Docker Implementation
Version 3

TRIUMF ATLAS Tier-1

by

Denice Deatrich

Last update:

2017-10-27

Index

1 Introduction

The time has come to investigate using Docker, a Containers-based technology, for worker node deployment. Worker nodes, because of their uniformity in configuration and requirements, are good candidates for Containers. With KVM virtualization for example, there is much more overhead in process, memory and disk space running KVM guests for each worker. With Containers a much more light-weight environment is needed to sustain a worker implementation – system-wide changes are shared – only the application is sand-boxed.

This document summarizes the initial setup and experience using Docker for a worker node infrastructure. As always at the Tier-1 we like to initially investigate new technologies assuming a 'scratch' implementation, where we start from the basics to better understand it. Thus, though I show how to access external Docker images, I focus instead on building and deploying our own.

Because we now use Ansible to manage server configurations this document will explain deployment by showing both the commands and configurations needed, as well as some associated Ansible role-based task snippets.

The Docker infrastructure documented here includes installation, configuration and run-time examples for:

- Docker Engine – the software running on the host that sponsors containers
- Docker Registry/Distribution – software that provides storage, searching and retrieval of docker images
- Image building – how to build you own container images
- Docker Containers – run-time considerations

2 Getting Docker

The Red Hat implementation of Docker is not deployed in the mainstream installation base. Instead it is made available by Red Hat in their 'extras' channel – this description is on their web site[1]:

> *Red Hat is introducing a new channel in the Red Hat Customer Portal for Red Hat Enterprise Linux called "Extras." The Extras channel is intended to give customers access to select, rapidly evolving technologies. These technologies may be updated more frequently than they would otherwise be in a Red Hat Enterprise Linux minor release. The technologies delivered in the Extras channel are fully supported.*
>
> *Over time, these technologies will continue to mature and stabilize and may eventually be added to the Red Hat Enterprise Linux channel to which the Red Hat Enterprise Linux life-cycle policies apply.*

1 https://access.redhat.com/support/policy/updates/extras

Note that this does not guarantee that the RHEL implementation of Docker will always be there. At this point in time, Docker for RHEL **7** is re-built and published by both CentOS and Scientific Linux (SL). As the CentOS versions were more recent at the time of initial testing they have been mirrored by us, and the software can be installed at the Tier-1 using our ***t1-release-rh-extras*** RPM.

This document is based on our experience running Docker on SL; however as usual it applies equally to a CentOS experience.

Docker Engine RPMs for SL and CentOS **6** can be found in EPEL, however they are much older versions, and have no supported path forward[1]. However Docker on SL7 systems can run both SL6 and SL7 application containers. You give up some disk space in running SL6 containers on an SL7 engine because you necessarily need to install SL6-based supporting libraries for any SL6 container.

3 Pre-install Checklist

3.1 Storage preparation

The default behaviour of the Red Hat RPMs is to use loopback devices if you have made no provision for storage for your containers. Loopback devices should not be used in production; you will get a warning when you use them:

> *Usage of loopback devices is strongly discouraged for production use. Either use `--storage-opt dm.thinpooldev` or use `--storage-opt dm.no_warn_on_loop_devices=true` to suppress this warning.*

Moreover, the server should be configured with the necessary *spare* disk space, either as complete LUNs or devices, or as complete volume groups. In this document I use a spare volume group named ***vg1*** for our thin-provisioned storage for the running containers.

Another issue is local docker image storage. Each *worker* docker engine may have a number of pulled local images for testing and validation. Rather than storing these in the default location, /var/lib/docker, I decide to locate them in a specific directory not used by the OS, with adequate space. It is mounted as ***/docker***

3.2 Network setup

I did not want to use the default bridging setup on **172.17.0.0/16** provided by the docker RPMs – this is an issue to revisit later. In initial testing that private IP space is reported to the condor server remotely, where that space is not route-able. Instead I decided to bridge to our existing private network on **10.0.0.0/16**, or **10.1.0.0/16** which are accessible to grid servers in the data centre. To that

1 https://access.redhat.com/solutions/1378023

end I configure the bridge on the Docker engine node ahead of time using an Ansible role. In this document the name of the bridge is ***br0***.

In our data centre each blade chassis housing worker nodes has a local /24 sub-net range in our private 10.0.0.0/16 space – e.g. 10.0.3.0/24. At this early phase of testing I further restrict the IP range for running containers to a fixed CIDR-range in a unique /30 group for each host within the blade chassis /24 group. One of the advantages of this setup is that running containers have a well-known IP address and hostname. As you will see, the container sets it hostname accordingly, and Ansible renames the container to match that hostname.

10.0.0.0/16 → 10.0.3.0/24

→ 10.0.3.1 (Engine 1 of 14)

→ 10.0.1.128/30 → containers:10.0.3.128, 10.0.3.129

→ 10.0.3.2 (Engine 2 of 14)

→ 10.0.1.132/30 → containers: 10.0.3.132, 10.0.3.133

→ ...

4 Engine Installation

4.1 Docker Engine installation and configuration

For the Tier-1 I install the local release rpm on the Docker Engine worker node host for the Red Hat Extras repository, and then install the docker packages. On SL7, your yum configuration must also enable SL 'fastbugs'. In case you normally do not enable it, then it must also be enabled:

```
# yum install t1-release-rh-extras
# yum –enablerepo=rh-extras,sl-fastbugs install docker atomic python-docker-py
```

A few configuration files need to be modified before you enable and start docker in daemon mode.

These files need to be modified to address issues specific to our data centre. The configuration file differences are enumerated below; an Ansible Docker Engine role was used to configure the host:

4.1.1 /etc/sysconfig/docker-network

```
## this is just an example for this test blade
# diff docker-network docker-network.original
2,3c2
< DOCKER_NETWORK_OPTIONS="--ipv6=false --mtu=9000 –bridge=br0 --fixed-
cidr=10.0.3.144/28"
<
---
```

```
> DOCKER_NETWORK_OPTIONS=
```

- I want a customized network bridge setup
- IPv6 is disabled at this time
- You need to explicitly give the MTU if it is not the default 1500

4.1.2 /etc/sysconfig/docker-storage-setup

```
# diff docker-storage-setup docker-storage-setup.original
5,7d4
<
< VG='vg1'
<
```

- I use the free volume group here which was provisioned during kickstart

4.1.3 /etc/sysconfig/docker

```
# diff docker docker.original
4c4
< OPTIONS='--selinux-enabled=false -g /docker'
---
> OPTIONS='--selinux-enabled'
13d12
< ADD_REGISTRY='--add-registry docker-registry.lcg.triumf.ca'
19c18
< BLOCK_REGISTRY='--block-registry docker.io'
---
> # BLOCK_REGISTRY='--block-registry'
34c32
< DOCKER_TMPDIR=/var/tmp
---
> # DOCKER_TMPDIR=/var/tmp
44,47d41
```

- during initial testing I turn SELinux off
- I move the root of the docker daemon run-time to */docker* ; as already mentioned the default otherwise is */var/lib/docker*. We need some space to grow while we understand how many images we need to keep locally on the node.
- I block access to images at docker.io[1]. Instead we will use a local, private registry running on host docker-registry.lcg.triumf.ca which is firewall ed off from the world at this time. By default Docker registries run on port 5000, but can obviously run on any unprivileged port. We will secure the registry host on port 443. If we did not, and used an insecure port then it would need to be explicitly labelled *insecure* in the configuration before a Docker engine can successfully interact with it.

1 https://docs.docker.com/docker-hub/repos/

- I switch to using /var/tmp for temporary files; the default temporary file location would otherwise be /var/lib/docker/tmp/

Once these files are changed one can enable and start docker:

```
# systemctl enable docker
# systemctl start docker
```

Further configuration is needed if you will only accept digitally-signed images. That part is covered in the image signing chapter.

4.1.3.1 External Images

For the Tier-1 data centre, our light-path nodes cannot get to the commercial network – docker.io or redhat.com - without a proxy. This is possible in the /etc/sysconfig/docker with a setting like:

https_proxy=somehost.lcg.triumf.ca:3136

However, this is a global docker setting, meaning that the same proxy would also be used to get to any local private registries, which may not be desired.

5 Docker Registry installation and configuration

There are two branches of software that provide a Docker Registry:

1. The legacy registry (v1) which you get when you install the *docker-registry* RPM
2. The next generation (v2) registry provided by the *docker-distribution* RPM

It is time to move on to Docker Distribution – v1 is deprecated, and is ugly to configure. Note however that Docker Distribution is *still* missing a functional search interface[1]; this is tracked in this Github issue URL:

https://github.com/docker/distribution/issues/206

In the previous version of this document[2] skeletal instructions on setting up a Docker Registry were added. Here we move on to the v2 configuration. Though the phrase 'registry' is often used in this document we are now referring to the v2 registry.

I use host docker-registry.lcg.triumf.ca for an initial local test v2 registry. The registry daemon is configured to run on the secure port, 443 rather than the default port, 5000. Only data centre nodes are allowed to access this port at this time, but when it is moved to a permanent host and home it could be opened up.

1 https://bugzilla.redhat.com/show_bug.cgi?id=1277572
2 https://twiki.atlas-canada.ca/pub/AtlasCanada/TRIUMFDocker/Tier1Docker-v2.pdf

5.1 Trust issues

The goal with running a private and/or local registry is to consider some trust issues like securely signing images, as well as securely transporting images. Signing images is a work in progress – there are two technologies for image signing – Project Atomic and Docker Notary. We use 'atomic' for image signing, and our implementation is documented in the next chapter.

Here are a some useful links to read concerning container security:

- https://blog.docker.com/2013/11/introducing-trusted-builds/
- https://access.redhat.com/blogs/766093/posts/1976473
- https://titanous.com/posts/docker-insecurity
- https://thenewstack.io/assessing-the-state-current-container-security/
- http://rhelblog.redhat.com/2015/09/03/what-is-deep-container-inspection-dci-and-why-is-it-important/

5.2 Secure Transport

While you can install an insecure (http-type) registry, you will be limited with authorization options in such a setting. Instead we will enable a TLS https-type registry using a host x509 certificate signed by the national grid certificate authority, in our case Grid Canada[1]. With grid CRLs in place via the fetch-crl RPM, and the host certificate and key in place as /etc/grid-security/hostcert.pem and /etc/grid-security/hostkey.pem we can use this configuration in the config.yml file below to secure our registry.

5.2.1 Configuring docker client access

There is a client-side issue that we must take care of for any host that accesses the registry when using non-commercial x509 certificates – make sure that the CA certificate is known to the host.

There are two ways of doing this. I describe both, but favour the second option for its simplicity and universality.

5.2.1.1 - Adding a configuration to /etc/docker/

One way of doing it is to put the certificate in the /etc/docker tree. If you are missing it, then you may see this message:

```
x509: certificate signed by unknown authority. If this private registry
supports only HTTP or HTTPS with an unknown CA certificate, please add
`--insecure-registry docker-registry.lcg.triumf.ca:5000` to the daemon's
arguments. In the case of HTTPS, if you have access to the registry's CA
certificate, no need for the flag; simply place the CA certificate at
```

1 https://cert.gridcanada.ca/pki/pub

/etc/docker/certs.d/docker-registry.lcg.triumf.ca/ca.crt

Thus we could create the directory corresponding to the fully-qualified host name and port, and copy the CA certificate into place:

```
# cd /etc/docker/certs.d/
# mkdir docker-registry.lcg.triumf.ca:443
# cd docker-registry.lcg.triumf.ca:443
# cp -p /etc/grid-security/certificates/GridCanada.pem  ca.crt
```

5.2.1.2 - Modifying the PKI ca-trust

This option is not docker-specific, and is also easier:

```
# cp -p /etc/grid-security/certificates/GridCanada.pem /etc/pki/ca-trust/source/anchors/
# /usr/bin/update-ca-trust
```

5.3 Storage

Before installing the software a file system is created with enough grow-able space for the initial registry environment. The mount point is named */data/docker/* and it will house the non-registry container images as well as a scratch area for playing with images.

An active registry using local disk storage may be an impediment, especially if hundreds of workers pull images from it. The registry software supports many storage types, mostly 'cloud' types with special high-availability and distributed fast-access storage configurations such as azure, gcs, s3, swift and oss. However for our tests we use the 'filesystem' storage type. I make use of a couple of large hardware-raid-based partitions and create a stripe zero mirror of them with LVM, yielding a fairly performant yet reliable local file-system implementation. This area is mounted under */data/registry/* where the registry images will reside.

5.4 Installation

Install the RPM:

```
# yum –enablerepo=rh-extras,sl-fastbugs install \
   docker-distribution,atomic,docker-common,docker
```

Though docker is not needed for a local registry I install it to be able to import or create, sign and push images on the same node in this test environment. Clearly it will be better in the future to create new images in another docker environment, and upload the images to the docker-registry.

Before enabling and starting the daemon I modify the main registry configuration file, config.yml, to suit our environment.

5.4.1 /etc/docker-distribution/registry/config.yml

Here is our configuration. The 'rootdirectory' points into the LVM raid-0 mirror. The logging level is set to info with text-mode formatting, but can be toggled to debug, and our certificate paths are added under the 'tls' area.

For now we comment out the basic-realm http-style authentication with user and path to the password file; we can enable it in the future if we decide to require authentication to the registry.

```
---
## to avoid ipv6 here we use the ipv4 address as the http URL
version: 0.1
log:
#  level: debug
  level: info
#  formatter: logstash
  formatter: text
  accesslog:
    disabled: false
#  fields:
#    service: registry
storage:
    delete:
        enabled: true
    cache:
        layerinfo: inmemory
    filesystem:
        rootdirectory: /data/registry
        maxthreads: 100
http:
    addr: 206.12.1.176:443
    tls:
        certificate: /etc/grid-security/hostcert.pem
        key: /etc/grid-security/hostkey.pem
#auth:
#    htpasswd:
#        realm: basic-realm
#        path: /etc/docker-distribution/registry/registry_passwd
```

5.4.2 Generate the file: htpasswd

In case you want to require a user name and password to access the site then to use basic authentication you must generate a password hash file. There is a useful recipe on using htpasswd in a registry on the IBM web site[1]. To use the '-B' option to htpasswd at least a RHEL-7 based version of httpd-tools is needed:

1 https://www.ibm.com/developerworks/library/l-docker-private-reg/

```
# cd /etc/docker-distribution/registry
# htpasswd -Bc ./registry_passwd t1user
New password:
Re-type new password:
Adding password for user t1user
# chmod 400 registry_passwd
```

If 'auth' is enabled in the registry's config.yml file, then we can test it from a remote Docker client:

```
# docker login https://docker-registry.lcg.triumf.ca
Username: t1user
Password:
Email:
WARNING: login credentials saved in /root/.docker/config.json
Login Succeeded
## and we can see in the above output what file we would need to replicate
## on clients if we used authentication on docker worker nodes
# cd /root/.docker/
# ls -la
total 12
drwx------  2 root root 4096 Mar 24 18:13 .
dr-xr-x---. 7 root root 4096 Mar 24 18:13 ..
-rw-------  1 root root  116 Mar 24 18:13 config.json
# cat config.json
{
        "auths": {
                "https://docker-registry.lcg.triumf.ca": {
                        "auth": "__obfuscated__",
                        "email": ""
                }
        }
}
```

5.5 Starting the registry

Now the Docker Distribution registry can be enabled and started:

```
# systemctl enable docker-distribution
# systemctl start docker-distribution
```

5.6 Querying the registry

As already mentioned, a search implementation is not yet done for Docker Registry V2. There are a few primitive curl commands that provide some useful information.

```
## get a list of images from this server
$ curl https://docker-registry.lcg.triumf.ca/v2/_catalog
{"repositories":["sl6-base","sl6.9_base","sl6.9_umd3","sl6.9_umd3_condor"]}

## get the list of images into a shell variable
$ images=$(curl -s https://docker-registry.lcg.triumf.ca/v2/_catalog| \
 cut  -d: -f 2|sed -r -e 's%[]["}]+%%g' -e 's%,% %g')

$ echo $images
sl6-base sl6.9_base sl6.9_umd3 sl6.9_umd3_condor

## get any tags associated with each image (always 'latest' if it was not
## otherwise specified during tagging)
$ for i in $images ; do
  curl -X GET https://docker-registry.lcg.triumf.ca/v2/$i/tags/list
done
{"name":"sl6-base","tags":["latest"]}
{"name":"sl6.9_base","tags":["latest"]}
{"name":"sl6.9_umd3","tags":["latest"]}
{"name":"sl6.9_umd3_condor","tags":["latest"]}

## a lot of information is in the manifest for each image
$ for i in $images ; do
  echo $i
  curl -s -H "Accept:*" \
  https://docker-registry.lcg.triumf.ca/v2/$i/manifests/latest
  echo ""
done
...
sl6.9_base
{
   "schemaVersion": 1,
   "name": "sl6.9_base",
   "tag": "latest",
   "architecture": "amd64",
   "fsLayers": [
      {
         "blobSum": "sha256:a3e..."
      },
      ...
   ],
   "history": [
      {
         "v1Compatibility": "{\"architecture\":\"amd64\",\"author\"...}"
      },
      {
         ...
      },
   ],
   "signatures": [
      {
         "header": {
            "jwk": {
               "crv": "P-256",
               "kid": "HDCN:...
            },
            "alg": "ES256"
         },
         "signature": "rh3...",
         "protected": "eyJm..."
      }
   ]
}
```

6 Image signing

This is a work in progress. Earlier there was a blocking bug with a *get_manifest* with Red Hat's atomic implementation that was recently fixed. In the image building chapter I show examples of image signing.

6.1 Using atomic

Recent versions of atomic now allow image signing in private registries.

It is important to note that image signatures are installed separately in the registry tree, and are not actually part of the image itself. This is different than signed RPMs for example, where the image signature is embedded in the RPM package.

We create a user account which owns the signing key, and modify the configuration files which govern image signing in the following sections.

6.1.1 GPG key pair for signing

First you need a GPG key pair. This is well-documented already in the Red Hat guide[1], but here are some quick steps as a reminder:

```
# yum install rng-tools
# rngd -r /dev/urandom
## now as a regular user
$ unset DISPLAY
$ GPG_TTY=$(tty)  gpg2 --gen-key
Please select what kind of key you want:
   (1) RSA and RSA (default)
   (2) DSA and Elgamal
   (3) DSA (sign only)
   (4) RSA (sign only)
Your selection? 4
What keysize do you want? (2048)
Requested keysize is 2048 bits
Please specify how long the key should be valid.
         0 = key does not expire
      <n>  = key expires in n days
      <n>w = key expires in n weeks
      <n>m = key expires in n months
      <n>y = key expires in n years
Key is valid for? (0)
Key does not expire at all
Is this correct? (y/N) y

GnuPG needs to construct a user ID to identify your key.

Real name: TRIUMF Tier-1
```

1 https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/

```
Email address: docker@lcg.triumf.ca
Comment: Image Signing Key
You selected this USER-ID:
    "TRIUMF Tier-1 (Image Signing Key) <docker@lcg.triumf.ca>"

Change (N)ame, (C)omment, (E)mail or (O)kay/(Q)uit? O
You need a Passphrase to protect your secret key.

                ┌──────────────────────────────────────────────────────┐
                │ Please re-enter this passphrase                      │
                │                                                      │
                │ Passphrase **************************_______________ │
                │                                                      │
                │       <OK>                              <Cancel>     │
                └──────────────────────────────────────────────────────┘

We need to generate a lot of random bytes. It is
 …
gpg: /home/unpriv/.gnupg/trustdb.gpg: trustdb created
gpg: key 31628D81 marked as ultimately trusted
public and secret key created and signed.

gpg: checking the trustdb
gpg: 3 marginal(s) needed, 1 complete(s) needed, PGP trust model
gpg: depth: 0  valid:   1  signed:   0  trust: 0-, 0q, 0n, 0m, 0f, 1u
pub   2048R/31628D81 2017-03-17
      Key fingerprint = 60B3 69DA F7D5 6B05 790F  6CF6 A87C C7F3 3162 8D81
uid                  TRIUMF Tier-1 (Image Signing Key) <docker@lcg.triumf.ca>
```

It is also important to export an ASCII version of the public key to be used by docker worker node clients:

```
$ gpg2 --armor --export --output TRIUMF_Tier-1_PublicSig.gpg docker@lcg.triumf.ca
```

6.1.2 User account for signing

Create a user account and set the login shell to /sbin/nologin – we unimaginatively call the account *docker*. In our case we set the home to /var/docker, and mark this directory to be backed up in our archival system. We copy the GPG private key (secring.gpg with mode 0400) and public key (pubring.gpg) into ~docker/.gnupg/.

6.1.3 Configuration files on the server

6.1.3.1 /etc/atomic.conf

We add entries for default_signer, matching the email address declared in the signature, and gnupg_homedir in this file.

default_signer: docker@lcg.triumf.ca
gnupg_homedir: /var/docker/.gnupg

6.1.3.2 /etc/containers/registries.d/default.yaml

The fully qualified hostname of this registry server needs to be used in the path to the signature store; the default path is */var/lib/atomic/sigstore*. Thus:

```
default-docker:
#  sigstore: file:///var/lib/atomic/sigstore
  sigstore: file:///var/lib/atomic/docker-registry.lcg.triumf.ca
```

6.1.4 Configuration files on the client

On the client to enforce and manage a policy of only using digitally signed images there are two files that need to be configured.

6.1.4.1 /etc/containers/policy.json

We set signing policy in this json file[1]. In our case we reject by default all images, and then selectively accept signed content from our own registry server. (I need to review why one might want to leave the docker-daemon transport with 'insecureAcceptAnything').

```
{
    "default": [
        {
            "type": "reject"
        }
    ],
    "transports": {
        "docker": {
            "docker-registry.lcg.triumf.ca": [
                {
                    "keyType": "GPGKeys",
                    "type": "signedBy",
                    "keyPath": "/etc/pki/containers/TRIUMF_Tier-1_PublicSig.gpg"
                }
            ]
        },
        "docker-daemon": {
            "": [
                {
                    "type": "insecureAcceptAnything"
                }
            ]
        }
    }
}
```

1 https://github.com/containers/image/blob/master/docs/policy.json.md

With this policy in place we cannot pull an unsigned image:

```
# docker pull sl6-base
Using default tag: latest
Trying to pull repository docker-registry.lcg.triumf.ca/sl6-base ...
docker-registry.lcg.triumf.ca/sl6-base:latest isn't allowed: A signature was required, but no
signature exists
```

Note that the file lists the key path for the public key, that is:

/etc/pki/containers/TRIUMF_Tier-1_PublicSig.gpg

This file needs to be on the clients – recall that it was created when the GPG key pair was created.

6.1.4.2 /etc/containers/registries.d/Your_Server_Name.yaml

For each registry using signed images, we need to configure the URL for this server on each client which looks for signatures:

```
# cat /etc/containers/registries.d/docker-registry.lcg.triumf.ca.yaml

docker:
  docker-registry.lcg.triumf.ca:
    sigstore: http://docker-registry.lcg.triumf.ca/signatures/
```

6.2 Using notary

This is yet to be tested. In the case of Docker 'notary', there is the annoying issue of the lack of RPM packaging for these Docker tools. I am easily able to build notary client and server from source, so I can generate RPMs myself. If Docker Compose is really needed then I need to figure out how to generate RPMs for them.

7 Image building

7.1 A full image from a shell script

This was the initial method we used to create images – this is now deprecated, but we leave it here as a reference.

I used another test node already installed with SL6 to build some SL images for containers. A test build system should be considered expendable, since mistakes in building test images might overwrite files or render it inoperable – do not use an important production system for this purpose!

The Docker 'contrib' area on Github has a number of example scripts that can be used to build images, depending on your underlying build tools. I modified the 'mkimage-yum.sh[1]' script. Our modified script is available at the URL[2] in the footnote. It essentially creates an image in a few yum stanzas into a target directory using yum. A few of the options for yum are important, as they allow us to install into an alternate root, selecting only mandatory packages without accompanying documentation; they are:

- —releasever
- —installroot
- --setopt=tsflags=nodocs
- --setopt=group_package_types=mandatory

The script writes into a target directory, which is tarred and gzipped at the end. It is this tarball which is copied to a working directory on the registry node, where it is later imported into the registry.

The script first sets up the bare minimum number of device files in the target area. Note that the Tier-1 release RPMs use a yum variable that determines if the mirror is accessed by NFS or by HTTP. For a docker image we want to avoid NFS since it introduced complexity in the container setup, so the variable gets set to an HTTP URL at the end of the script.

7.1.1 Yum stanzas

These are descriptions of the yum stanzas invoked in the script. By breaking it down we can inspect disk space used so far at each step.

1. Because we wish to use the local mirror for our installation I first install the needed Tier-1 release RPMs in the first 'yum install' stanza. This of course drags in a minimal base system.
   a) Size: 333 MB
2. Then the script does a yum group-install of core and development.
   a) Size: 517 MB
3. The next stanza installs a list of packages that we always install on workers in the Tier-1. (This could be skipped I think, relying instead on the HEP_Oslibs_SL6 RPM to pull in dependencies)
   a) Size: 1.1 GB
4. Then we install the UMD-3 middleware.
   a) Size: 1.7 GB

1 https://github.com/docker/docker/blob/master/contrib/mkimage-yum.sh
2 http://grid.triumf.ca/share/

5. Finally we install HEP_Oslibs_SL6 and condor
   a) Size: 1.8 GB

7.1.2 Final image setup

After the software is installed the script expunges any remaining space-gobbling features like man pages, language files, icons, etc. and then the size falls to **1.5 GB**.

At the end of the script we add local customization – the group and user configuration is appended to passwd, shadow, group, gshadow in the target etc directory; the Tier-1 profile is added as well as the local condor configuration. A tarball is created, and the final size of this file is **506 MB**.

To create a truly portable image for use outside of the Tier-1, we could use release RPMs from global open HTTP mirrors instead.

The final 'local' configuration could also be done on the container at launch time, or could be applied to a separate layered image, so that the generic image would be untainted.

As part of the image creation we have installed a locally-built RPM named dumb-init. The ***dumb-init***[1] application is useful in container environments because it allows better control over launch-time commands. See ahead in the Container section on how it is used.

7.2 Image layers built from Kickstart and dockerfiles

Currently we are building images in a manner similar to the way that Red Hat builds images, judging from their image collection at *registry.access.redhat.com*.

We start with a kickstart-ed image created using lorax[2], and then use a sequence of dockerfiles to create subsequent image layers. On the atlas-canada wiki I will provide the SL6 lorax RPMs, docker files and kickstart file used to create these images.

7.2.1 Kickstart-ed image

We use *lorax*, software that provides livemedia-creator, a tool which uses Anaconda and Kickstart to create images for use with virtualization. It uses a minimal kickstart file and emits a compressed tar file that we can import into docker.

Red Hat provides this software as an RPM starting at RHEL7. In principle lorax on SL7 should be able to build RHEL6 anaconda images, but I had trouble getting it to work – this is an issue to

1 https://github.com/Yelp/dumb-init
2 https://github.com/rhinstaller/lorax

revisit. This RPM is not available on RHEL6, so I created an RPM that works on my SL6-based KVM servers. Only a few modifications were needed to the RPM.

The 'livemedia-creator' utility needs the path to the kickstart file and the path to the distribution boot ISO image. If TMPDIR space is limited then you can tell it to use a different scratch area.

```
# livemedia-creator --make-tar --iso=../scratch/sl-boot.iso –ks=sl6.ks \
  --image-name=sl6-base.tar.xz  --tmp /data/scratch/ \
  --kernel-args="console=tty0 console=ttyS0,115200"
# ls -lah
total 590M
drwxr-xr-x 2 root root 4.0K Aug  1 10:30 .
drwxr-xr-x 5 root root 4.0K Jul 31 09:16 ..
-rw-r--r-- 1 root root  82M Jul 31 18:40 centos6.tar.xz
-rw-r--r-- 1 root root 230M Mar 28  2017 cos-boot.iso
-rw-r--r-- 1 root root  988 Jul 31 15:47 livemedia.log
-rw-r--r-- 1 root root    0 Jul 31 15:47 program.log
-rw-r--r-- 1 root root 232M Apr 11  2017 sl-boot.iso
-rw-r--r-- 1 root root  47M Aug  1 10:30 sl6-base.tar.xz
```

We then import this image into the registry server:

```
# docker import sl6-base.tar.xz sl6-base
sha256:bbf4d72da5d3e038eb3b861fcb96747071168bb6a8f677e4f1689e91f15aa8dd
```

We need to tag this image so that we can refer to it in our first dockerfile, and then push it into the registry. The image size 202 MB.

```
# docker tag sl6-base docker-registry.lcg.triumf.ca/sl6-base
# docker -D push docker-registry.lcg.triumf.ca/sl6-base
…
ddb896626840: Pushed
latest: digest:
sha256:63e4cf57e7f1749f8d78f1fd0cb534a3411d1ea980bb92c4a77637a39a293237 size: 528
```

7.2.2 Base image with annotation

Now we build a new base image using our first dockerfile. The image size will not change, since we are in effect only adding some meta-data -- annotations (labels) -- to the image.

```
# docker build -t sl6.9_base -f Dockerfile.base .
Sending build context to Docker daemon 3.072 kB
Step 1 : FROM sl6-base
 ---> bbf4d72da5d3
Step 2 : MAINTAINER docker@lcg.triumf.ca
 ---> Running in c245bd23a9cd
 ---> c72d7d8c5adf
Removing intermediate container c245bd23a9cd
Step 3 : ENV container docker
 ---> Running in 86c80e8a1dcf
 ---> 0da9eb15ffaf
```

```
Removing intermediate container 86c80e8a1dcf
Step 4 : ENV PATH /bin:/usr/bin:/sbin:/usr/sbin
 ---> Running in aa8a9fa27f25
 ---> 01aab0a8a428
Removing intermediate container aa8a9fa27f25
Step 5 : CMD /bin/bash
 ---> Running in 2e4df6919251
 ---> 70918bffea21
Removing intermediate container 2e4df6919251
Step 6 : LABEL ca.triumf.lcg.vendor "TRIUMF Tier-1" maintainer
"docker@lcg.triumf.ca" distribution "Scientific Linux 6" name "sl6" version "6.9"
release "01" description "SL 6 base image"
 ---> Running in 8d9604b30f2d
 ---> e2658b084461
Removing intermediate container 8d9604b30f2d
Successfully built e2658b084461
```

Again we tag this image and push it to the registry; we also sign this one because it is the first of the images that we may want to use for other purposes.

```
# docker tag sl6.9_base docker-registry.lcg.triumf.ca/sl6.9_base
# docker push docker-registry.lcg.triumf.ca/sl6.9_base
# atomic sign docker-registry.lcg.triumf.ca/sl6.9_base
Created: /var/lib/atomic/docker-
registry.lcg.triumf.ca/sl6.9_base@sha256=9eb88232100e32914e948c72311bdb90bb4526cf135c1d3833c76
c4fa14f0cdf/signature-1
```

7.2.3 Middleware image with annotation

Then we build a middleware image using our docker file, Dockerfile.umd3. The image is based on the sl6.9_base image, and again we tag, push and sign the resulting image, which is 1.37 GB.

```
# docker build -t sl6.9_umd3 -f Dockerfile.umd3 .
Sending build context to Docker daemon 33.79 kB
Step 1 : FROM sl6.9_base
 ---> e2658b084461
Step 2 : ENV PATH /bin:/usr/bin:/sbin:/usr/sbin
 ---> Running in 323d88e1fd57
 ---> 06d16cc2bd81
Removing intermediate container 323d88e1fd57
Step 3 : LABEL ca.triumf.lcg.vendor "TRIUMF Tier-1" maintainer
"docker@lcg.triumf.ca" distribution "Scientific Linux 6" name "sl6-umd3" version
"3" release "01" description "SL 6 image with Middleware for ATLAS"
 ---> Running in a107627a8f6a
 ---> e0f7e80352c7
…....
Removing intermediate container 270b7640ae4e
Successfully built c2bec29c131b
# docker tag sl6.9_umd3 docker-registry.lcg.triumf.ca/sl6.9_umd3
# docker -D push docker-registry.lcg.triumf.ca/sl6.9_umd3
# atomic sign docker-registry.lcg.triumf.ca/sl6.9_umd3
```

There is some work to do on this docker file – for example, I forgot to run a yum cleanup to release some disk space. As well, experience shows that two packages are missing which are needed in ATLAS environments – wget and unzip – and we could install them here, and also take the opportunity to update any packages in the SL base image. A cleaned-up Dockerfile.umd3 might look like this:

```
# Dockerfile containing EGI/UMD3 middleware
FROM sl6.9_base

ENV PATH /bin:/usr/bin:/sbin:/usr/sbin

LABEL   ca.triumf.lcg.vendor="TRIUMF Tier-1" \
        maintainer="docker@lcg.triumf.ca" \
        distribution="Scientific Linux 6" \
        name="sl6-umd3" \
        version="3" \
        release="02" \
        description="SL 6 image with Middleware for ATLAS"

COPY releases/*.rpm /root/

## each RUN command below should all be on one line
RUN yum  --setopt=tsflags=nodocs --setopt=group_package_types=mandatory -y
localinstall /root/*.noarch.rpm && rm /root/*.rpm

RUN yum --disablerepo=EGI-trustanchors --setopt=tsflags=nodocs
--setopt=group_package_types=mandatory -y install emi-wn xrootd-client glexec-wn
yaim-glexec-wn HEP_OSlibs_SL6 wget unzip

RUN yum -y update && yum clean all

RUN rpm -e --nodeps dracut-kernel dracut grubby kernel-firmware kernel

CMD ["/bin/bash"]
```

7.2.4 TRIUMF-specific htcondor image with annotation

Finally we build the Tier-1 specific htcondor image, tag, push and sign it.

```
 docker build  -t sl6.9_umd3_condor -f Dockerfile.condor .
Sending build context to Docker daemon 304.1 kB
Step 1 : FROM sl6.9_umd3
 ---> c2bec29c131b
Step 2 : ENV PATH /bin:/usr/bin:/sbin:/usr/sbin
 ---> Running in a521322fdf98
 ---> 974513072436
Removing intermediate container a521322fdf98
Step 3 : LABEL ca.triumf.lcg.vendor "TRIUMF Tier-1" maintainer
"docker@lcg.triumf.ca" distribution "Scientific Linux 6" name "sl6-umd3-condor"
version "1" release "01" description "SL 6 image with Middleware and htcondor for
TRIUMF Tier-1"
 ---> Running in 508c658e1921
…....
Removing intermediate container 155fdf60b185
Successfully built 87060e85969c
```

```
# docker tag sl6.9_umd3_condor docker-registry.lcg.triumf.ca/sl6.9_umd3_condor
# docker -D push docker-registry.lcg.triumf.ca/sl6.9_umd3_condor
# atomic sign docker-registry.lcg.triumf.ca/sl6.9_umd3_condor
```

This final image is 1.5 GB, and is composed of 10 layers. The history of the layers is best inspected on the host where the image was created. Presumably only those operations that contributed size to the image result in a layer, as you can see from this image's history:

```
# docker inspect sl6.9_umd3_condor
...
"Type": "layers",
"Layers": [
"sha256:ddb896626840c2b28329368596e2db2a949cbbccb23369a5b60180fb74191550",
"sha256:4b06468ea88a80c8a3d123171263a9caed23e3b77592cab85655fcecfd0e853f",
"sha256:008a3aba4434f8548f81f713cf10d5ff642e19e93d4bc531a64f55fdc7c32d7b",
"sha256:bca0d3946213fd13378726623f81460577e82cec6e9a9a40e666d7bbaaeb0faa",
"sha256:e070d8b0898dc9d7de33edb88122468678910998ac1cb31bc81db82fe1c5472c",
"sha256:195bec4885ce7d6ab13db344b4e8ebb0750886376e85d000ba2f6b3f9b4a1027",
"sha256:259786139dfb069a79d1b8233b19c10f8f246d6a216f4ee74d8ec2f5d6d61450",
"sha256:f268c94fab6c70b90068d29f1bde2c536fff9d471b6fce4c702b29124e127ba4",
"sha256:cea7fde541a301c6734bfb7b291ce0886929d6bed52c32b37ad81c86f26e585b",
"sha256:b0ea83cfb5a576f1cfdbc1f7c808408a9f095746b5058fb722e092eb60add776"
]

# docker history sl6.9_umd3_condor
IMAGE               CREATED BY                                      SIZE
e4113613018a        /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
e2e4ae6ca277        /bin/sh -c yum  --disablerepo=* --enablerepo=   80.59 MB
a984db7d7b77        /bin/sh -c echo "http://gridadm.triumf.ca/mir   32 B
b3798cdc226c        /bin/sh -c groupadd -g 64 condor &&  useradd    152 kB
491c60fb0f99        /bin/sh -c yum  --setopt=tsflags=nodocs --set   47.39 MB
bb6484da5bea        /bin/sh -c #(nop) COPY multi:439ebcd9df18f261   297.8 kB
3def4c0626cd        /bin/sh -c #(nop)  LABEL ca.triumf.lcg.vendor   0 B
54ba2302f57b        /bin/sh -c #(nop)  ENV PATH=/bin:/usr/bin:/sb   0 B
c2bec29c131b        /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
a3b54475445f        /bin/sh -c rpm -e --nodeps dracut-kernel drac   45.06 MB
8ebf74e14de5        /bin/sh -c yum  --enablerepo=UMD-3-base,UMD-3   1.062 GB
865876db73a4        /bin/sh -c yum  --setopt=tsflags=nodocs --set   63.15 MB
3797695661c4        /bin/sh -c #(nop) COPY multi:0aa03c6c272f8d08   28.57 kB
e0f7e80352c7        /bin/sh -c #(nop)  LABEL ca.triumf.lcg.vendor   0 B
06d16cc2bd81        /bin/sh -c #(nop)  ENV PATH=/bin:/usr/bin:/sb   0 B
e2658b084461        /bin/sh -c #(nop)  LABEL ca.triumf.lcg.vendor   0 B
70918bffea21        /bin/sh -c #(nop)  CMD ["/bin/bash"]            0 B
01aab0a8a428        /bin/sh -c #(nop)  ENV PATH=/bin:/usr/bin:/sb   0 B
0da9eb15ffaf        /bin/sh -c #(nop)  ENV container=docker         0 B
c72d7d8c5adf        /bin/sh -c #(nop)  MAINTAINER docker@lcg.triu   0 B
bbf4d72da5d3                                                        202.2 MB
```

8 Docker Containers

When you launch a Docker container, you need to pull an image if it is not yet local. The docker command knows where to find images by virtue of the configuration as previously shown. The image will be pulled automatically – here is an example from an last year with an older test registry:

```
# docker run -d -it --cap-add SYS_ADMIN -v /home:/home \
 /etc/grid-security:/etc/grid-security -v /cvmfs:/cvmfs \
 /opt/glite:/opt/glite -p 4080:4080 sl6-umd3condv4  /root/init


Unable to find image 'sl6-umd3condv4:latest' locally
Trying to pull repository pps04.lcg.triumf.ca:5000/sl6-umd3condv4 ...
Pulling repository pps04.lcg.triumf.ca:5000/sl6-umd3condv4

762758303a84: Extracting [===========>                          ] 72.42 MB/312.8 MB
. . .
Status: Downloaded newer image for pps04.lcg.triumf.ca:5000/sl6-umd3condv4:latest
pps04.lcg.triumf.ca:5000/sl6-umd3condv4: this image was pulled from a legacy
registry.  Important: This registry version will not be supported in future versions
of docker.
```

Note the warning message above, concerning the registry version (V1). Indeed, at Ansible version 1.9 any Ansible docker command failed with this warning. At least at Ansible 1.10 a better supported 'docker_container' command succeeds and issues the warning.

8.1 Running Docker containers

8.1.1 Bind mounts

A very useful feature of docker containers is the ability to bind-mount volumes. We make liberal use of this in early docker worker node containers to preserve disk space and to reuse components. The list of bind-mounts used in the test bed:

- /etc/grid-security
  - → let the Engine host manage the CRLs
- /etc/localtime
  - → this no longer works, instead you must pass timezone information as an environment variable in current docker implementations
- /opt/glite
  - → probably will not be needed, but this directory on the Engine host contains yaim configuration settings that we could access if needed
- /cvmfs
  - → we want to avoid nfs mounts in the container. Instead we install cvmfs on the Engine host, and bind-mount it on the container.
  - → note that we configure the Engine host to never unmount the cvmfs trees:

```
[root@wn024 ~]# tail -1 /etc/auto.master
/cvmfs   /etc/auto.cvmfs --timeout 0
```

→ To automatially trigger mounting of cvmfs mount points before a container is started we use a systemd service; see the appendix for details.

- /home
  - → this is the traditional scratch space for user jobs in the Tier-1. Therefore the Engine host had a large home directory with user and group ID numbers that must match the uid/gid setup on the docker container.
  - → The Engine host can be responsible for cleanup of scratch space and home directories via its own cron jobs. At the Tier-1, the disk-checking cron jobs would also continue to run from the Engine host – however, assuming a condor configuration the way of stopping the container's condor_master in case of imminent disk failure would need to be revisited.

8.1.2 Port Mappings

To simplify the configuration we start condor using the shared-port option:

```
# diff condor_config.local condor_config.local.t1-ppsce
150,154d149
<
< ## Tier-1 docker test settings
< DISCARD_SESSION_KEYRING_ON_STARTUP = False
< USE_SHARED_PORT = True
< SHARED_PORT_ARGS = -p 4080
```

We can then map the internal port to any unused port on the Engine host with the command-line option '-p'; e.g.

-p 4080:4080

or

-p 4088:4080

8.1.3 Hostnames and IP addresses

Docker networking is not like virtualization networking – unless you create user-defined subnets then you cannot control the IP address of containers at launch – Docker Engine will sequentially assign IP addresses from its IP base. However to set the local hostname of a container to the DNS-assigned hostname one can use the trick of looking up the hostname from its IP address in the launch script and assigning it. However, the SYS_ADMIN capability is needed to allow

the container to change its own hostname. This is less secure – we need to look more carefully at this issue – see the link in the footnote[1].

```
# cat /root/init
#!/usr/bin/dumb-init /bin/bash
## get the ip address and hostname - set the hostname before starting condor
thisip=$(ip -4 route get 1 | awk '{print $NF;exit}')
if [ "$thisip" != "" ] ; then
  thishost=$(host $thisip) 2>/dev/null
  if [ "$thishost" != "" ] ; then
    h=$(echo "${thishost##* }")
    h=$(echo "${h%?}")
    hostname $h
  fi
fi

if [ -x /home/docker/bin/docker-worker-container-setup.sh ] ; then
  if [ ! -f /etc/condor/docker-worker-container-setup.done ] ; then
    /home/docker/bin/docker-worker-container-setup.sh
  fi
fi

## set up condor environment before running condor_master in the foreground
. /etc/sysconfig/condor
/usr/sbin/condor_master -f
```

8.2 Docker and Ansible

The command-line is a bit unwieldy when launching containers. This is a use-case for tools like Ansible. We have an example docker role named 'docker-worker' which sets up a docker environment on an SL7 worker node. Then we can use an Ansible playbook to launch containers; here is an example (*note: we currently have an ansible bug with container exit on creation using docker_container – to be understood, but will probably need a local update to Ansible version >= 2.3*):

```
## example command-line invocation without ansible
# docker run -d -it --name vns0005.triumf.lcg --hostname vns0005.triumf.lcg \
 -v /home:/home -v /etc/grid-security:/etc/grid-security -v /cvmfs:/cvmfs \
 -v /opt/glite:/opt/glite -v /opt/Atlas:/opt/Atlas \
 --cap-add SYS_ADMIN -p 4080:4080 \
 -e "LANG=en_US.UTF-8" -e "TZ=America/Vancouver" \
 sl6.9_umd3_condor /root/init


## example with ansible
# ansible-playbook docker/condor-container.yml \
  -e "target='wn024' image=sl6.9_umd3_condor"
```

1 https://docs.docker.com/engine/reference/run/#/runtime-privilege-and-linux-capabilities

```
# cat playbooks/docker/condor-container.yml
---
# Usage:
#  ansible-playbook THIS_FILE -e "image='sl6' target='some_engine_host'"
#  the internal and external port assignments can be given default values or
#  can be overridden.  We should make the command into a variable as well.

- name: Launch docker worker condor-based containers
  hosts: "{{ target }}"
  gather_facts: false
  user: root  vars:
    - cmd: '/root/init'
    - port: 4080
    - external_port: "{{ port }}"
    - thisname: "{{ ansible_date_time.epoch }}"

  tasks:
  - name: Only run if this is a docker engine server
    fail: msg="This is not a docker engine"
    when: not is_docker_engine

  - name: Launch a condor-enabled htcondor container
    docker_container:
      name: "{{ thisname }}"
      image: "{{ image }}"
      exposed_ports: "{{ port }}"
      published_ports: "{{ external_port }}:{{ port }}"
      #command: "/root/init"
      entrypoint: "{{ cmd }}"
      capabilities: SYS_ADMIN
      env:
        LANG: en_US.UTF-8
        TZ: America/Vancouver
      volumes:
          - /etc/grid-security
          - /cvmfs
          - /home
          - /opt/glite
          - /opt/Atlas ## a few local config files

  - set_fact: oldname="{{ ansible_docker_container.Config.Hostname }}"
  -  set_fact: thishost="{{ lookup('dig', '{{ansible_docker_container.NetworkSettings.IPAddress }}/PTR') }}"

  - name: Rename container
    command: docker rename {{ oldname }} {{ thishost|regex_replace('.$', '') }}
```

Here is an Ansible playbook which runs a command on demand; the container's hostname which is associated with this worker node is in the Ansible inventory as 'my_container':

```
## run any command inside the container and view the output
# cat docker-command.yml
---
# Usage:
#  ansible-playbook THIS_FILE -e "target='wns0001' cmd='/bin/date'"
#

- name: Send command to a docker container
  hosts: "{{ target }}"
  gather_facts: false
  user: root
  vars:
```

```
tasks:
- name: Only run if this is a docker engine
  fail: msg="This is not a docker engine"
  when: not is_docker_engine

- name: Run a command inside a container
  command: docker exec -i '{{ my_container }}' {{ cmd }}
  register: dout

- debug: var=dout.stdout_lines
```

9 Next Steps

So far we have only used containers to run a condor-based worker node environment at the Tier-1. Aside from getting more experience with running containers for ATLAS jobs, we also need look at maintenance issues and logging. Finally, it would be interesting to also look at Docker Registry configured as a cache – we have not done this yet.

9.1 Logging

We will want to capture logs from containers in a production environment. There are a number of logging options[1] that need to be investigated. Another option is to create container-named sub-directories in /var/log/ on the Engine host and bind-mount those volumes to the container – for example – capturing the container's internal logs from /var/log/condor under a sub-directory on the Docker Engine host at /var/log/SOME_NAME_timestamp/ or elsewhere as needed. This is easily done with Ansible at container launch time.

9.2 Image Maintenance and Tracking

We will need a plan on how images are maintained and tracked:

- When and where do we patch the images? Should they be patched at the registry hub, or would you update them inside the container before launching the application?
- How will we track changes? Should images, or just their scripts, and/or their signature hashes be tracked in a version control system?
- What monitoring tools do we need?

1 https://docs.docker.com/engine/admin/logging/overview/

10 Glossary

There are a number of technologies associated with Docker that are not mentioned in this document. Here I will provide in the near future a short reference of terms you may come across in your research, or even in this document, with a short definition of each. For now I just list some of the terms.

- Docker Compose

  Compose is a tool for defining and running *multi-container* Docker applications.

  Ref: https://github.com/docker/compose

  Ref: https://github.com/docker/docker.github.io/blob/master/compose/compose-file/index.md

- atomic
- cockpit
- etcd
- flannel
- koji
- runc
- skopeo
- Notary

10.1 Orchestrators

Launch and manage a cluster of containers.

- Docker Swarm

  Docker Swarm 'standalone' is a native clustering system for Docker; it typically turns a pool of Docker hosts into a *single, virtual host* using an API proxy system. It is the original orchestration project from Docker.

  It is now part of Docker Engine, starting at Docker version 1.12.

  Ref: https://docs.docker.com/engine/swarm/

- Apache Mesos

  Mesos manages computer clusters. It consists of a master daemon that manages agents running on each cluster node, and Mesos frameworks which run tasks on these agents. A

framework running on top of Mesos consists of two components: a scheduler that registers with the master to be offered resources, and an executor process that is launched on agent nodes to run the framework's tasks. Mesos has a Docker executor; thus Mesos can manage a cluster of such containers. Docker containers are natively supported by Mesos starting at version 0.20.0.

Ref: https://mesosphere.com/blog/2013/09/26/docker-on-mesos/

- Kubernetes

  Kubernetes was originally developed by Google to manage clusters of containerized applications; naturally it also supports Docker. Some other technologies are required to use Kubernetes – etcd, flannel – they are defined in the glossary.

  Ref: https://en.wikipedia.org/wiki/Kubernetes

Some interesting links:

https://technologyconversations.com/2015/11/04/docker-clustering-tools-compared-kubernetes-vs-docker-swarm/

11 Appendix

11.1 Configure systemd for auto cvmfs mounts

```
# cat /etc/systemd/system/trigger-cvmfs-mount.service
[Unit]
After=autofs.service

[Service]
ExecStart=/root/bin/trigger-cvmfs-mount.sh

[Install]
WantedBy=default.target

# cat /root/bin/trigger-cvmfs-mount.sh

#!/bin/sh

## list major cvmfs mount points on docker container worker nodes
/bin/ls /cvmfs/atlas.cern.ch/ /cvmfs/atlas-condb.cern.ch/ \
    /cvmfs/atlas-nightlies.cern.ch/ /cvmfs/cvmfs-config.cern.ch/ \
        /cvmfs/sft.cern.ch >/dev/null 2>&1

## Then just enable the service:
# systemctl enable trigger-cvmfs-mount
```