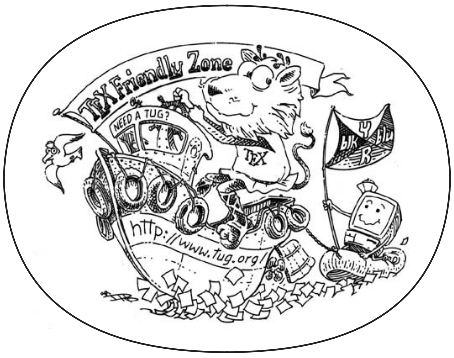
ANDRÉ MIEDE & IVO PLETIKOSIĆ

A CLASSIC THESIS STYLE

A CLASSIC THESIS STYLE

ANDRÉ MIEDE & IVO PLETIKOSIĆ



An Homage to The Elements of Typographic Style

June 2018 – classicthesis v4.6

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

TITOLO DELLA TESI

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

Tesi di Laurea di:
Simone Mosciatti Matr. 878758

Anno Accademico 2017-2018

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

— Lilo & Stitch

Dedicated to the loving memory of Rudolf Miede.

1939 – 2005

ABSTRACT

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

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

ZUSAMMENFASSUNG

Kurze Zusammenfassung des Inhaltes in deutscher Sprache. . .

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

— 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)

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

1 INTRODUCTION

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

Its mission is to:

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

It host the instrument of the 4 biggest physic collaboration:

- ALICE
- ATLAS
- CMS
- LHCb

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

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

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

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

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

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

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

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

WLCG

The Worldwide LHC Computing Grid is an global collaboration of more than 170 datacenters in 42 countries. The mission of the WLCG is to provide the computing resource to store, distribute and analyze the data from the operation of the LHC.

The organization of the WLCG follow a hierarchical model, where each level is called "Tier" The most central Tier is the Tier-0 hosted by CERN in the Geneva Area and in Budapest. There are 13 Tier-1 datacenter with enough storage and computing capabilities to support the Grid operation around the clock, the Tier-1 are connected to the Tier-0 with at least 10Gb/sec links. Tiers-1 are geographically distributed, 8 of them are in Europe, 3 in the North American and the rest in Asia. Finally the Tier-2 datacenter do not have strict requirements and are generally operated by research centers and universities.

The work on the WLCG is mostly divided in two big classes: analysis of the data from the LHC detectors and Monte Carlo simulations. The WLCG assume and support a batch computing paradigm. Analysis are splitted in smaller jobs that are distributed to different servers that can work in parallel.

Before to start each job it is necessary to install the software on the server. Unfortunately the amount of software potentially needed in each computing node and the velocity at which the software is updated makes the installation challenging. Moreover simpler installation techniques that relies on packages managers are not applicable since they would put the package managers themselves under too much load.

Several solution have been proposed and used, but eventually it settled for the use of CernVM-FileSystem (CVMFS).

CVMFS

CernVM-FileSystem 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 (Filesystem in USErspace) and standard webserver technologies such as Apache or NGNIX.

CVMFS organize its content in repositories where we can approximate each repository as a CVMFS instance.

CVMFS is engineered to support repository of size on the order of the Terabyte with billions of files.

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

The content of the files are served using the HTTP protocol by a standard webserver. 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 filesystem. 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 filesystem 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.

CONTAINERS

While CERN solved its problems of software distribution with CVMFS the industry opted for a different approach, containers.

Containers are a standard unit of software that packages up code and all its dependencies so that computer applications run quickly and reliable from one computing environment to the others.

There are two implementation of containers technology widely used at CERN: Singularity and Docker.

Singularity

Singularity focus on running computation on HPC clusters, its main advantage is to be able to run a containers from a standard directory. It is sufficient that all the content of the containers is unpacked on a single directory to run the container itself using Singularity. Finally, Singularity does not rely on a daemon running in the host machine unlike docker.

Singularity is able to run specific Singularity images as well as more generic Docker images.

Given the capability of Singularity to bootstrap the containers from a simple directory the integration with CVMFS is straightforward: create on CVMFS a directory for each kind of Singularity container we wish to run. The Singularity software will read those directories and run the correct container.

Docker

Docker is more widespread in the industry and benefits from more mature tools and ecosystem, however docker needs root permissions in order to run a container and a background daemon (dockerd.) Moreover the filesystem of docker containers is encapsulated inside the docker daemon which needs to download it from an external sources, uncompress it, cache it, and mount it.

Given the advantages and widespreads of containers they started to be used also inside CERN and integration with the CERN existing software distribution infrastructure (CVMFS) was required. Without the integration with CVMFS, to run a container it is necessary to download all the container content on the machine, while it has been showed that, on average, only 7% of the content of containers is actually needed. The integration would allow to download only the strict necessary files.

The integration with docker was more complex. In this work we are not concerned about the implementation details of the integration between docker and CVMFS, but we are concerned with the interface of such integration.

It is not possible to host docker layers on CVMFS and use them directly. The solution was to create a new kind of images, called "Thin-images". Thin-images consist in only a file, the "recipe", that describe the content of the original image in terms of layers. The recipe file is then read by a docker plugin. The plugin makes the content of the original image available to the docker daemon through CVMFS. This procedure allows to run docker images without the need to download all the files in the image itself. The creation of a thin-images consist in making the several layer of the original docker images available in CVMFS, create the "recipe" file which contains the location of all those layers, and finally create a docker image that contains only the "recipe" file.

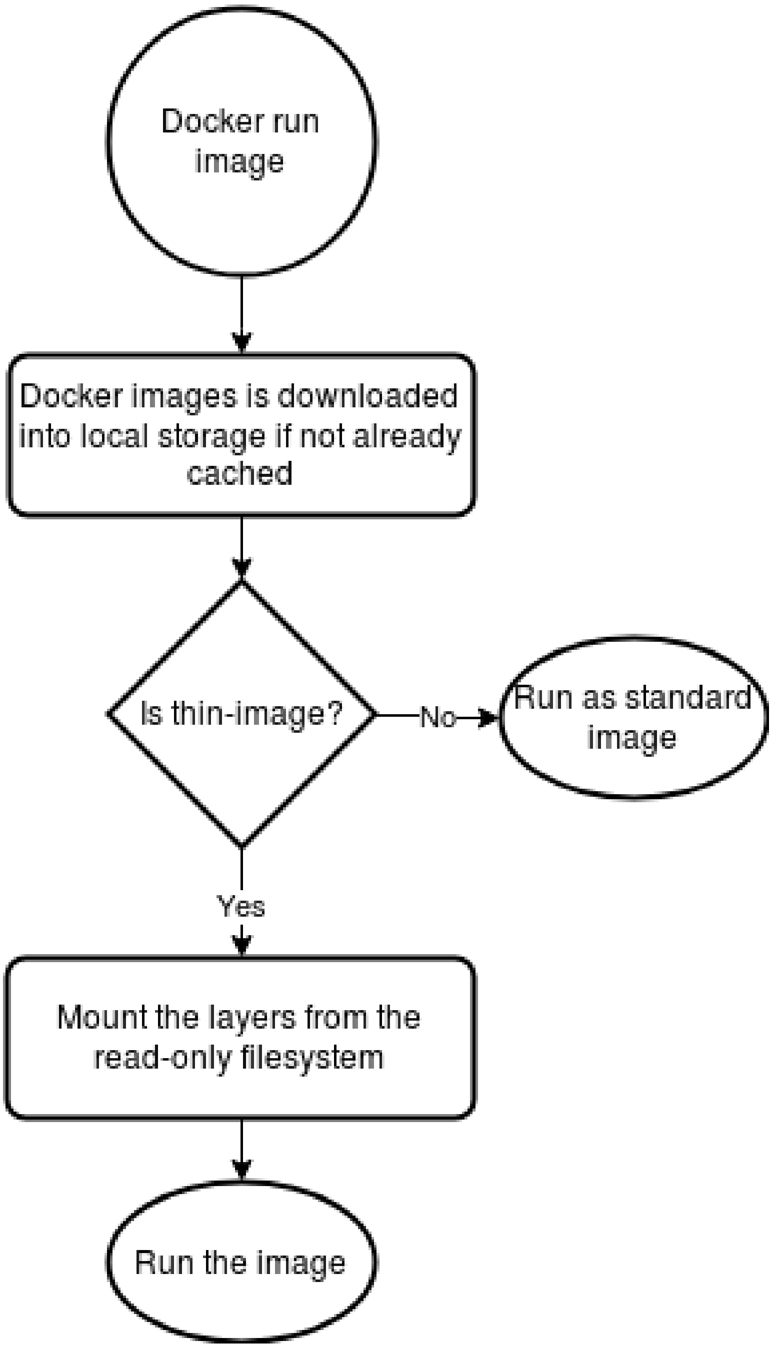


Figure 1.1: Decision proces for running docker thin images

Containers distribution

Containers can be distributed to the host in several ways, in this works we are mostly concern with the simplest alternative, the use of registries and, specifically, the use of docker registries.

The content of a container is packed in "layer", each layer stores the "delta" between a computing environment and the other, an "image" is an immutable set of layers. Containers are an instances of images.

Each layer is identify by its digest. The whole content of the layers is hashed (generaly using the hash256 function) and the result of the hash function is the digest of the layer.

The same schema is adopted for images as well, indeed, each image is identifies by the digest which is the hash of all its content.

An image is defined by it's manifest, that declares the digest of the image itself, and the set of layers that compose the image, the layers are referenced by their digests.

The layers are distributed, generally, over the network as compressed tar archives that need to be downloaded, then uncompressed and potentially cached.

The components responsible for distributing the layers are the registries. The registries expose an HTTP interfaces that serve the required layer as compressed tar archive.

Docker images are served by docker registries. Moreover, since Singularity can run Docker images, one of the most common way to share Singularity images is to make them available as normal docker images into standard docker registries.

At runtime layers are composed one on top of each other to re-create the original environment where to execute the application.

This approach to container distribution is standardize in the OCI standard.

While containers ensures a complete reproducibility of the environment every files in every layer need to be present in the hosting machines, i.e. in the machine running the application.

2 STATE OF THE ART

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

ANSIBLE

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

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

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

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

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

PUPPET

Puppet is another declarative provisioning system.

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

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

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

KUBERNETES

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

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

PACKAGE MANAGERS

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

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

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

ALICE SOFTWARE INSTALLATION SYSTEM

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

DIFFERENCE WITH CVMFS

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

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

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

3 PROBLEM DEFINITION

The goal of this work is to describe a suitable structure of a read only filesystem to host containers images.

The result of this work has been applied at CERN to distribute containers images in a global read-only filesystem, CVMFS.

In the context of CVMFS we aim at:

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

4 METHODOLOGY

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

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

The second section will analyze how we host 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.

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

SINGULARITY IMAGES

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

As mentioned in 1.3.3 docker images have a hierarchical structure. The first level of the hierarchy is the docker registry where the image is hosted. The most common registries in our case are the official docker hub [1] and the CERN internal registry [2].

The second level in the hierarchical structure is the namespace of the docker image. If the image is one of the official docker images it will be the standard namespace: "library". In all the other cases, the namespace will be the same as the original docker image. For

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

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

Figure 4.1: Visualization of the Filesystem structure, the arrows indicate symbolic links

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 users.

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

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

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

We show the directory structure of the file-system on figure 4.1.

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

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

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

Figure 4.2: Visualization of the "super directories" in the ".flat" subdirectory

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 4.2 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".

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 1.3.2 the recipe of the docker thin-image contains the path of the directories where each layer of the original docker image is hosted, those directories will be mounted by the docker plugin.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

When we decide to remove an image, for any layer we check that it is used only by the image we want to remove, if this is the case, we

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
```

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
```

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
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 4.3

```
/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 4.3: 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 4.4: Structure of the `.metadata/` directory

keeping track of the work already done

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

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

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

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

As already mentioned in 1.3.3 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 is possible to understand if the image should be updated or nor.

The structure of the `.metadata/` folder is show in figure 4.4.

closing remarks

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

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

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

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

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 decide to ingest Singularity images.

We will move on to describe the challenges in ingesting docker images while transforming them into docker thin-images, hence we will explore the custom modification done to CVMFS in order to accomodate 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 filesystem.

Finally we will show the interface to the administrator.

CVMFS WRITE INTERFACE

CVMFS provides a transactional interface, hence is possible to open a transaction, modify the filesystem 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 modifications of the filesystem can be carried out with standard linux commands or even using a graphical file explorer. Moreover is possible to test locally the new filesystem without actually commit the changes.

Finally is important to keep in mind that only a single transaction can be open at any given time, the system 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 order to minimize the possibilities of inconsistency or of error. Once the download finish successfully 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 describe above.

This few step are sufficient to make the Singularity images available through CVMFS.

Docker Ingestion

Ingest docker images is much more complex than ingesting the Singularity one.

The first big difference is that during the ingestion is necessary to create also the thin images that then will be distributed through the standard docker registries.

Another difference is the necessities to keep reference to the layers so that is possible to delete layers when they are not necessary anymore.

CVMFS Ingestion of Tarball

In order to make more ergonomic the ingestion of docker images we decide to add a new command to CVMFS, the 'ingest' command.

The 'ingest' command takes as input a tarball and a location. The command expand the tarball into the location materializing all the files.

The command automatically open and commit the transaction, hence is possible to have only a single concurrent ingestion.

Docker Ingestion Algorithm

The first step is to download the manifest of the docker image.

In the manifest is describe which layers compose the image, hence we start to download each layer.

The layers are all stored first in a temporary location, then each one of them is ingested inside the CVMFS file system using the 'ingest' command under the subdirectory ".layers" as described above.

With the information about the layers necessary for an image we can start to create the docker thin image, as soon as the thin-image is created we proceed to upload it to the docker registry.

Then, after all layers have been ingested we write the references necessary to delete the layers in an hidden folder.

Finally, if no error happened during the whole process we store, in another hidden folder, information about the successful conversion and ingestion of the image.

garbage collection of images

We have only described how we add new images to the CVMFS filesystem, however updating an image is quite common, especially if the images are refered 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 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 filesystem. The remove-schedule.json files contains a collections of the manifest of all the images we are not interested in anymore.

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

The remotion 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 filesystem. If the list is now empty we proceed to remove also that specific layer from the filesystem.

administrator interface

In order to store images in the CVMFS filesystem is necessary to know what image store, in which repository store it and how to call the respective thin-image.

We decide to call the triplet (Input image, CVMFS Repository, Output image) a “wish”.

Now, we need a way 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.

The syntactical transformation depends on the input image and is applied to obtain the final name of the output image.

There are several benefits of this approach:

1. The YAML file is a simple text file that is simple to understand and to edit

2. It can be hosted on version control system

Since the wish list can be hosted on a version-control system like Github or Gitlab and since it is a simple file to modificate it allow a Pull-Request based approach to modificate the wish list itself.

Users who wish a new image to be added to the repository can simply make a pull request adding their image to the wish list. The administrator of the system act as a gatekeeper, inspect the image that is been required to be add and decide if add the image or not.

6 RESULTS

In this section we explore the result of this works.

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

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

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

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

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

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

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

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

SPACE REQUIREMENT

CONTAINER STARTUP TIME

FILE IN THE SUB-CATALOGS

COMPLEXITY

7 FUTURE WORKS

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

DECLARATION

Put your declaration here.

Saarbrücken, June 2018

André Miede & Ivo Pletikosić

COLOPHON

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

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

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

http://postcards.miede.de/

Thank you very much for your feedback and contribution.