CVMFS Containers integration

Updates on the integration between CVMFS and container technologies

Simone Mosciatti CERN EP-SFT
CVMFS Workshop 01 - Feb -2021

Outline

1. Provide background information and refresh technical concept in container technology
2. CVMFS and container from the client side
3. CVMFS and container from the server side

What are containers

- Containers encapsulate an environment for a specific computation
- Mostly linux cgroups and tarball
- The filesystem of a container is distributed as a tarball
- Requires all the tarballs to be download and stored in the local disk

Why containers on CVMFS

- OCI distribution format based on tarball is sub-optimal
  - Identical files in different layers are replicated
  - All the filesystem must be available before running the container
- CVMFS perfectly address those tarball replication
  - Provides file deduplication
  - Provides lazy loading of files
- Disk storage is less and less a concern, but bandwidth keep being an issue

Filesystems implementation for containers

- Layers are diff between the snapshot of two filesystems
- Applying a layer on top of a filesystem create a new filesystem
- This process is deterministic
- The OCI standard calls `Chain` the snapshot of a filesystem
- Applying a layer to a `Chain` creates a new `Chain`

Different containers runtime

In the HEP community we are interested in mostly:

- Singularity (to run computation on the WLCG)
- containerd (to use k8s as orchestrator)
- podman (since it is the container runtime pushed by RedHat)

Different storages schemas

- Those runtimes 3 implement 2 different storage schemas.
- Layer based
  - All the layers need to be present in the filesystem
  - The runtime mounts each layer in an overlay filesystem
  - podman
  - containerd
- Flat filesystem based
  - The runtime mounts a single directory that contains all the filesystem of the container image
  - Singularity

Different kind of container images

- **Base images**
- Changes rarely, small, most of other images are based on those
  - library/ubuntu
  - library/centos

- **Experiment images**
- Changes rarely, big, used as base by users
  - atlas/athena

- **Users images**
- Change often, big, created by user, based on Experiment or Base images

Client part

Reading containers filesystem from CVMFS

- The first step is to allow containers runtime to use the filesystems stored in CVMFS
- We can successfully interface with 3 different container runtimes
  - Singularity
  - containerd
  - podman
- Each requires different structures

Singularity

- The most widely used container runtime in HEP
- Simple clients implementation
- It can just read a directory as base filesystem for the container
- Only need the filesystem of the container in CVMFS

Singularity

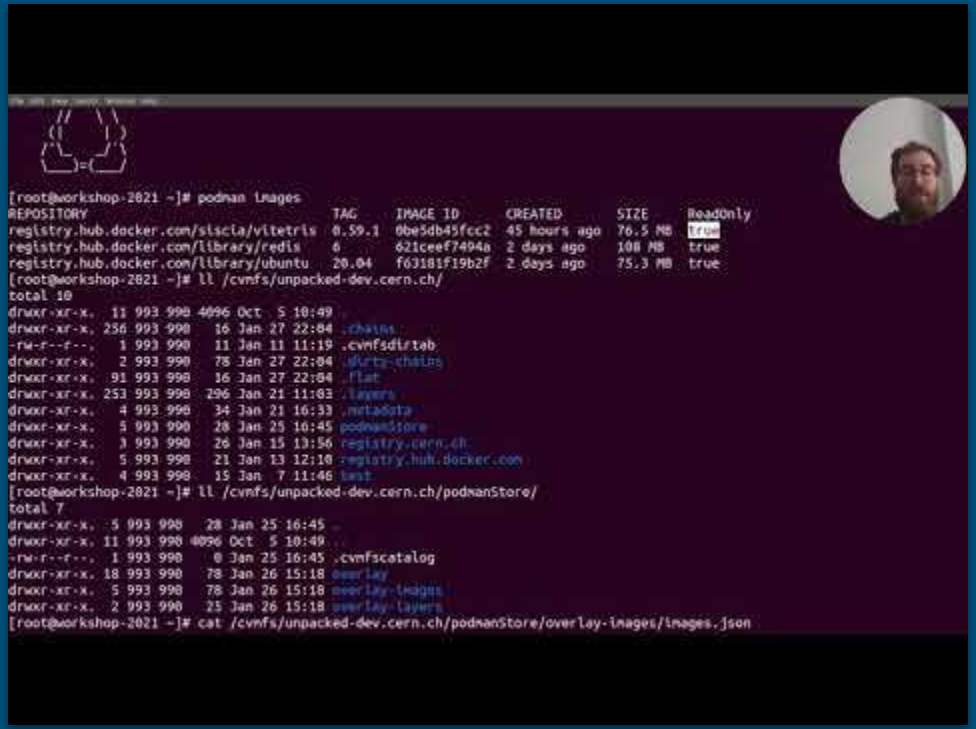
containerd

- Great interest due to Kubernetes
- Allows kubernetes to spawn containers
- Requires the layers of the image to be available in CVMFS
- Requires an extra piece of software
  - cvmfs-containerd-snapshotter
- The snapshotter mounts the layers from CVMFS in a filesystem that containerd can use
- Requires small configuration changes

podman

- Most experimental one
- Does not requires any additional piece of software beside podman itself
- Share CLI with the more common docker
- Can run rootless
- Requires configuration changes

podman


[root@workshop-2021 ~]# podman images
REPOSITORY TAG IMAGE ID CREATED SIZE ReadOnly
registry.hub.docker.com/siscia/vitetris 0.59.1 0be5db45fcc2 45 hours ago 76.5 MB true
registry.hub.docker.com/library/redis 6 621ceef7494a 2 days ago 108 MB true
registry.hub.docker.com/library/ubuntu 20.04 f63181f19b2f 2 days ago 75.3 MB true
[root@workshop-2021 ~]# ll /cvmfs/unpacked-dev.cern.ch/
total 10
drwxr-xr-x. 11 993 990 4096 Oct 5 10:49 .
drwxr-xr-x. 256 993 990 16 Jan 27 22:04 .chains
-rw-r--r--. 1 993 990 11 Jan 11 11:19 .cvmfsdirtab
drwxr-xr-x. 2 993 990 78 Jan 27 22:04 .dirty-chains
drwxr-xr-x. 91 993 990 16 Jan 27 22:04 .flat
drwxr-xr-x. 253 993 990 296 Jan 21 11:03 .layers
drwxr-xr-x. 4 993 990 34 Jan 21 16:33 .metadata
drwxr-xr-x. 5 993 990 28 Jan 25 16:45 podmanStore
drwxr-xr-x. 3 993 990 26 Jan 15 13:56 registry.cern.ch
drwxr-xr-x. 5 993 990 21 Jan 13 12:10 registry.hub.docker.com
drwxr-xr-x. 4 993 990 15 Jan 7 11:46 test
[root@workshop-2021 ~]# ll /cvmfs/unpacked-dev.cern.ch/podmanStore/
total 7
drwxr-xr-x. 5 993 990 28 Jan 25 16:45 .
drwxr-xr-x. 11 993 990 4096 Oct 5 10:49 ..
-rw-r--r--. 1 993 990 0 Jan 25 16:45 .cvmfscatalog
drwxr-xr-x. 18 993 990 78 Jan 26 15:18 overlay
drwxr-xr-x. 5 993 990 78 Jan 26 15:18 overlay-images
drwxr-xr-x. 2 993 990 25 Jan 26 15:18 overlay-layers
[root@workshop-2021 ~]# cat /cvmfs/unpacked-dev.cern.ch/podmanStore/overlay-images/images.json

Server part

Storing filesystems in CVMFS

- We can run containers from content stored in CVMFS
- How to put that content in CVMFS
- DUCC automatize the process for us

Ingestions of Layers

- Layers are ingested one by one
- Concurrent download and ingestion of layer
- Using the `cvmfs_server ingest` command to be as close as possible to the original file and permissions
- Layers stored in .layers

Ingestions of Flat filesystem

- Most time consuming step
- A lot of work is replicated

Fast flat filesystem and chains

- Caching already done work to avoid repeating that
- Storing each intermediated chains on the filesystem
- When a new images need to be ingested, we first check if any of the chain it is based upon are already in CVMFS
- If there is at least one layer, we start the ingestion from that one
- Otherwise we start from scratch

Sneaky ingestion

- Layers can delete content from Chain
- We would need to implement the whole overlay logic in user code
- What if we use the Linux overlayfs engine
- Before to open a transaction replicate the content of the layer in /var/spool/cvmfs
- Then CVMFS will pick-up whatever change in the spool area
- Only need to translate from OCI-overlay format (AUFS) to the overlayfs format

Fast and simple Singularity & CVMFS

- ChainID + sneaky ingestion
- Fast and simple to ingest whole filesystems in CVMFS
- The DUCC code got rid of the singularity dependency

DUCC operations

- Wishlist based
  - Users list all the images they need
  - Periodically DUCC check that all the images listed are up to date in CVMFS
  - Base of unpacked.cern.ch
  - ~ 700 images
- Single image conversion
  - A single image can be converted by DUCC
  - If the image is not up to date, it is upgraded
  - Useful to integrate simple workflows in bash
  - Testing on unpacked-dev.cern.ch

unpacked-dev.cern.ch

- Experimental place to support user level containers
- The intention is to learn from it, and then upgrade unpacked.cern.ch
- Uses chains + sneaky layers
- Based on webhooks notifications
- Need to be fast
  - If the ingestion time is close to the replication delay of CVMFS, it is fast enough

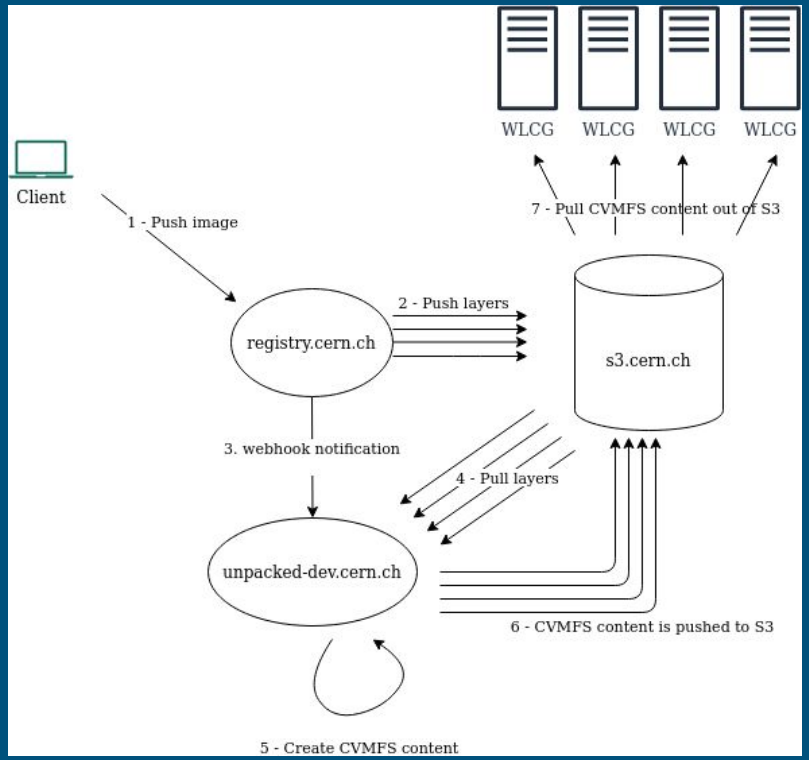
DUCC webhooks integration

- Need to scale the amount of images that can be ingested
- Difficult to maintains and cumbersome to update
- Integration with docker registry webhook
- When a new image is pushed, a webhook is invoked
- The webhook payload is parsed
- The image name is passed to DUCC
- DUCC convert the newly pushed image
- Work on unpacked-dev.cern.ch

Repository as wishlist

- Parsing webhook can be unreliable
- Experimenting with keeping the docker registry as wishlist
- Periodically check if all the images in the registry are in CVMFS
- If not ingest them again

Webhook implementation in unpacked-dev.cern.ch



- `unpacked-dev.cern.ch` synchronize with registry.cern.ch a harbor base docker registry
- Harbor docker registries can mirror other docker registries like the gitlab one or the docker-hub

Ingestion performances

- Overall reasonable ingestion performance
- Goal is to be close to the replication delay of CVMFS

Image	Size	Time
atlas/athanalysis:21.2.155	~ 6GB	~ 5 min
atlas/athena:21.0.77_300.0.1	~ 32GB	~ 26 min

- We consider ~1GB / min

Thank you!