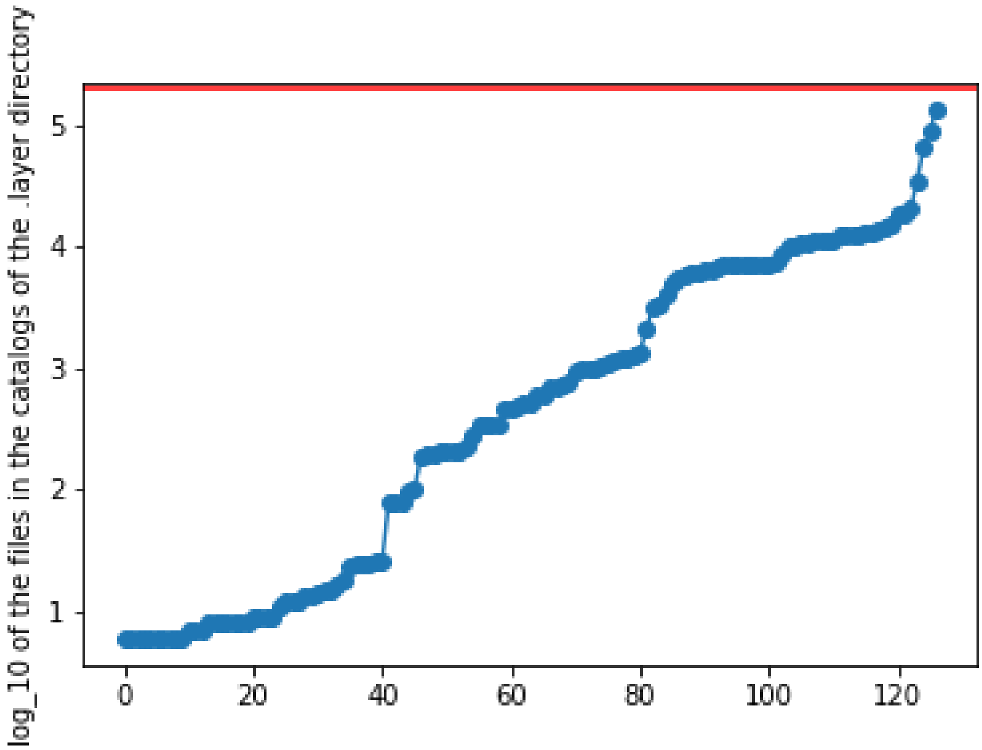


Figure 7.3: Logarithm of the number of files inside the catalogs in the *.layer* directory.

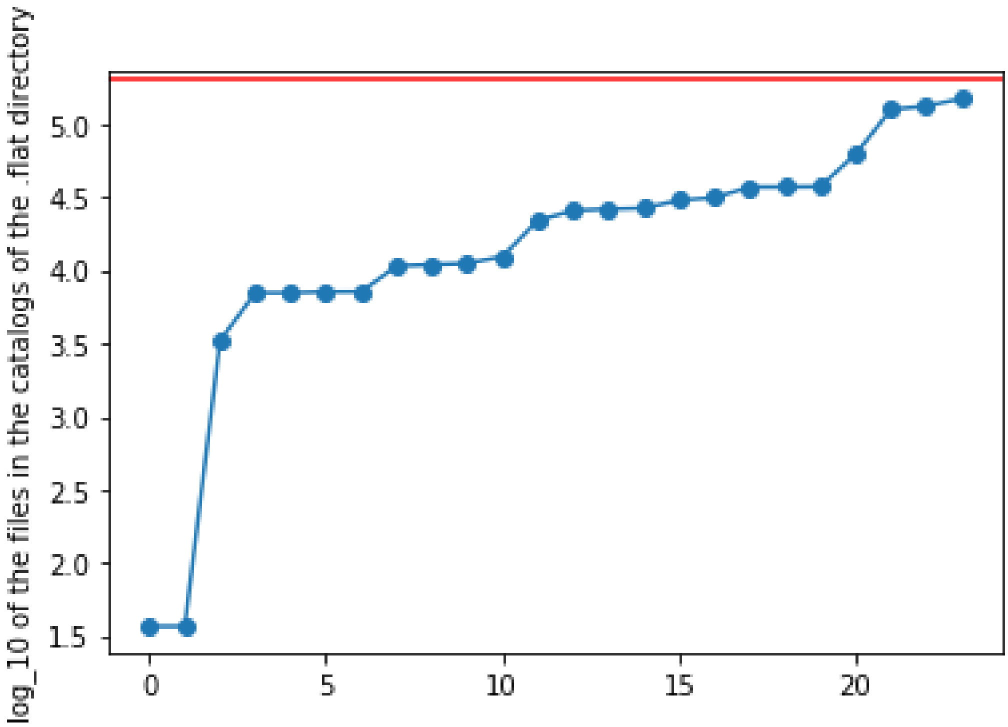


Figure 7.4: Logarithm of the number of files inside the catalogs in the *.flat* directory.

COMPLEXITY

In order to estimate the complexity of managing the file-system we decide to use the cyclomatic complexity of he tool that create and manage the file-system itself that we discuss on Chapter **??**. We used *gocylo* **[gocylo]** a command line tool that calculate the cyclomatic complexity of functions written in the Go(lang) languange.

Listing 7.2: Result of the cyclomatic complexity analysis, only function with complexity greater or equal to 3 are shown

```
>> gocyclo main.go cmd/ lib/ docker-api/
37 lib ConvertWish lib/conversion.go:33:1
15 lib ParseImage lib/parse.go:9:1
14 lib AddManifestToRemoveScheduler lib/cvmfs.go:299:1
12 lib GarbageCollectSingleLayer lib/garbage_collection.go:62:1
12 lib (Image).GetChanges lib/image.go:145:1
12 lib (Image).downloadLayer lib/image.go:436:1
11 lib SaveLayersBacklink lib/cvmfs.go:219:1
11 lib IngestIntoCVMFS lib/cvmfs.go:23:1
9 lib requestAuthToken lib/image.go:519:1
9 lib CreateSymlinkIntoCVMFS lib/cvmfs.go:99:1
9 lib RemoveDirectory lib/cvmfs.go:426:1
8 lib (Image).GetLayers lib/image.go:375:1
7 lib FindImageToGarbageCollect lib/garbage_collection.go:15:1
```

```
7 lib (Image).GetReference lib/image.go:70:1
6 lib AlreadyConverted lib/conversion.go:317:1
6 lib getBacklinkFromLayer lib/cvmfs.go:176:1
6 lib (*execCmd).Start lib/exec_commands.go:35:1
5 lib getManifestWithUsernameAndPassword lib/image.go:303:1
5 lib (Image).GetManifest lib/image.go:125:1
5 lib ParseYamlRecipeV1 lib/recipe.go:23:1
5 lib firstRequestForAuth lib/image.go:337:1
4 lib (Singularity).IngestIntoCVMFS lib/image.go:262:1
4 lib parseBearerToken lib/image.go:499:1
4 lib (Image).PrintImage lib/image.go:93:1
4 lib (Image).DownloadSingularityDirectory lib/image.go:238:1
3 dockerutil MakeThinImage docker-api/util.go:47:1
3 lib (Image).GetSimpleReference lib/image.go:83:1
3 lib (Image).WholeName lib/image.go:49:1
3 lib CreateWish lib/wish.go:26:1
3 lib ExecCommand lib/exec_commands.go:18:1
```

While the cyclomatic complexity is very high is important to note that idiomatic golang codes requires to manually check every possible error returned by other functions, all these checks increase dramatically the cyclomatic complexity of the code. Indeed there are 19 error check without any logic in the code but simply returning early in the ConvertWish function.

CODE TO GET THE NUMBERS

Listing 7.3: Script used to capture the startup time of singularity with image hostes in CVMFS using CVMFS cache

```
#!/bin/bash

for i in {1..101};
do
    /usr/bin/time -f "%U,%S,%E" \
        singularity exec \
            /cvmfs/thin.osg.cern.ch/library/python:latest \
            python -c "quit()";
done
```

Listing 7.4: Script used to capture the startup time of singularity with image hostes in CVMFS without cache

```
#!/bin/bash

for i in {1..101};
do
    #cleanup the cvmfs cache
    cvmfs_config wipecache >> /dev/null;

    /usr/bin/time -f "%U,%S,%E" \
```

```
        singularity exec \
            /cvmfs/thin.osg.cern.ch/library/python:latest \
            python -c "quit()";
done
```

Listing 7.5: Script used to capture the startup time of docker thin-images using both CVMFS and Docker cache

```
#!/bin/bash

docker pull thin-python:latest

for i in {1..101};
do
    /usr/bin/time -f "%U,%S,%E" \
        docker run thin-python:latest python -c "quit()";
done
```

Listing 7.6: Script used to capture the startup time of docker thin-images without Docker nor CVMFS cache

```
#!/bin/bash

for i in {1..101};
do
    #cleanup the cvmfs cache
    cvmfs_config wipecache >> /dev/null;

    # remove the layers from the docker cache
    docker rmi thin-python:latest -f >> /dev/null;

    # cleanup any remaining data from the system
    docker system prune -a -f;

    /usr/bin/time -f "%U,%S,%E" \
        docker run thin-python:latest python -c "quit()";
done
```

Listing 7.7: Script used to capture the startup time of Docker standard images without Docker cache

```
#!/bin/bash

for i in {1..101};
do
    # remove the local image of python
    docker rmi -f python:latest;

    # cleanup any remaining data from the system
    docker system prune -a -f;

    /usr/bin/time -f "%S,%U,%E" \
        -o ~/start-time-docker-standard-no-cache.csv -a \
```

```
        docker run python:latest python -c "quit()";
done
```

Listing 7.8: Script used to capture the startup time of Singularity running Docker standard images without Singularity cache

```
#!/bin/bash

for i in {1..101};
do
    # cleanup the singularity cache directory
    rm -rf /root/.singularity/docker;

    /usr/bin/time -f "%S,%U,%E" \
        -o ~/start-time-singularity-standard-no-cache.csv -a \
        singularity exec docker://python:latest \
            python -c "quit()";
done
```

Listing 7.9: Script used to capture the startup time of Singularity running images unpacked on the local file-system, hence with cache

```
#!/bin/bash

for i in {1..101};
do
    /usr/bin/time -f "%S,%U,%E" \
        -o ~/start-time-singularity-standard.csv -a \
        singularity exec python_latest/ python -c "quit()";
done
```

Listing 7.10: Script used to capture the startup time of docker thin-images without cache

```
#!/bin/bash
for i in {1..101};
do
    # clean the CVMFS cache
    cvmfs_config wipecache >> /dev/null;

    # clean the docker cache
    docker rmi thin-osg/library/python:latest -f >> /dev/null;

    # run and measure the start up time
    /usr/bin/time -f "%S,%U,%E" \
        -o ~/start-time-docker-no-cache.csv -a \
        docker run thin-osg/library/python:latest \
            python -c "quit()";
done
```

8 FUTURE WORKS

1. New containers technologies
2. Improve ingestion of layers into the filesystem making everything in a single transaction
3. Decrease cyclomatic complexity

BIBLIOGRAPHY

[1] I Bird et al. *Update of the Computing Models of the WLCG and the LHC Experiments*. Tech. rep. CERN-LHCC-2014-014. LCG-TDR-002. 2014. URL: http://cds.cern.ch/record/1695401.

[2] J. Blomer, P. Buncic, R. Meusel, G. Ganis, I. Sfiligoi, and D. Thain. "The Evolution of Global Scale Filesystems for Scientific Software Distribution." In: *Computing in Science Engineering* 17.6 (2015), pp. 61–71. ISSN: 1521-9615. DOI: 10.1109/MCSE.2015.111.

[3] CERN. *WLCG*. URL: https://web.archive.org/web/20181122162700/http://wlcg.web.cern.ch/ (visited on 11/22/2018).

[4] Christoph Eck et al. *LHC computing Grid: Technical Design Report. Version 1.06 (20 Jun 2005)*. Technical Design Report LCG. Geneva: CERN, 2005. URL: http://cds.cern.ch/record/840543.

[5] The Linux Foundation. *Image Layer Filesystem Changeset*. URL: https://web.archive.org/web/20181122170713/https://github.com/opencontainers/image-spec/blob/master/layer.md (visited on 11/22/2018).

[6] The Linux Foundation. *OCI Content Descriptors*. URL: https://web.archive.org/web/20181122171756/https://github.com/opencontainers/image-spec/blob/master/descriptor.md (visited on 11/22/2018).

[7] The Linux Foundation. *Open Containers Initiative*. URL: https://web.archive.org/web/20181122170509/https://www.opencontainers.org/ (visited on 11/22/2018).

[8] N Hardi, J Blomer, G Ganis, and R Popescu. "Making containers lazy with Docker and CernVM-FS." In: *Journal of Physics: Conference Series* 1085.3 (2018), p. 032019. URL: http://stacks.iop.org/1742-6596/1085/i=3/a=032019.

[9] Tyler Harter, Brandon Salmon, Rose Liu, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. "Slacker: Fast Distribution with Lazy Docker Containers." In: *14th USENIX Conference on File and Storage Technologies (FAST 16)*. Santa Clara, CA: USENIX Association, 2016, pp. 181–195. ISBN: 978-1-931971-28-7. URL: https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter.

[10] Docker Inc. *About storage drivers*. URL: https://web.archive.org/web/20181122172916/https://docs.docker.com/storage/storagedriver/ (visited on 11/22/2018).

[11] Docker Inc. *Docker Engine managed plugin system*. URL: https://web.archive.org/web/20181122174052/https://docs.docker.com/engine/extend/ (visited on 11/22/2018).

[12] Docker Inc. *Docker Registry*. URL: https://web.archive.org/web/20181122172148/https://docs.docker.com/registry/ (visited on 11/22/2018).

[13] Docker Inc. *Docker overview*. URL: https://web.archive.org/web/20181122172634/https://docs.docker.com/engine/docker-overview/ (visited on 11/22/2018).

[14] Docker Inc. *Docker tag*. URL: https://web.archive.org/web/20181122172506/https://docs.docker.com/engine/reference/commandline/tag/ (visited on 11/22/2018).

[15] Docker Inc. *What is a Container*. URL: https://web.archive.org/web/20181122172321/https://www.docker.com/resources/what-container (visited on 11/22/2018).

[16] Donald E. Knuth. "Computer Programming as an Art." In: *Communications of the ACM* 17.12 (1974), pp. 667–673.

[17] Simone Mosciatti. *repository-manager, utlity to manage the unpacked.cern.ch CVMFS repository*. URL: https://web.archive.org/web/20181125115146/https://github.com/cvmfs/docker-graphdriver/tree/devel/repository-manager (visited on 11/25/2018).

[18] Nikolaus Rath. *libfuse, The reference implementation of the Linux FUSE (Filesystem in Userspace) interface*. URL: https://web.archive.org/web/20181124093000/https://github.com/libfuse/libfuse (visited on 11/24/2018).

[19] Sylabs.io. *Singularity Appendix | run*. URL: https://web.archive.org/web/20181122175328/https://www.sylabs.io/guides/2.6/user-guide/appendix.html#run (visited on 11/22/2018).

[20] Sylabs.io. *Singularity and Docker*. URL: https://web.archive.org/web/20181122174545/https://www.sylabs.io/guides/2.6/user-guide/singularity_and_docker.html (visited on 11/22/2018).

[21] Sylabs.io. *Singularity*. URL: https://web.archive.org/web/20181122154728/https://www.sylabs.io/ (visited on 11/22/2018).

DECLARATION

Put your declaration here.

Saarbrücken, June 2018

André Miede & Ivo Pletikosić

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.