Creating a Docker Container for Use as a CernVM-FS Release Manager with a S3-Hosted Repository

Lillian Huang

August 11, 2017

1 Introduction

The CernVM File System (CVMFS) is a read-only filesystem that provides a low-maintenance, reliable software distribution service for high-energy physics experiments at CERN [1]. CVMFS deploys data analysis, reconstruction, and simulation software on the Worldwide LHC Computing Grid [2]. The file system focuses on software acquisition and use, as most clients using CVMFS only need to access the software, not change it. Thus, data is aggressively cached, de-duplicated across different software releases, and transported through HTTP proxy servers for the fastest possible retrieval of files. However, HTTP transfer is sometimes insecure, so in order to ensure integrity of the files while still keeping all data cacheable, all file catalogs are cryptographically signed with content hashes, which verify file integrity for all users [1]. CVMFS is actively used by many experiments at CERN, such as ATLAS and LHCb, and is integral to many collaborations' distributed computing infrastructure [3].

1.1 CernVM-FS: Overall Architecture

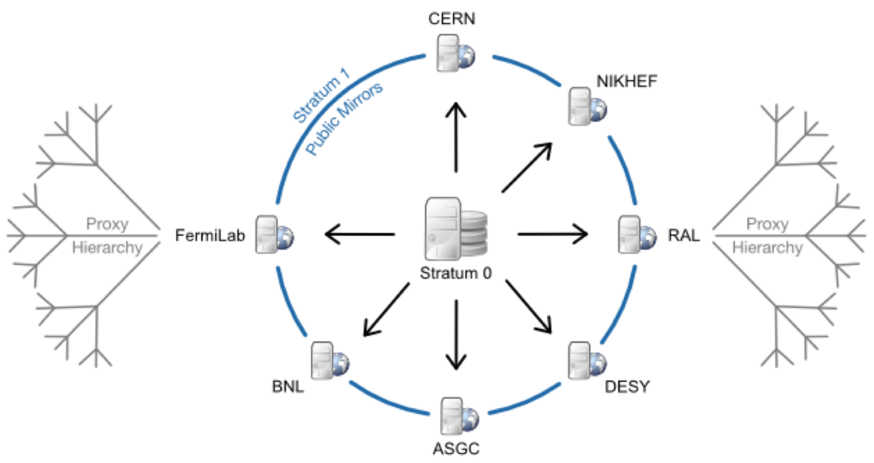

Figure 1: The architecture of CVMFS [4].

The architecture of CVMFS begins with the Stratum 0, a server located at CERN where new data is written, stored, and distributed on a number of repositories, where repositories are a form of content-addressable storage [5]. All repositories are replicated to around five Worldwide LHC Computing Grid Tier 1 sites, called Stratum 1's, which is where CVMFS clients connect in order to access the files. The Stratum 0 and Stratum 1's synchronize via a cron job periodically so that all the data remains up-to-date. There are web proxy servers at each computing center that cache traffic to the Stratum 1 services. If there are connection problems with one server, clients will be automatically redirected to another. Files are transferred and cached upon client demand [3].

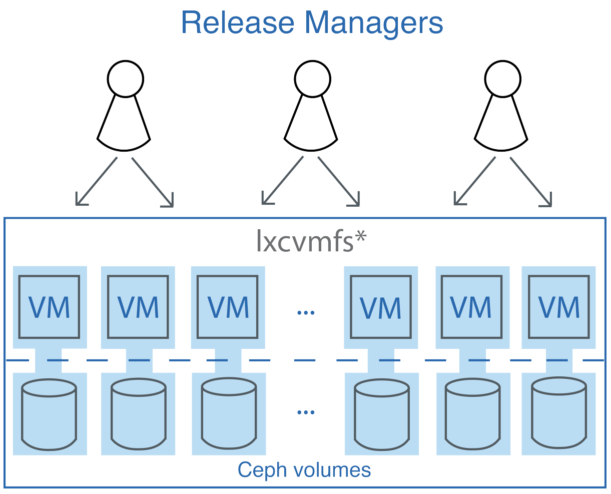

Figure 2: Current release management system within the CERN Stratum 0 Architecture.

1.2 Release Management

Although CVMFS is a read-only file system, it is necessary for some clients, known as release managers, to make changes to certain repositories in order to continue to distribute new releases of their software. This update procedure is commonly called a “transaction.” Currently, CVMFS release managers must connect to one of many special “release management” `lxcvmfs*` virtual machines dedicated to making changes to existing repositories stored in the Ceph [6] volumes. These are located at the Stratum 0, where each separate virtual machine manages one or a few repositories on a mounted Ceph volume. There are currently a total of 39 repositories and 25 virtual machines. Figure 2 shows the current architecture and workflow for release management at CERN.

1.3 Project Outline

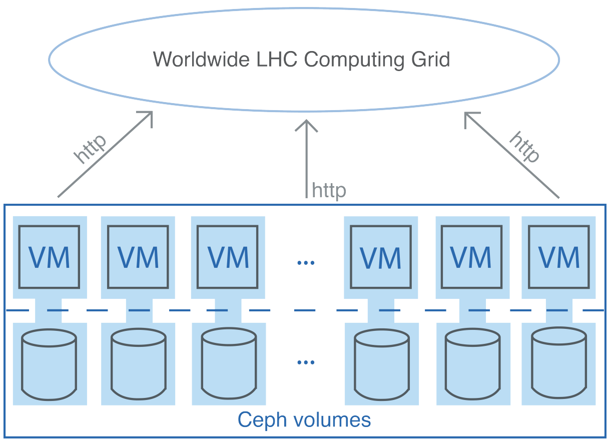

Figure 3: Architecture of the Distribution from Stratum 0 to Stratum 1 for CVMFS at CERN

There are a number of improvements to be made in the release management system.

- Figure 3 shows that according to the current architecture, release managers write to the same virtual machines and volumes that also transfer the data to the Stratum 1. This means that if the load for one of these tasks is especially heavy, performance will be slowed for the other.
- Every new repository requires a new virtual machine to manage it, which is inefficient.
- Many existing repositories do not get updated daily; some even go without changes for weeks. Consequently, some release management machines sit idly for long periods of time, taking up space and otherwise wasting resources.

- The repositories themselves are not easily independently scalable from the virtual machines, so if one repository becomes very big, there is no way to increase performance in terms of CPU power or faster storage.
- The Stratum 0 is a single point of failure in terms of storage. If the Stratum 0 crashes, there is no way to recover what was there. Much of the data will be cached, but problems will arise when the Stratum 1's try to pull from the Stratum 0.

The aim of this project was to rectify these issues by creating and running a release management Docker [7] image to replace the virtual machines, as well as having remote shared storage as the back-end to store the data. The Docker container will be started on demand by release managers by running the image, and after they have completed their transaction, they can exit the container and it will be gone from the host. Thus, clients can use the same container to update any and all repositories, instead of needing a new machine for each new repo. In addition, since the containers are launched only on demand and are removed after use, there is no idle waste of resources. The load balance problem between release management and serving to the Stratum 1 will also be rectified, as the load of release management would fall on containers only, and the performance of one task would not interfere with the other. Furthermore, the remote repository will be more scalable, as it is possible to make remote storage faster, and containers are very easily scalable, i.e. one can run more than one container to publish at a time which adds CPU power. Finally, the remote repository will not provide a single point of failure, thanks to its design. Thus the goal of this project is to create a working Docker image with release management capabilities, and to configure it to work with a remote repository.

2 Implementation

2.1 Creating the Remote Repository

A remotely hosted repository was created using S3 protocol on a currently-running release management virtual machine. First, an S3 bucket, called `cvmfs-stratum0-testng`, was created using Cyberduck on a remote server `cs3.cern.ch`. Then, using the `cvmfs_server mkfs` command, a new S3-hosted repository was created and configured to store data on `cs3.cern.ch` and in the `cvmfs-stratum0-testng` bucket. The path to the S3 configuration file necessary for writing to a cloud repository, as well as the server and bucket information, and of course the repository name, were passed as parameters to the command:

```
cvmfs_server mkfs -s /etc/cvmfs/cvmfs_stratum0_testng/mys3.conf \
     -w http://cs3.cern.ch/cvmfs-stratum0-testng testng.cern.ch
```

In general, this was relatively straightforward. For the configuration, it was needed to indicate the name of the bucket and the host url, as well as add a configuration file at `/etc/cvmfs/.../mys3.conf`. Documentation for this can be found at [8].

2.2 Creating the Docker Image

A Docker container image was developed to provide the functionality of a CVMFS release management machine; documentation for creating and running Docker images can be found at [9]. The release management container image is stored in a private GitLab registry at [10]. It was necessary to configure a number of aspects of current release managers on the new container, such as mounting scratch space volumes for repository changes, creating the required directories for a union mount file system to be able to write to the repository, and placing configuration files in their correct locations. This was ultimately implemented by properly creating the image stored in the private registry (by writing the Dockerfile) in combination with executing the correct `docker run` command.

The Dockerfile is shown below:

```
FROM cern/cc7-base

#this environment variable should be overwritten in run command
ENV REPO_NAME default_repo_name
```

```
#RUN yum -y install cvmfs cvmfs-server

RUN curl -O http://ecsft.cern.ch/dist/cvmfs/nightlies/cvmfs-git-485/\
cvmfs-server-2.4.0-0.485.5a5c8a4c2ef5b745git.el7.centos.x86_64.rpm ; \
curl -O http://ecsft.cern.ch/dist/cvmfs/nightlies/cvmfs-git-485/\
cvmfs-2.4.0-0.485.5a5c8a4c2ef5b745git.el7.centos.x86_64.rpm ; \
yum -y install cvmfs-server-2.4.0-0.485.5a5c8a4c2ef5b745git.el7.centos.x86_64.rpm \
cvmfs-2.4.0-0.485.5a5c8a4c2ef5b745git.el7.centos.x86_64.rpm ;

RUN echo '[ ! -z "$TERM" -a -r /etc/motd ] && cat /etc/motd' \
>> /etc/bashrc \
; printf "\
* *****************************************************************\n\
* CVMFS Dockerized Release Manager!                                \n\
* This is a test message of the day because IDK what goes here yet    \n\
* #################################################################\n\
\n\
To start a transaction:\n\
   cvmfs_server transaction testng.cern.ch\n\
To publish a transaction:\n\
   cvmfs_server publish testng.cern.ch\n\
*Eventually I'll put a link here for a man-page?*\n\
\n\
Currently Dockerized Release Managers only support S3-hosted repositories.\n\
We will add support to existing repositories later.\n\
\n\
* *****************************************************************\n" > /etc/motd

COPY dockerfile_commands.sh dockerfile_commands.sh

CMD chmod +x dockerfile_commands.sh ; ./dockerfile_commands.sh
```

This image uses the CERN CentOS 7 image as a base image, as the current release management machines all run on CentOS 7. It has a global environment variable called `REPO_NAME`, which indicates to the container the repository it should be able to write to. The Dockerfile automatically sets this to `default_repo_name`, but this should be overwritten when the container is run as the user inputs the correct repository name as a parameter. This variable gets passed into the command statements in order to create the necessary directories to be able to write to the given repository. Next, the first `RUN` line installs both the `cvmfs` and `cvmfs_server` packages. Note that in this Dockerfile, the development versions of `cvmfs` and `cvmfs_server` are installed rather than the stable versions. This is because there was a bug in the S3 repository upload process, but in the future the stable versions should be used. The next `RUN` statement includes a "message of the day" to give user instructions for general release management commands at container start-up.

The last two lines are integral to allowing the union mount file system (called OverlayFS [11]) to write to the repository. The `dockerfile_commands.sh` file is a bash script that resides in the same directory as the Dockerfile. It gets copied to the work directory of the container so that it is accessible, and then it is executed right away upon start-up. The `dockerfile_commands.sh` file is shown below:

```
#!/bin/bash

mkdir -p /var/spool/cvmfs/$REPO_NAME/rdonly
mkdir -p /var/spool/cvmfs/$REPO_NAME/scratch/current
mkdir /var/spool/cvmfs/$REPO_NAME/scratch/wastebin
mkdir /var/spool/cvmfs/$REPO_NAME/ofs_workdir
mkdir /var/spool/cvmfs/$REPO_NAME/tmp
mkdir /var/spool/cvmfs/$REPO_NAME/cache
mkdir /etc/cvmfs/repositories.d/$REPO_NAME/
mkdir /cvmfs/$REPO_NAME
mys3path=$(grep --only-matching -E "\/etc\/cvmfs\/(.*?)\/" /tmp/config_files/server.conf)
mkdir -p $mys3path
cp /tmp/config_files/client.conf /etc/cvmfs/repositories.d/$REPO_NAME/
cp /tmp/config_files/server.conf /etc/cvmfs/repositories.d/$REPO_NAME/
cp /tmp/config_files/$REPO_NAME.* /etc/cvmfs/keys/
cp /tmp/config_files/mys3.conf $mys3path
cp /tmp/config_files/fstab /etc/
bucketname=$(grep --only-matching -P "(?<=CVMFS_S3_BUCKET\=).*" /tmp/config_files/mys3.conf)
curl http://cs3.cern.ch/$bucketname/$REPO_NAME/.cvmfsreflog | cvmfs_swissknife hash -a sha1 \
     > /var/spool/cvmfs/$REPO_NAME/reflog.chksum
cvmfs_server mount -a
/bin/bash
```

The first few `mkdir` lines create the necessary repositories for OverlayFS. Note that they all use the `REPO_NAME` environment variable, given by the user when running the container. The following `cp` lines take the configuration files necessary for CVMFS transactions from a mounted volume at `/tmp/config_files` and copy them into the correct directories. This is done so that the process that updates the repository can access them properly when publishing the changes. The `curl` statement re-evaluates the expected reflog hash for the repository and injects it into the correct checksum file. Essentially, the reflog is a list of hashes of a repository's root catalogs over time, and it cannot be verified for data integrity upon download with the content hash signatures the same way all the other files can. Thus, the reflog is hashed and stored on the release manager machine after every transaction. At the next transaction, users can fetch the reflog from the repository and can verify it is what is expected for the repository, and that there is no data corruption. At the moment, this command gets around having a real data integrity check because it just evaluates the reflog for the repository as it currently is, no matter what the repository looks like, and therefore the "check" will always be valid. However, the CERN S3 setup is a trusted environment, so for the moment this check is not necessary to begin with; it was simply included in the Dockerfile to safeguard against errors [12]. Past the checksum command, everything is mounted, and then the `/bin/bash` command is run for interactivity with a client. It should be noted that this is the default command file, but this can be changed to a client's own bash script in case they have different needs.

2.3 Integrating Dockerized Release Management into cvmfs_server Commands

Currently, in order to make changes to a repository, clients must log into an `lxcvmfs*` machine, and then run `cvmfs_server` commands to interact with their repository. Such commands include `cvmfs_server transaction`, which begins the release management process, and `cvmfs_server publish`, which ends the release management process and saves the changes in the repository. In order for users to be able to run this release management container easily, a new function was added to the CVMFS source code integrating a new `docker` command to the list of `cvmfs_server` subcommands. The biggest contribution was adding the `cvmfs_server_docker.sh` file to the source code, which allowed Dockerized release management to be integrated into the rest of the `cvmfs_server` command code. The source code can be found at [13].

The `cvmfs_server_docker.sh` file follows the structure of the rest of the `cvmfs_server` commands, so it creates a function that gets called when a user runs the `cvmfs_server docker` command. This `cvmfs_server_docker()` function first parses through the options, which include the following:

- -i: indicate any non-default images the user might want to run
- -c: indicate the absolute path to a configuration file directory that may need to be mounted at `/tmp/config_files` on the container
- -f: indicate the absolute path to a non-default command file to be mounted at `/dockerfile_commands.sh` on the container, which will be run at launch time

The function then checks that the user put in exactly one repository name, as the release management container currently can only support one repository at a time. Then, the specified Docker image is pulled and run based on the options given. The most general run statement is shown here:

```
docker run -i -t -e REPO_NAME=$repo_name -v $command_file:/dockerfile_commands.sh \
     -v /var/spool/cvmfs -v $config_file_dir:/tmp/config_files:ro --cap-add SYS_ADMIN \
     --device /dev/fuse --rm $image_name
```

In the run statement for this Docker image, there are a number of considerations to note. First, it was necessary to mount an ext4 volume at `/var/spool/cvmfs` as scratch space for OverlayFS [5], using the `-v` option in the `docker run` command. Due to the current default image set-up, it was also necessary to add a volume from directory on the parent machine that contained all the configuration files needed for release management functionality. This was also done with the `-v` option, and was mounted at `/tmp/config_files` (as seen in the `dockerfile_commands.sh` file when these configuration files are copied from this location to others). If the user wants to run a bash script upon container launch different from the default `dockerfile_commands.sh` file, then they can also mount this new script (given as an absolute path to this script) at `/dockerfile_commands.sh` in the work directory, again using the `-v` option. The `-i` and `-t` options are to ensure interactivity with the user when the container runs, while the `-e` option lets the user redefine the repository's name at runtime. Finally, `--cap-add SYS_ADMIN --device /dev/fuse` mounts the `/dev/fuse` device in the container with proper permissions, which is necessary for CVMFS to work.

In implementing the new function, the CVMFS GitHub repository was forked and a new `docker` branch was added to the forked repository. All new changes were made in this branch, located at `https://github.com/lilhuang/cvmfs/tree/docker`.

3 Results

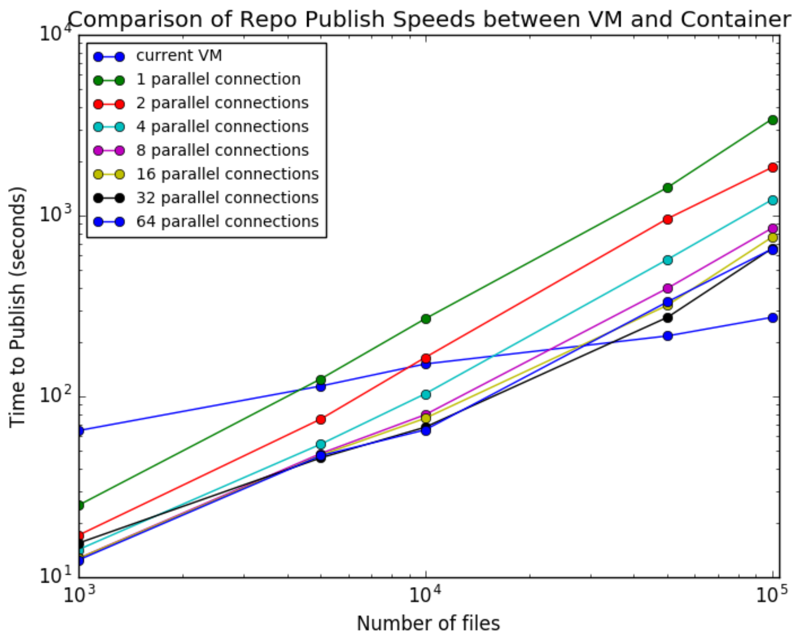

Figure 4: Comparison of current release management VM publish speeds to container publish speeds, uploading at different numbers of parallel connections and different numbers of tiny text files.

After implementing a Dockerized release management system with a remote repository, it was of interest to test its performance. The first question, as well as the most important, was to determine whether it works; whether release management was possible using this new system. It was also important to investigate whether running in a container would slow the publishing process to any significant capacity, or if the S3 repository would have a negative impact on publish speeds. With respect to the

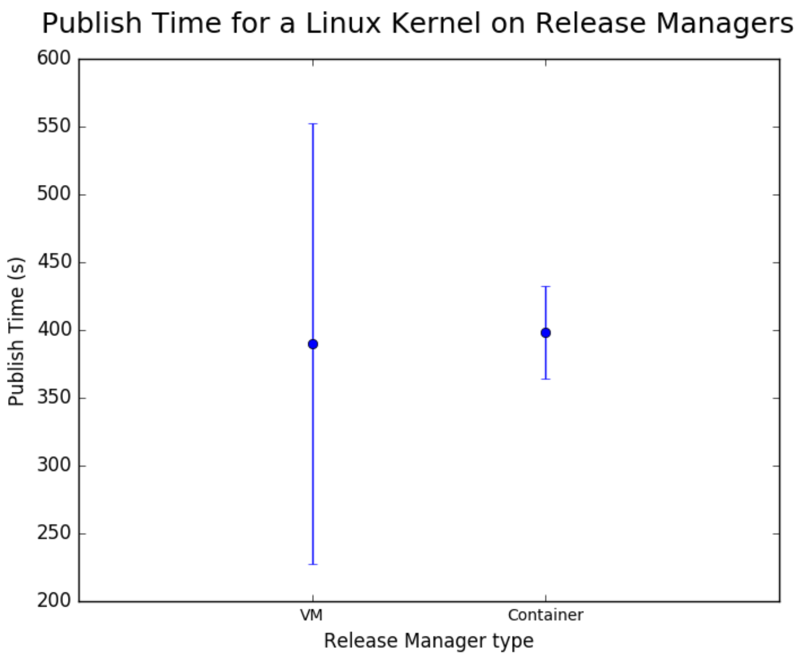

Figure 5: Comparison of current release management VM publish speeds to container publish speeds at 32 parallel connections, uploading the current stable Linux Kernel.

lattermost question, it was hypothesized that increasing the number of parallel connections at which data was uploaded to the repository would speed up publishing.

Unfortunately, it was not possible to separate the effects of the S3 repository and the container, due to time constraints. Nonetheless, the publishing times of the release management machine for different publish sizes (indicated by the file number) was compared to the publish times of the container, configured to upload with various numbers of parallel connections. The results are shown in Figure 4. Ten tests of the same kind were performed for each set of parameters (number of distinct files, number of parallel connections, container or VM). Firstly, this plot indicates that the container is able to upload changes to a remote repository, and therefore the system works. It is also evident that in general, when the publish size is relatively small (1k files), the container-and-S3 combination publishes faster, while with a larger publish size (100k files), the original release management system is faster. In addition, increasing the number of parallel connections increases publish speeds in general. However, there seems to be a limit to the efficacy of this, as 32 parallel connections performed about as well as 64. This implies that increasing past 64 parallel connections would not have much impact on publish speeds.

However, the first test featured only tiny unique files (less than 50 characters in each file), so a further test was needed to see if publishing more substantial data would make a difference. So, the times for publishing the most recent stable Linux Kernel (around 50k files) on the virtual machine as opposed to the container were compared, and the results are shown in Figure 5. Due to the previous test showing that more parallel connections would yield the best results, the container was configured to upload at 32 parallel connections. Ten iterations of this test were performed for each release management system, and the medians of these publish times were plotted along with the standard deviation as error bars. As indicated by the plot, while the virtual machine had similar performance to the container, the standard deviation of the publish times was much bigger, showing a larger fluctuation in performance.

4 Conclusion

We were ultimately able to determine that a containerized release management system is possible, and that it is made possible with remote storage. We tested a S3-hosted repository, which works out-of-the-box with just a little tweaking of configuration files, and created a Docker image for a release management machine that is able to perform transactions on the remote repository. Although we were not able to test the performance of this new system extensively, we were able to perform a few tests centered on the publish speeds of current and developing release management systems, and the results are not discouraging. A pull request to integrate the `cvmfs_server_docker()` function into the next release of CVMFS is the next step, and will make it possible to continue this work in the future.

References

[1] Blomer J, Aguado-Sanchez C, Buncic P, and Harutyunyan A, 2011, Distributing LHC application software and conditions databases using the CernVM file system, *Journal of Physics: Conference Series* **331**

[2] De Salvo A, De Silva A, Benjamin D, Blomer J, Buncic P, Harutyunyan A, Undrus A, Yao Y, 2012, Software installation and condition data distribution via CernVM File System in ATLAS, *Journal of Physics: Conference Series* **396**

[3] Blomer J, Buncic P, Charalampidis I, Harutyunyan A, Larsen D, and Meusel R, 2012, Status and future perspectives of CernVM-FS, *Journal of Physics: Conference Series* **396**

[4] CVMFS documentation: Setting up a Replica Server (Stratum 1), `http://cvmfs.readthedocs.io/en/stable/cpt-replica.html`

[5] CVMFS documentation: Creating a Repository (Stratum 0), 2016, `http://cvmfs.readthedocs.io/en/stable/cpt-repo.html`

[6] Ceph, 2017, `http://ceph.com/`

[7] Docker, 2017, `https://www.docker.com/`

[8] CVMFS documentation: Creating a Repository (Stratum 0): CernVM-FS Repository Creation and Updating: S3 Compatible Storage Systems, 2016, `http://cvmfs.readthedocs.io/en/stable/cpt-repo.html#s3-compatible-storage-systems`

[9] Docker documentation, `https://docs.docker.com/`

[10] GitLab Container Registry, `https://gitlab.cern.ch/cvmfs/it-cvmfs-docker/container_registry`

[11] OverlayFS Wikipedia page, `https://en.wikipedia.org/wiki/OverlayFS`

[12] Conversation with Jakob Blomer, CERN EP-SFT Department

[13] CVMFS GitHub repository, `https://github.com/cvmfs/cvmfs`