Politecnico di Milano
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica, Informazione e Bioingegneria
Scuola di Ingegneria Industriale e dell'Informazione

Efficient containers distribution at global scale

Relatore: Prof. Marco Domenico SANTAMBROGIO
Correlatore: Dott. Ing. Rolando BRONDOLIN

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

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.

— knuth:1974 [knuth:1974]

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)

ABSTRACT

The problem of software distribution has always been an issue at CERN, given the massive dimension of the software used for analyzing the data they collect from the LHC. The problem of software distribution has been historically solved by the use of a distributed, read-only file system like CernVM-FS that allows distributing binaries to all the geographically distributed data-centers used for High Energy Physics computation. However, the problem of managing run-time dependencies is still open, indeed, some application can run perfectly well in an environment while not working in a different environment.

The problem of run-time dependencies is already been solved in the industry with the use of containers, immutable computing environments which encapsulate all the run-time dependencies of an application allowing it to run with ease on different machines. Moreover, containers are becoming more widespread also inside CERN, suggesting that they will be a key component for running future High Energy Physics workload. However the integration between containers and CernVM-FS is not yet mature enough and no automatic tool to manage the whole lifetime of a container images inside CernVM-FS exists yet.

Indeed, containers are difficult to distribute in an efficient way since their content is stored inside a few large files. Moreover it has been shown that most of the content of containers is not used while running the application itself. Efficient content distribution and managing runtime dependencies should not be contrasting goals. Hence, in this work, we present a way to efficiently distribute the content of containers in order to avoid waste of bandwidth and time.

Il CERN e' una organizzazione Europea che opera il piu' grande laboratorio della fisica particellare al mondo.

La sua missione e' di:

- Rendere disponibile un unico assortimento di accelleratori per permettere la ricerca sui confini della conoscienza umana
- Effettuare ricerca di primo ordine nel campo della fisica fondamentale
- Unire persone da tutto il mondo per migliorare i confini della scienza e della tecnologia per il beneficio di tutti.

Il CERN ospita la strumentazione delle 4 piu' grandi collaborazioni nel campo della fisica: ALICE, ATLAS, CMS e LHCb.

Sono ospitati al CERN anche un insieme di collaborazioni nel campo della fisica piu' piccole che beneficiano delle strumentazioni, conoscienze, effetto network e servizi disponibili.

Tra i servizi che il CERN offre il ai suoi utenti quello informatico e' uno dei piu' interesanti. Infatti il CERN ospita e gestisce il WLCG [6], uno dei piu' grandi centri di calcolo usato per la ricerca pubblica.

Un problema che influenza le operazioni dentro il centro di calcolo e' come installare e impostare il software sui singoli computer. Il software usato nell ambito scientifico e' di grandi dimensioni ed e' molto dispendioso in tempo di banda e tempo spostare ogni binario da una posizione centrale ad ogni singolo computer. Diverse soluzioni sono state proposte e alla fine is problema e' stato risolto con l'uso di CernVM-FileSystem [4], un file system di sola lettura che mette a disposizione un sistema di distribuzione software che e' scalabile, solido e che richiede poca manutenzione.

Provedere solo alla distribuzione del software, pero' non e' sufficiente. Infatti anche le dipendenze a run time devono essere gestite per garantire operazioni senza problemi. Una possibile soluzion per gestire le dipendenze a run time e' l'uso dei containers. I containers impacchettano tutte le dipendenze in un file system immutabile cosi' da poter eseguire l'applicazione sempre nello stesso ambiente anche se su macchine diverse.

Distribuire il contenuto dei containers usando CVMFS non e' pratico, visto che i containers sono distribuiti come un insieme di grandi files e questo e' contrario ai principi di funzionamento di CVMFS che funziona meglio con tanti piccoli file.

Questa tesi esplorera' il problema di create un file system di sola lettura addatto per distribuire il contenuto di containers sui nodi di

computazione. Descriveremo la struttura di un file system di sola lettura generico e implementero la metodologia proposta con CVMFS.

Questa tesi e' divisa in diverse parti: Inizieremo fornendo tutte le informazioni necessarie sulla architettura del centro di calcolo del CERN e sul WLCG, poi esploreremo CVMFS e spiegheremo perche' e' un modello molto adatto alla architetura geograficamente distribuita del WLCG. Continueremo discutendo l'integrazione tra i containers e CVMFS introducendo il problema che questa tesi sta affrontando. Poi esploreremo lo stato dell'arte nei file system distribuiti, come centri di calcolo di diverse dimensioni sono gestiti ed infine esploreremo altri progetti in letteratura che lavorano in modo pigro come CVMFS. A questo punto introduremmo la nostra metodologia per integrare CVMFS e containers. Dopo che abbiamo introdotto la metodologia discuteremo come essa e' stata implementata nel nostro lavoro. Infine valuteremo i risultati del nostro lavoro e suggeriremo possibili miglioramenti.

La tesi e' strutturata come segue:

- Il Capitolo 2 fornisce tutte le informazioni necessarie di base. Inizia esplorando l'architettura del centro di calcolo del CERN, poi spiega il ruolo di CVMFS, la sua architettura di alto livello e come e perche' e' uno strumentto adatto all ambiente del CERN. Il capitolo illustrera' anche le run time di containers che verranno usate in questo lavoro. Alla fine forniremo la definizione del problema che questa tesi si pone.

- Il Capitolo 3 descrivera' lo stato dell'arte. Inizieremo illustrando alcuni file system distribuiti e la loro architettura, Poi descriveremo come centri di calcolo di tutte le dimensioni sono operati e gestiti. L'ultima parte di questo capitolo esplorera' altri progetti nella letteratura che lavorano in modo pigro come CVMFS.

- Il Capitolo 4 descrivera' la metodologia che puo' essere usata da un file system di sola lettura generico come CVMFS per ospitare immagini di containers e raggiungere una distribuzione del contenuto efficiente.

- Il Capitolo 5 mostrera' i dettagli della implementazione e come il file system e' gestito e creato sopra CVMFS usando del software apposito.

- Il Capitolo 6 mostrera' i risultati di questo lavoro comparando il tempo di avvio di containers in scenari diversi, cosi' come l'uso di banda.

- Il Capitolo 7, infine, esplorera' come questo lavoro puo' essere migliorato. Mostreremo anche alcune direzioni per lo sviluppo della distribuzione dei containers che pensiamo siano di interesse.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

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 collaborations: ALICE, ATLAS, CMS and LHCb.

CERN hosts also a plethora of smaller physic collaborations that benefits from the instruments, know how, network effects and services available.

Between the services offered to its users the computing service is one of the most interesting. Indeed CERN host and manage the WLCG [6], one of the biggest computing data center used for public research.

An issue that affect the operations inside the data center is the provisioning of software on the computing servers. Scientific software is large and is very time and bandwidth consuming moving every binaries from a central repository to the computing nodes. Several solutions have been proposed and eventually the problem has been solved with the use of CernVM-FileSystem [4], a read only file-system that provides a scalable, reliable and low maintenance software distribution system.

However simply provide the servers with the necessary software is not enough. Indeed also run time dependencies need to be managed to ensure smooth operations. A possible solution to manage run time dependencies is the use of containers. Containers pack up all the run time dependencies in an immutable file system so that they can execute their application always in the same environment, even if running on different machines.

However, distribute containers content using CVMFS is not convenient, since the containers are distributed as a set of big files and this works against the working principle and performance of CVMFS that works better with a lot of small files instead of few large files.

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 divided in several parts: We will start by providing all the necessary information on the CERN computing architecture and the WLCG, then we will explore CVMFS and we will explain why is a good fit for the geographically distributed model that follows the WLCG. Moreover we will discuss of the integration between containers and CVMFS itself and we will introduce the problem that this thesis is addressing. Then we will explore the state of the art exploring alternative distributed file system, different way to provisioning machines inside data center of different size and finally we will explore other project in the literature that work lazily like CVMFS. At this point we will introduce our methodology to integrate CVMFS and containers. Once the methodology has been introduced we will discuss how it is been implemented. Finally we will evaluate the results of the proposed methodology and implementation and propose future works and enhancement to the implementation.

This thesis is structured as follows:

- Chapter 2 provides the necessary background. It start by exploring the computing architecture at CERN, then it explain the role of CVMFS, its high level architecture and how and why it fits well in the CERN environment. The chapter will then illustrate the container run-times that we will exploit in this work. Finally we are going to state the problem definition of this thesis.

- Chapter 3 will describe the state of the art. We will start by illustrating few distributed file systems and their architecture. Then we will move on to describe how provisioning of servers is done in environments of different size and dimensions. The last part of this chapter will explore other projects in the literature that work lazily as CVMFS.

- Chapter 4 will provide the methodology that can be used by a generic read-only file system like CVMFS to host containers images and achieve efficient content distribution.

- Chapter 5 will dig into the details of how the file system is managed and created on top of CVMFS using custom software.

- Chapter 6 will show the result of this work comparing start up times of containers in different scenarios as well as the bandwidth used.

- Chapter 7 will finally explore how this work can be enhanced and improved. Moreover we will show other directions regarding the distribution of containers content that we believe are of interest.

2

BACKGROUND AND PROBLEM DEFINITION

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 file system. 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 a 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 [6],[8].

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 [8]. 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 assumes and supports a batch computing paradigm. Analysis and simulations are split in smaller jobs that are distributed to different computing node that can work in parallel [8], [3].

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

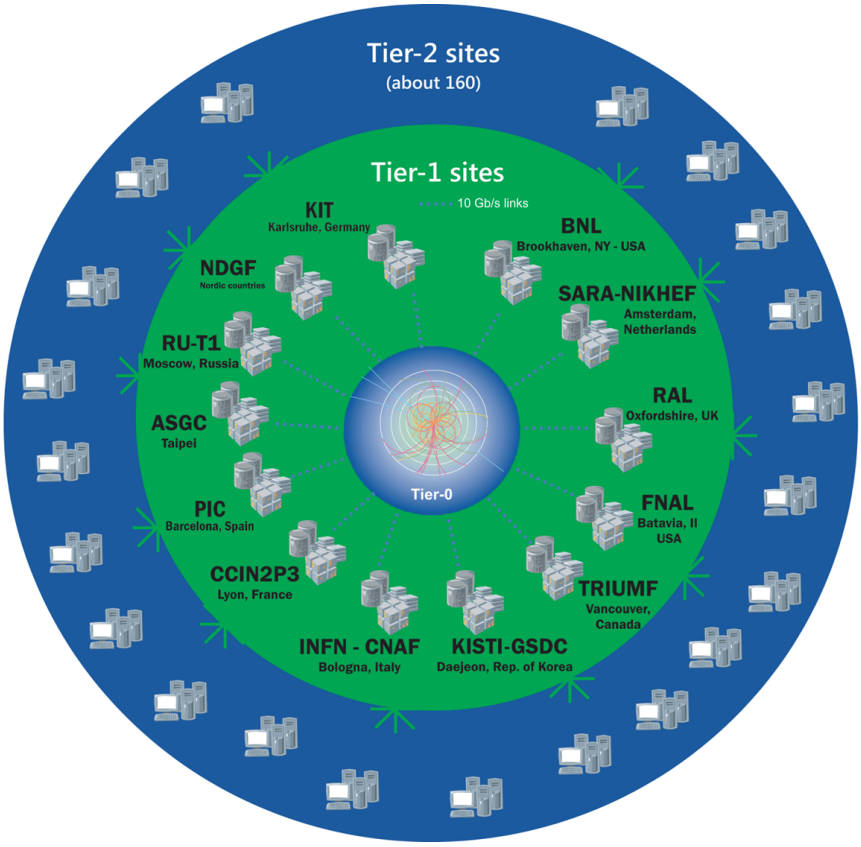

Figure 2.1: Schematic representation of the WLCG

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) [4].

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 [4] 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) [29] 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 model 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. Moreover, while normal files hosted in CVMFS can be

split into small blocks and each block can be serve separately, this is not supported for catalogs. Indeed each sub-catalog needs to be fully downloaded before it can be used. Hence big catalogs can be a bottleneck both with the respect of query time, download time and bandwidth.

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 it 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 [21].

In order to standardize containers and make the technology interoperable in 2015 the *Open Container Initiative (OCI)* was founded [12]. The OCI defined a standard format to use to pack containers into *images* [9], this format have been adopted by Docker and 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 [10]. 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 [11].

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 [18].

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 [20]. 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 [19].

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 [16]. On average less than 7% of the content inside a Docker image is used during the runtime of the image [14]. The layers are distributed as a single tar file [10], 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" [13]. In this work we are not interested in the internals of the `cvmfs/graphdriver` plugin, it will be sufficient to know that the Docker daemon can be enhanced with the use of plugins [17] 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* an example of *recipe* file is show on Listing 2.1. *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 [13].

Listing 2.1: Example of a *recipe* file of a Docker *thin image*

```
{
  "version": "1.0",
  "origin": "https://registry.hub.docker.com/library/python:
      latest",
  "layers": [
    {
      "digest": "bc9ab7...0257b7",
      "url": "cvmfs://thin.osg.cern.ch/.layers/bc/bc9ab7...0257b
          7/layerfs"
    },
    {
      "digest": "193a63...110d7f",
      "url": "cvmfs://thin.osg.cern.ch/.layers/19/193a63...110d7f
          /layerfs"
    },
    {
      "digest": "e5c3f8...818580",
      "url": "cvmfs://thin.osg.cern.ch/.layers/e5/e5c3f
          8...818580/layerfs"
    },
    {
      "digest": "b61a0d...b615a9",
      "url": "cvmfs://thin.osg.cern.ch/.layers/b6/b61a0d...b615a
          9/layerfs"
    }
  ]
}
```

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.

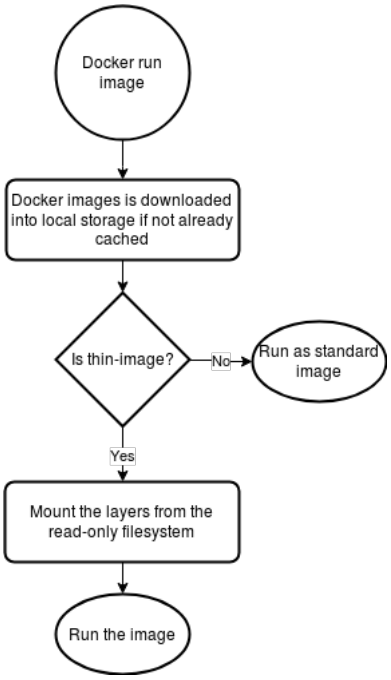

Figure 2.2: Decision proces for running docker thin images

Moreover the `cvmfs/graphdriver` plugin is able to run standard (*fat*) Docker images, hence conventional Docker images are supported seamlessly [13].

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 runs it as a standard image, if it detects 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, stored in the local storage and successfully run using the content provide by CVMFS. On a second moment the images can be updated, 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 tries 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 [32] is another container runtime, it provides its own image format but it is capable to run standard Docker images [31] as well. Moreover it is capable of running containers, also Docker containers, directly from a directory containing the unpacked container file-system itself [30].

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 introduced how CERN has overcome the challenges of software distribution using CVMFS. Unfortunately this is not enough since run-time dependencies between software components can break applications 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, working principles and performance characteristics of CVMFS.

Both CVMFS and containers aim to solve the same challenge of server provisioning, however they made different trade-offs. CVMFS opted for an efficient distribution of the content, making difficult and inconvenient to pack all the runtime dependencies of a particular application. Containers, instead, opted to make each application self-contained so that the application can run reliably on different computer environments, however losing efficiency in the distribution of the content.

Efficient content distribution and efficient dependency management should not be contrasting goals. For this reason, in this thesis work we will tackle the problem of how to efficiently distribute software in HPC clusters while efficiently preserving all its runtime dependencies.

3

STATE OF THE ART

In this chapter we will provide the state of the art for the technologies that are of interest for this work. In Section 3.1 we will discuss the state of the art in distributed file system where we will see the major difference between CVMFS and other distributed file system. In Section 3.2 we will explore how provisioning and installation are done on computing clusters of various size, starting from the smallest up to the globally distributed one. Finally in Section 3.3 we will explore early achievement of working lazily when provision machines.

DISTRIBUTED FILE SYSTEM

Several distributed file systems have been proposed and are available. One of the first implementations of a distributed file system is the Andrew File System (AFS) [24]. It provides a read-write file system, however it guarantees only weak consistency, hence read and write operations are directed only to locally cached copy of the file. The changes are propagated to the server when a file is eventually closed. AFS informs the client if a different user modifies a cached file using callbacks, but those callbacks are discarded after any communication error between the client and the server, including network failures or timeouts. The need to keep a constant connection open between the client and the server makes the client/server ration quite small, 200. Hence for every 200 clients a new server machine is needed.

Another commercial distributed file system is BeeGFS [33] it has a different approach than AFS, since it is composed by three different components: the clients, the metadata servers and the storage servers. The client coordinate and informs the metadata servers which in turns coordinate the storage servers that, finally, are accessed by the clients. Also this architecture of distributed file system relies on metadata components that bounds the number of clients imposing an overhead.

Another approach to store huge amount of data is Amazon S3 (Simple Storage Service) [15]. S3 does not provide a classical POSIX interface but an eventually consistent REST interface. Objects are created immediately but deletions will eventually happen in the future. Moreover it has no knowledge of permissions outside of a simple access control list, symbolic links or even folders. Due to this shortcomings S3 is not directly usable to distribute and instal software one HPC clusters, however it can be used as a simple layer to access the file leaving all the metadata handling to CVMFS. As mentioned in Section 2.2 the files provided by CVMFS are accessed using the HTTP protocol.

And they don't have to be in a web server managed by CVMFS but they can be in any accessible location.

To the best of our knowledge, no distributed file system handles metadata like CVMFS does, indeed all the metadata queries to CVMFS are handled by the client themselves reading the catalogs files, hence the metadata operations are not a limitation anymore since they scale out with the number of clients. Of course this is possible since CVMFS provides to the clients only a read interface and the write are all serialized in the CVMFS server component.

PROVISING OF MACHINES

In this section we are going to explore how it is possible to provision and install software on cluster of machines. The proposed methodologies will grow in complexity as the size of the cluster grows. In parallel we will see how the problem shifts from the handling of files to the handling of metadata.

In Section 3.2.1 we will describe how installation and provisioning of machines is done in humble set up with very few machines. The need of automation will arise on bigger clusters, hence in Section 3.2.2 we will describe how automatic tools help in provisioning local clusters. In Section 3.2.3 we will explore the solution that were applied to the WLCG before CVMFS and their weaknesses. Finally in Section 3.2.4 we will explore proposed solution to the problem of managing run time dependencies in a globally distributed cluster like the WLCG.

Single o Local Machines and Small Clusters

Several systems have been proposed to manage installation and provisioning of machines.

The simpler approach is of course the manual approach were the source code is downloaded on the local machine, compiled and finally installed. While this works fine for the expert user, it introduces a lot of difficulties and friction for users that are not experts. The compilation part is error prone and it may require dependencies, so the user must be ready to do a lot of research and even reading makefiles if necessary. Moreover, if the software has run-time dependencies, it is necessary for the user to install them as well, and so recursively.

Of course there is no need to re-compile the software every time, moreover compilation is an expensive and quite technical step that not all users are confident in doing. Indeed the package managers [34], [27] solves this kind of problem, the software is compiled once by expert on the software itself and distributed along with a script that install it in the most appropriate location in the system and with a series of dependencies. This is a major shift in the interface since now the user simply installs a package without caring of all its dependencies

or without even needing a compiler. It is the package manager that will eventually install run time dependencies, directly or indirectly.

The distribution of packages poses already a challenge, if enough users are interested in the packages hosted in a single package manager, it would be necessary to load balance the request between several mirrors site, indeed this is what happens in big linux distribution that serves a lot of users. The package manager itself is mirrored to different data center each one serving a part of the traffic.

Local cluster

As long as the installation of software is manual and done in relatively small clusters, package managers are suitable for hosting all the software and provision all the servers.

When the cluster starts to grow, manual installation is not a solution anymore, indeed specialized tools are available to manage the state of a fleet of machines.

Ansible [22], [2] is a tool that helps in managing the installation of software in a fleet of machines. It is based on two simple concepts, the *inventory* and the *playbook*. The inventory associates each machine that is managed to a set of tags. The playbook is a set of statuses each associated with a tag. When provisioning a fleet of machines, Ansible tries to adapt the status of each server in a way that it respects the status declared in the playbook. As an example in the playbook we may declare that the servers tagged as `web-server` need to have the `apache` service on, while the servers tagged as `build-node` needs to have the `gcc` package installed. In the first case Ansible will try to start the `apache` service while in the second case it will try to install the `gcc` package.

A possible issue with Ansible is about the use of those servers: users may login into the server, change their configuration to fix an urgent problem and then do not update the Ansible playbooks. In the long term this causes a drift on configuration and settings that makes quite difficult to manage the fleet of services.

A solution to the drift in configuration is the use of a different tool, *Puppet* [28], [2] which works on a similar principle of Ansible but it constantly monitor the status of each machine reverting any changes.

If different users need to access a shared pool of computing resources, another solution consist in the use of *virtual machines*. Each physical machine gets partitioned into a set of different, isolated virtual machines where each users can install and configure its software. Moreover the image of the virtual machine itself may come with software already installed and configured.

Providing software already installed in a machine is a major conceptual shift where we move from the user having the responsibility to install all the needed software to a model where the software is already

in the system itself. However, virtual machines while providing the stronger level of isolation possible in a shared environment are slow to set up and also slower than native performance.

An evolution of virtual machines are the containers: containers provide a smaller level of isolation than virtual machines but are much quicker to set up and have close to native performance. Moreover containers are simple to build and their popularity is raising between developers as well. Indeed it is possible to set up a container that encapsulates all the status necessary to run a specific computation: this allows to move the container from a machine to another without worrying anymore of run-time dependencies.

All these solutions work well in local clusters, however all of them requires to move all the necessary software on a local machine, also components that are not needed. Starting a fleet-wide installation using tools like *Ansible* or *Puppet* means that the package managers needs to serve at roughly the same time all the packages necessary by all the machines, which can quickly sum up to a lot of bandwidth. The use of virtual machines or containers is not better in this cases since it still requires to move a lot of content. In big cluster it is common to provide local package managers and local container registries.

Providing the content to several different machines is a problem that can be solved by a careful use of caches and optimizations. Diving deeper into the containers world, the OCI standard defines that each layer is identified by a single unique digest. This helps since it provides an unique identifier to cache and, more importantly, by bundling together a lot of different files in a single tar files it sidesteps the need of managing metadata for all the files in a container. Still this comes at the cost of needing to download a big file even if its content is not all necessary. Similar consideration can be done for the provisioning of Virtual Machines.

WLCG environment

The WLCG poses several challenges with the respect of provisioning. First of all it is a geographically distributed data center, hence in order to avoid slow and inter-data center communication is necessary to cache all the necessary binaries in each data center. Then it operates in batch mode, hence if several jobs started all together in different machines and if each of them would need to download binaries or containers, they will quickly overload both the network infrastructure and the content provider. Moreover, scientific software is usually quite large, as an example the *atlas/analysisbase* [1] Docker image, compressed, is roughly 1GB on size. Indeed, several solution have been proposed and tested to address these issues.

1 We are refering to the Docker image relative to the version 21.2.56 of the atlas/analysisbase tools.

The *Andrew File System* [24] (AFS) is a distributed read-write file system optimized for home directories on globally distributed workstations. AFS provides a global name space (/afs) partitioned into *cells* and *volumes*. Scientific collaborations can host the authoritative copies of their experiment software on public volumes (cell /afs/cern.ch). Using AFS, scientific software need to be installed only once in the server in order to be used by several clients. Since AFS needs to keep the connection state for all its clients it has a low client/server ration with is limited to about 200:1.

Another proposed mechanism to manage installation on the WLCG are the *grid installation jobs*, described in [5]. Jobs that pre-install software releases on worker node. In order to avoid to install the software on every single node WLCG Tiers provides a *share software areas*, a NFS volume mounted to every worker node that can be used to provide the software itself. While the *share software areas* are a necessity, they also introduce a single point of failure and high chance to overload the NFS with meta-data operations.

An evolution of the *grid installation jobs* is the *ALICE Software Installation System* described in [5]. The software releases are still distributed as packages but those packages are physically distributed to the worker nodes using the BitTorrent protocol. This improves the installation times using all the bandwidth not only from the package manager through the worker nodes but also between the worker node themselves. Unfortunately this architecture required to still transfer all the packages to all the nodes which is a waste of bandwidth, introduce sources of errors and it is time consuming.

Another evolution of the *grid installation jobs* is *GROW-FS* [7] which pioneered the idea of meta-data handling on the worker nodes and not in a central node. *GROW-FS* provides as well a shared areas for installation using a simple web server. *GROW-FS* uses a single catalog for the entire directory tree and changes to the file system structure requires to completely reconstruct the catalog as well as re-mounting the file system on the worker nodes.

Managing run time dependencies on the WLCG

Other than the distribution of content, CERN faced the problem of managing run-time dependencies. A possible solution to this problem would be to statically link all the dependencies, but this would generate extremely big binaries. Another solution would be to carefully managing all the software installed on the machines, including all the recursive dependencies. However this clash with the WLCG model where jobs can migrate from a data center to another along with their dependencies. Moreover sometimes even a very careful installation is not sufficient since it is possible that two application that could run on the same machine have clashing dependencies.

The problem of how to manage runtime dependency is been solved outside CERN in several different ways.

On small scale it is possible to use simple package managers that automatically install the dependencies, however they suffer of few different problem. First and foremost in case of dependencies clashing the package manager simply refuse to install the required package, which is not an acceptable solution. Then at the scale of the WLCG the package manager quickly become a bottleneck with respect to the bandwidth necessary to distribute the content but also with respect to the size of the internal databases.

Another possible solution is the use of containers. Packaging all the runtime dependencies along the application code in an immutable file system allow containers to exactly reproduce the same environment and condition to ensure smooth operations. The use of containers however is not convenient in very large clusters like the WLCG, indeed the content necessary to run a container is distributed as few large tar files and this makes the distribution itself inefficient since a lot of content not useful for the application itself is downloaded anyway.

Several system have been propose to decrease the downloading time of containers images, however all of them work *ahead of runtime* trying to optimize how fast the overall system is capable of delivering the whole image into the host. FID (Faster Image Distribution) [23] is a P2P Docker images distribution system that is able to accelerate the speed of distributing Docker images by taking full advantage of the bandwidth of not only the Docker Registry but also of the other nodes in the cluster, while decreasing the downloading time. However, FID still require to download all the image before to run the computation. Another work by Anwar et all. [1] characterize the workload of large-scale registries in order to derive design implications for more optimized registries, still they are working to optimize the time necessary to serve and download the whole image from the registry.

While the aim of this work is still to decrease the start up time of uncached containers we took a different road. The previous works focus on optimizing the delivery time of the whole content image, we decided to focus on minimizing the amount of content that the client needs to download which of course leads to shorter start up time for the containers and also to save bandwidth.

WORKING LAZILY

The experience with the installation mechanisms tested at CERN suggests that it is necessary to work lazily in order to provide software and content to a wide fleet of machines and this is exactly what CVMFS does for us. However we need another step in order to solve also the problem of run time dependencies.

Few other system have been proposed to address the problem using a similar architecture. The *cvmfs/graphdriver* [13] allows us to run Docker images starting from a *recipe* file that list the layers to mount and their location in the file system. Unfortunately a way to provide the *thin images* was not provided, moreover it was not provided a way to structure the file system itself. Slacker [14] tackle the same problem but it uses a very different implementation. Their work is based on NFS and flatten layers. All the layers of an image get flattened into a single layer, and such layers stored as a single file into a NFS shared between the Docker Registries and the Docker Client. While *cvmfs/graphdriver* shares a set of layers as Docker images, Slacker share a simple reference to the file stored in the NFS. Slacker is able to reach very interesting performance thanks to the implementation of the NFS they use that relies on network disks. We lack the details about the NFS used by Slacker but we believe that their performance may be strongly influenced by the size of the cluster.

Similarly to Slacker in another work Nicolae et al. [26] proposed another lazy structure this time to run Virtual Machines. They trap the IO to the Virtual Machine image using FUSE and redirect those call into a shared read-write file system backed by local disk on each machine. Similarly to Slacker we believe that this approach works well on a small set of machines but it won't scale well on a big cluster. Indeed their experimental set up was limited to 120 machines. As mentioned above for the *grid installation jobs* the use of NFS brings the risk of overloading the NFS itself with metadata operations.

In this work we aim to finally bridge the two worlds of containers and distributed read-only file system like CVMFS. Instead of working with blocks like Slacker and Nicolae et al. our primitives will be files. Similarly we are not going to provide a read-write interface to the worker node since this will definitely impact the scalability of the solution given the necessities to implements locks. The client will only be able to read the files.

4

METHODOLOGY

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

Section 4.1 will provide a high level overview of the proposed methodology explaining why we decided to focus our design on CVMFS, Singularity and Docker with the *cvmfs/graphdriver* plugin.

Section 4.2 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 of CVMFS.

Section 4.3 will analyze how we stored 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 Section 4.4 will show how we kept track of the images already present in the file-system.

HIGH LEVEL OVERVIEW OF THE PROPOSED FILE SYSTEM

The aim of this work is to provide an efficient content distribution system to run containerized application on big distributed clusters.

The aim of this work is to efficiently distribute software in HPC clusters while efficiently preserving its runtime dependencies. To tackle this problem we will resort on CVMFS to efficiently distribute the content of the containers together with containers technologies to preserving the runtime dependencies in such a way that only the strictly necessary files will be downloaded into the worker nodes of the clusters.

The content distribution part will be managed by CVMFS which provides the primitives necessary to efficiently distribute the content. Moreover, to the best of our knowledge, CVMFS is the only stable and tested system that allows to distribute lazily and efficiently Terabytes of data in a distributed data center like the WLCG. Starting from the

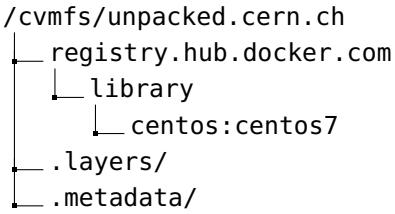

Figure 4.1: High level visualization of the proposed file-system

data distributed by CVMFS, there are two container run-times that we can use to actually run the containers. The first one is Singularity, introduced in Section 2.3.2 that allows to run containers whose content is already unpacked in a simple folder. The second one is Docker with the *cvmfs/graphdriver* plugin, which allows Docker to run *thin images* made of a simple *recipe* `json` file which describes where are in the file system the necessary layers that are needed to be mounted before to start the container itself.

The proposed structure is absolutely generic and even if not implemented with CVMFS it would allow the above container run-times to work with minimal modifications [1]. However some choices of the directory structures have been taken starting from the design of CVMFS and may not be necessary on a generic read-only file system.

The proposed file-system provides 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 structures another hidden directory will contain 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 mentioned in Section 2.3, hence the first directory is the name of the registry that host the image, it follows the namespace which refers to the user of the 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 aggregated 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, allowing to quickly locate the metadata information of an image given just the name of the image itself.

1 The implementation of *cvmfs/graphdriver* relies on a specific *cvmfs* url in the *recipe* format which is a simple implementation details that can be easily generalized.

```
/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 4.2: Visualization of the Filesystem structure, the arrows indicate symbolic links

SINGULARITY IMAGES

To run Docker images using Singularity it 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 Section 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 [2] and the CERN internal registry [3].

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

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

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

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

Figure 4.3: Visualization of the "super directories" in the ".flat" subdirectory

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. Moreover 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 4.2.

The folder that contains the real content of a Singularity image 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 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 4.3 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 decided 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 Section 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 layer 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 layers may be shared between images.

Removing layers has the important implication that once the layer is removed every thin image that relies on it would not work anymore. However those thin-images could be stored on the client side where we don’t have any access.

To not disrupt the user workflow while keeping the repository to a manageable size we considered several options:

1. Never remove layers

Listing 4.1: 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
```

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

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

Listing 4.2: 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
```

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 Figure 4.4

KEEPING TRACK OF THE WORK ALREADY DONE

To avoid to perform duplicated work it 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 are hosted in the file-system.

In order to know which image is already been added to the file system we need to uniquely identify each image. As already mentioned, using the combination of image name and tag is not enough, since the tag is 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 Section 2.3 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 it is possible to understand if the image should be updated or not.

The structure of the `.metadata/` folder is shown in Figure 4.5.

```
/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 4.4: Complete visualization of the `.flat` directory

```
/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 4.5: Structure of the `.metadata/` directory

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 made 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 concept 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 to remove an image from the file-system. We decided to keep a reference count to know when it 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 super-directories.

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

5 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 [25].

CVMFS WRITE INTERFACE

As mentioned in Section 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 it 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 it is possible to modify it using the standard POSIX API, Linux commands or even graphical file explorers. Moreover it 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 it is important to keep in mind that only a single transaction can be opened at any given time since 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, in

Listing 5.1: 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
```

order to minimize the possibilities of inconsistencies or of errors. Once the download completed 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 described in 4.2.

This few steps are sufficient to make the Singularity images available through CVMFS. The pseudocode 5.1 show the details of the algorithm.

DOCKER INGESTION

Converting Docker *fat images* into Docker *thin images* is a more complex task 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 the format used to distribute Docker images as mention in Section 2.3. Then 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 decided 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 directories in the tar file into the directory provided as input. This command implicitly opens and commits a CVMFS transaction, hence it 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 image is already been successfully ingested into the file system. This check happens using the metadata stored in the *.metadata* directory as mention in Section 4.4. The check consists 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 terminates.

The next step is to ingest each layer of the Docker image into the CVMFS file system. As previously we check if the layers already exist in the file system itself. Since the layers are stored under a path that embeds theirs own digest as describe in Section 4.3, checking if a layer is already in the file system consists in simply checking if the folder where we would ingest the layer already exists or not. 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 image, the layer is first downloaded into a temporary directory and then it 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 non 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 as mentioned in Section 2.3.1. To create this image it is sufficient to encode the location of the 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 Section 4.4.

GARBAGE COLLECTION OF IMAGES

We have described how we can 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 Section 4.3 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 it 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 to 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 single syntactical transformation for the Output images. An example of this specific *wish list* is show on Listing 5.2.

Listing 5.2: 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 5.1.

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 enables 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

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

Table 5.1: Available placeholders and their application to the image `https://registry.hub.docker.com/library/centos:centos6`

of the system act as a gatekeeper, inspect the image that has been required to be added and decides if it is the case to add the image or not.

THE *repository-manager* COMMAND LINE TOOL

All the work presented in this chapter has been implemented in the *repository-manager* [25] 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 Section 5.2 and follow all the procedures 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 needs to delete an image it adds 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 described in Section 5.4.

Finally another command of the utility is worth to mention, is the *loop* command, which simply executes the *convert* command in an infinite loop, each time reading again the *wish list* file. This allows the administrator to run the conversion in the background and set up a periodic job that updates the *wish list* file downloading it from a version control system.

6 RESULTS

In this section we will explore the experimental results of this work. First we will focus on the startup time of containers, and we will make two different comparison for this metric. First we will compare the use of CVMFS structured as explained in this work as content distribution mechanism against the standard way of distributing content. Then we will compare a more naive use of CVMFS without the use of *super-directories* and sub-catalogs against the proposed file system structure that instead exploit the sub-catalog mechanism.

Then we will compare the use of bandwidth, again we will make two different comparison. The first will compare the use of CVMFS as content distribution layer against the normal distribution mechanism used for containers base on Docker registries. Then we will compare the use of the proposed file system structure against a file system, still hosted in CVMFS, but without the use of sub-catalogs.

Finally we will measure the complexity of managing the proposed file system structure.

CONTAINERS STARTUP TIME

In the first part of this section we explore the startup time of container hosted in CVMFS using the proposed structure against containers hosted in a normal file system. The second part of this section will analyze the startup time of container hosted in CVMFS using the proposed file system structure against hosting containers in CVMFS but without the use of *super-directories* and subcatalogs.

On both parts we will use a representative use case, we will start the standard python container, start the python interpreter and immediately quit the interpreter itself executing the *quit()* command. We decide to use this use case since it requires to have access to the python interpreter in the container, which is an executable of not negligible size: 3.4MB. Moreover, we chose the standard python image because is a widely known image used during both development and production work. The size of the whole image is of roughly 923MB.

All the measurements are collected using the standard unix util *time*.

Use of CVMFS vs. Standard containers distribution mechanism

For this analysis we propose 8 scenarios that we synthesize in Table 6.1 and on the graph of Figure 6.2. The *Thin-Image on CVMFS* row

		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
Default	Singularity	7.38	2.07	1884.76	366.84
	Docker	95.26	5.29	1279.21	168.99

Table 6.1: 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. $n = 100$

represent the startup time of containers which content is hosted in the CVMFS architecture discussed in this work. The *Default* columns represent the start up time of containers using the default distribution system based on Docker Registries.

We can see that the use of CVMFS is a huge help when the cache is not available, moreover its overhead is almost negligible when the cache is present. Is interesting to know that the containers hosted in CVMFS without cache starts in an amount of time whose order of magnitude is 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.)

Use of the proposed file system structure vs. using CVMFS without sub catalogs

In this section we are going to analyze how the use of sub-catalogs affect the startup performance. Differently from the above test, in this case the differences lies mostly on the uncached case. Indeed when the sub-catalog is already downloaded and cached in the local client the time difference is negligible. For this test, as base case we use the containers served using the file system structure presented in this work, on the other side we serve the containers without using sub-catalogs. Moreover, in the case without sub-catalogs we present the case where the whole repository contains 10, 20, 30 and 40 images.

We used the Singularity runtime for this test that being faster to start up is less influenced by the run time overhead. Moreover, the base case is equivalent to the case showed in the previous test, using the Singularity run time, served by CVMFS without cache. Hence the base case we are comparing is the one with average startup time of 29.12 hundredths of second and standard deviation of 4.02 hundredths of seconds.

On Table 6.2 we can see how, on increasing the images in the catalogs, hence in increasing the size of the subcatalog the time to start a containers without using the cache increases as well.

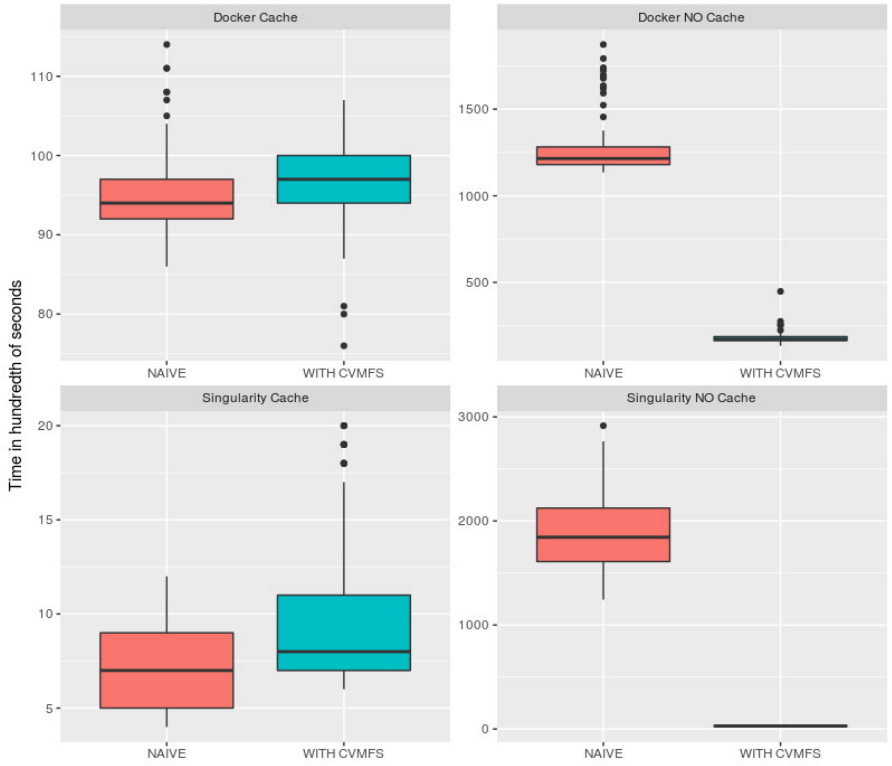

Figure 6.1: Startup times of the containers served using the proposed architecture and served using the standard Docker Registries.

Images in Catalogs	10		20		30		40	
Size of subcatalog	67MB		127MB		188MB		249MB	
Startup time	AVG	STD	AVG	STD	AVG	STD	AVG	STD
	57.61	6.01	88.76	7.52	129.50	15.95	150.42	29.84

Table 6.2: Startup time of Singularity containers with the respect of the subcatalog size. The units are in hundredths of seconds. $n = 100$

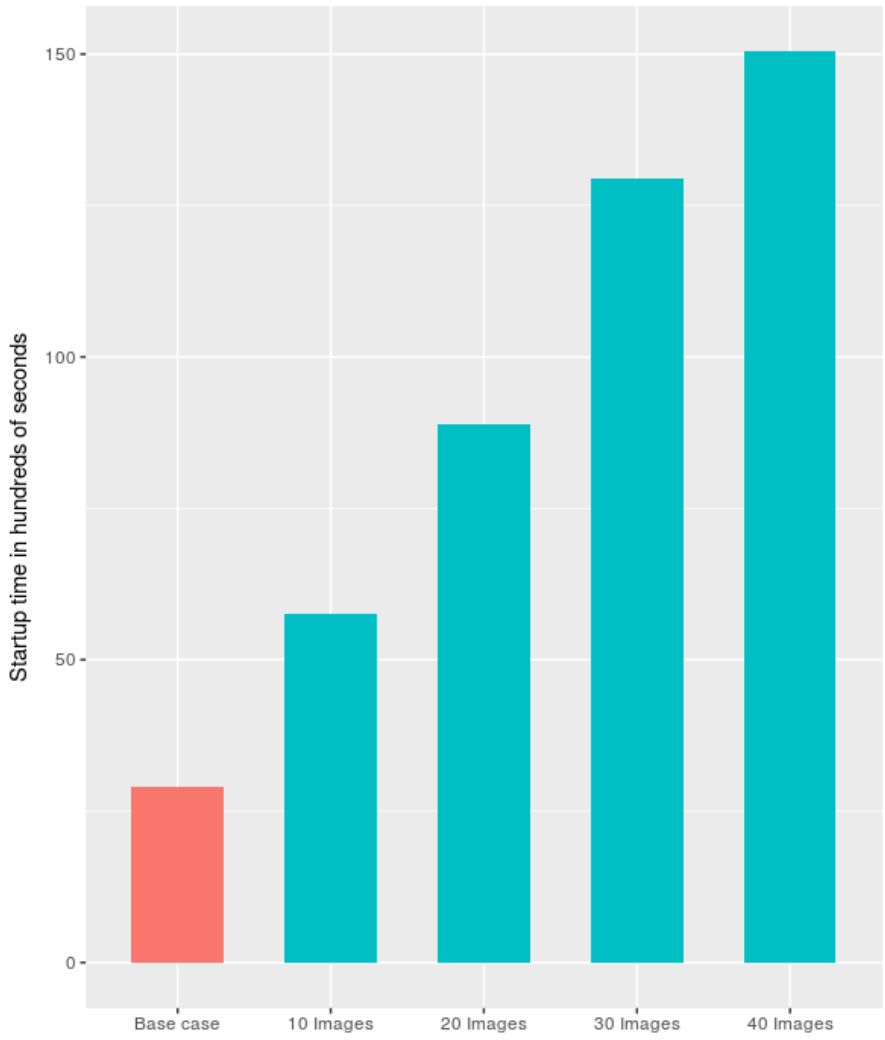

Figure 6.2: Comparison of the containers startup times with the proposed file system structure and without the use of subcatalogs.

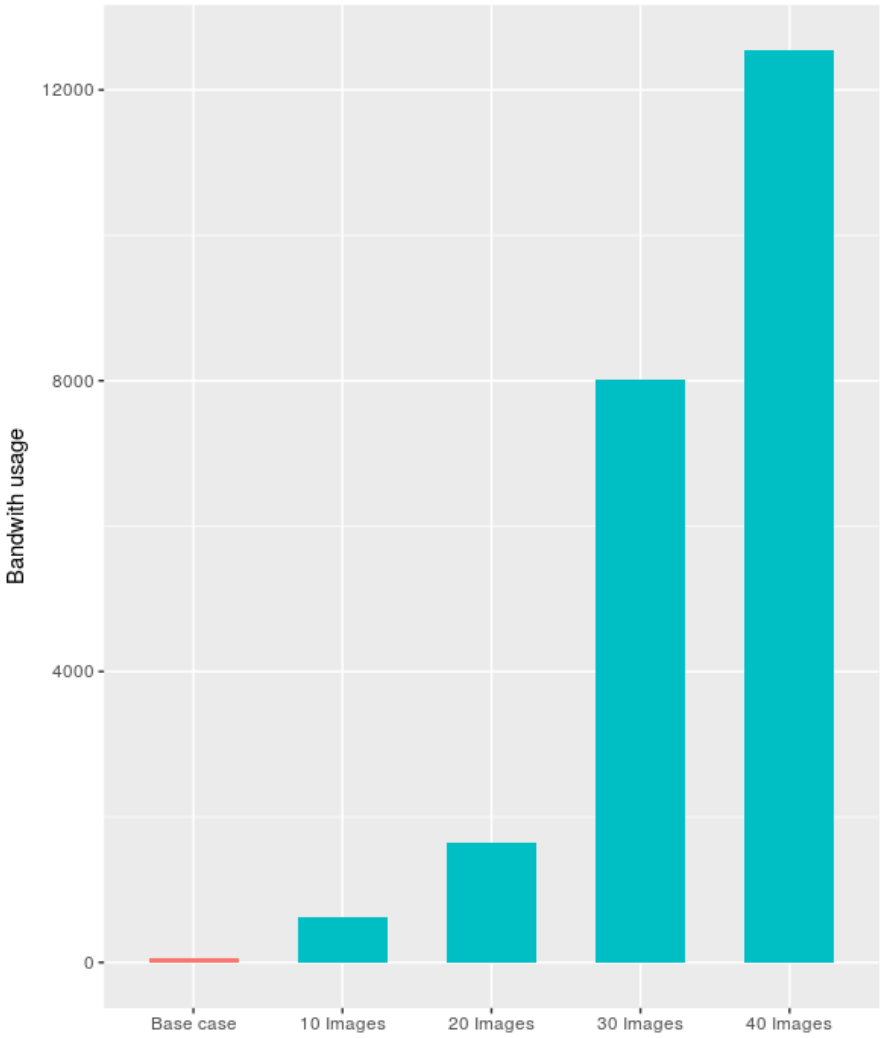

Figure 6.3: Comparison of the bandwidth with the proposed file system structure and without the use of subcatalogs.

Indeed, the catalog need to be downloaded completely before that it can be used, hence increasing the catalog size impact directly with the startup time of the containers. Which is the reason why we decide to use the concept of *super directories*.

CONTAINERS BANDWIDTH USE

In this section we are going to show how the presence of sub-catalogs affect the bandwidth usage. Also in this case we run the python image calling immediately the *quit()* command.

Similarly to the above case we present the case where the catalogs contains 10, 20, 30 and 40 images and we compare it with the baseline case, hence with the case that use the structured showed in this work.

The results are show on Table 6.3 and on Figure 6.3

	Base Case	10	20	30	40
Bandwidth Usage	64.25	624	1651	8020	12540

Table 6.3: Comparison of the bandwidth used to start the containers 100 times with and without use of subcatalogs. Units in MB.

	Storage	.flat	.layers
Size	14	24	26

Table 6.4: Apparent size in GB of the two folder *.layers* and *.flat*

Also in this case we can see how structuring the file system using sub-catalogs improves the bandwidth efficiency.

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 14G calculate with the `du` unix utility.

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 finally show all the measurement on Figure 6.4.

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.

COMPLEXITY

In order to estimate the complexity of managing the file-system we decided to use the cyclomatic complexity of the tool that create and manage the file-system itself that we discuss on Chapter 5. We used *gocylo* [36] a command line tool that calculates the cyclomatic complexity of functions written in the Go(lang) language. We show the result of the cyclomatic analysis on Table 6.5.

While the cyclomatic complexity is very high it 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.

Cyclomatic Complexity	Function
37	ConvertWish
15	ParseImage
14	AddManifestToRemoveScheduler
12	GarbageCollectSingleLayer
12	(Image).GetChanges
12	(Image).downloadLayer
11	SaveLayersBacklink
11	IngestIntoCVMFS
9	requestAuthToken
9	CreateSymlinkIntoCVMFS
9	RemoveDirectory
8	(Image).GetLayers
7	FindImageToGarbageCollect
7	(Image).GetReference
6	AlreadyConverted
6	getBacklinkFromLayer
6	(*execCmd).Start

Table 6.5: Result of the cyclomatic complexity analysis, only function with complexity greater or equal to 3 are shown

7 CONCLUSION AND FUTURE WORK

In this work we have presented a way to merge efficient run time dependency management using containers technologies with efficient content distribution provided by CVMFS.

The proposed methodology exposed on Chapter 4 is based on providing a POSIX file system that container runtime can use to load their content. The use of containers allow to efficiently manage runtime dependencies logically packing them in different layers exposed as directories. We build such file system on CVMFS to exploit its efficient content distribution mechanism. This forced some of our design decision regarding the file system tree like the use of *super directories*, however the exact same structure is applicable also to different file system, not only to CVMFS.

The implementation of the proposed methodology have been explored on Chapter 5 where we have described a software capable of creating and managing the whole proposed file system from ingesting the containers content to deleting it without breaking old images. Moreover we have concluded the work started with the *cvmfs/graphdriver* that provided only a way to use the *thin images* while we finally proposed a methodology to easily produce them starting from standard Docker *fat* image and deploy the *thin images* on standard Docker Registries for an easy distribution.

Finally, on Chapter 6 we have shown the advantages that the proposed system brings in terms of startup time of uncached containers and bandwidth consumption while at the same time not imposing run time penalties.

The final work is in a stable state and there are plans to actually deploy it in production inside CERN, nevertheless several further advancement are possible with respect to increasing the set of supported container run-times or improving the software that manages the file system.

For what concerns the run-times we believe that similar work to the one done for Docker with the *cvmfs/graphdriver* plugin can be done also for different technologies. In particular we are confident that the approach of *thin images* can be adopted also by *containerd* [35], this time implementing a custom *snapshotter* instead of a Docker graphdriver plugin. Different container run-times needs further investigation.

A big limitation to lazily serve the content for containers using lazy systems like CVMFS is the OCI standard. Container run-times that follow the OCI standard do not expose any interface to actually load custom content into the container itself, but they rely simply on tarballs

to provide the content, hence *hacks* like the one in the *cvmfs/graphdriver* are necessary. A solution could be a custom container run-time which provides the possibility to specify how the content should be loaded and from where.

Another improvement to the system is related to the tool that builds the file system structure. It could be re-structured to limit the cyclomatic complexity in order to allow further enhancement. Another big improvement would be to exploit to the fullest the transactional interface of CVMFS. Indeed several transactions are used when ingesting a new image into the file system. An improvement would be to use only a single transaction per image.

APPENDIX

CODE TO GET THE NUMBERS

Listing A.1: 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 A.2: 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 A.3: 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 A.4: 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 A.5: 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 A.6: 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 A.7: 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 A.8: 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
```

BIBLIOGRAPHY

[1] Ali Anwar et al. "Improving Docker Registry Design Based on Production Workload Analysis." In: *16th USENIX Conference on File and Storage Technologies (FAST 18)*. Oakland, CA: USENIX Association, 2018, pp. 265–278. ISBN: 978-1-931971-42-3. URL: https://www.usenix.org/conference/fast18/presentation/anwar.

[2] J. O. Benson, J. J. Prevost, and P. Rad. "Survey of automated software deployment for computational and engineering research." In: *2016 Annual IEEE Systems Conference (SysCon)*. 2016, pp. 1–6. DOI: 10.1109/SYSCON.2016.7490666.

[3] 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.

[4] 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.

[5] Jakob Blomer, Predrag Buncic, and Thomas Fuhrmann. "Decentralized Data Storage and Processing in the Context of the LHC Experiments at CERN." Presented 05 Jul 2012. 2011. URL: https://cds.cern.ch/record/1462821.

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

[7] G Compostella, S Pagan Griso, D Lucchesi, I Sfiligoi, and D Thain. "CDF software distribution on the Grid using Parrot." In: *Journal of Physics: Conference Series* 219.6 (2010), p. 062009. URL: http://stacks.iop.org/1742-6596/219/i=6/a=062009.

[8] 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.

[9] The Linux Foundation. *Image Format Specification*. URL: https://web.archive.org/web/20181122170529/https://github.com/opencontainers/image-spec/blob/master/spec.md (visited on 11/22/2018).

[10] 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).

[11] 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).

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

[13] 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.

[14] 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.

[15] Amazon Web Services Inc. *Cloud Object Storage | Store and Retrieve Data Anywhere | Amazon Simple Storage Service*. URL: https://web.archive.org/web/20181128173120/https://aws.amazon.com/s3/ (visited on 11/28/2018).

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

[17] 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).

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

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

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

[21] 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).

[22] Red Hat. Inc. *Ansible is Simple It Automation*. URL: https://web.archive.org/web/20181127163843/https://www.ansible.com/ (visited on 11/27/2018).

[23] W. Kangjin, Y. Yong, L. Ying, L. Hanmei, and M. Lin. "FID: A Faster Image Distribution System for Docker Platform." In: *2017 IEEE 2nd International Workshops on Foundations and Applications of Self* Systems (FAS*W)*. 2017, pp. 191–198. DOI: 10.1109/FAS-W.2017.147.

[24] James H. Morris, Mahadev Satyanarayanan, Michael H. Conner, John H. Howard, David S. H. Rosenthal, and F. Donelson Smith. "Andrew: A Distributed Personal Computing Environment." In: *Commun. ACM* 29 (1986), pp. 184–201.

[25] 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).

[26] Bogdan Nicolae, John Bresnahan, Kate Keahey, and Gabriel Antoniu. "Going Back and Forth: Efficient Multi-Deployment and Multi-Snapshotting on Clouds." In: *The 20th International ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC 2011)*. San José, CA, United States, June 2011. URL: https://hal.inria.fr/inria-00570682.

[27] The Debian Project. *APT - Advanced Package Tool*. URL: https://web.archive.org/web/20181128175621/https://wiki.debian.org/Apt (visited on 11/28/2018).

[28] Puppet. *Deliver better software, faster | Puppet*. URL: https://web.archive.org/web/20181127164038/https://puppet.com/ (visited on 11/27/2018).

[29] 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).

[30] 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).

[31] 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).

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

[33] ThinkParQ and Fraunhofer. *BeeGFS - The Leading Parallel Cluster File System*. URL: https://web.archive.org/web/20181128172734/https://www.beegfs.io/content/ (visited on 11/28/2018).

[34] Seth Vidal. *yum - Yellowdog Updater, Modified*. URL: https://web.archive.org/web/20181128175354/http://yum.baseurl.org/ (visited on 11/28/2018).

[35] The containerd authors. *containerd -An industry-standard container runtime with an emphasis on simplicity, robustness, and portability*. URL: https://web.archive.org/web/20181128165122/https://containerd.io/ (visited on 11/28/2018).

[36] fzipp. *gocycle, Calculate cyclomatic complexities of functions in Go source code*. URL: https://web.archive.org/web/20181124194738/https://github.com/fzipp/gocyclo (visited on 11/24/2018).

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.