New directions in the CernVM file system

IOP Conf. Series: Journal of Physics: Conf. Series **898** (2017) 062031 doi:10.1088/1742-6596/898/6/062031

New directions in the CernVM file system

Jakob Blomer, Predrag Buncic, Gerardo Ganis, Nikola Hardi, René Meusel, and Radu Popescu

CERN, Geneva, Switzerland

E-mail: jblomer@cern.ch

Abstract. The CernVM File System today is commonly used to host and distribute application software stacks. In addition to this core task, recent developments expand the scope of the file system into two new areas. Firstly, CernVM-FS emerges as a good match for container engines to distribute the container image contents. Compared to native container image distribution (e.g. through the "Docker registry"), CernVM-FS massively reduces the network traffic for image distribution. This has been shown, for instance, by a prototype integration of CernVM-FS into Mesos developed by Mesosphere, Inc. We present a path for a smooth integration of CernVM-FS and Docker. Secondly, CernVM-FS recently raised new interest as an option for the distribution of experiment conditions data. Here, the focus is on improved versioning capabilities of CernVM-FS that allows to link the conditions data of a run period to the state of a CernVM-FS repository. Lastly, CernVM-FS has been extended to provide a name space for physics data for the LIGO and CMS collaborations. Searching through a data namespace is often done by a central, experiment specific database service. A name space on CernVM-FS can particularly benefit from an existing, scalable infrastructure and from the POSIX file system interface.

1. Introduction

The CernVM File System is commonly used to host and distribute scientific application software to grids, clouds, and supercomputers [1]. Its key features include a POSIX compliant interface, HTTP transport, multi-level caching, versioning, strong consistency, and end-to-end data integrity. CernVM-FS not only serves the software distribution needs of large collaborations (such as LHC or neutrino experiments) but also a long tail of smaller experiments. As of September 2016, we are aware of more than 50 repositories that collectively host more than 350 Million files. Known software installation services are operated, for instance, by CERN, EGI, OSG, DESY, NIKHEF, and Compute Canada.

Today there are two popular use cases:

(i) The distribution of production software, such as /cvmfs/cms.cern.ch. New software versions are added to a stable directory tree on a weekly or daily basis.

(ii) The distribution of "nightly builds", such as /cvmfs/lhcbdev.cern.ch. These repositories face large volatility, where new software versions overwrite old versions, e.g. last week's builds. In contrast to production repositories, nightly build repositories are garbage collected, that is old and deleted versions are not captured by CernVM-FS' internal versioning. Nightly build repositories are not or only sparsely replicated in order to ensure a quick turn-around from publishing to availability on clients.

In the following sections, we present new use cases and the corresponding new and planned features in support of these use cases.

2. Container Distribution

Linux container virtualization [2] raised a great deal of interest in the computing industry and academia alike. For instance, the major cloud providers Amazon, Microsoft, and Google now all offer and propagate container technology as the next-generation application deployment means. Linux containers combine *cgroups* and kernel namespaces for application isolation with a *container image* that provides a Linux root file system different from the host's file system.

There are a number of *container engines*, such as *Docker*, *rkt*, or *Singularity* [3], which facilitate the maintenance and orchestration of containers. These container engines take a simple approach to container image distribution. Either a directory in the host's mount tree is used (i.e. the "distribution" of that directory needs to be addressed elsewhere) or the images are distributed as one or few tarballs containing *layers* of the final container image. Layers are a coarse-grained division of container images; a layer can be, for instance, the Ubuntu base operating system, or the Hadoop software stack. This results in the challenge to distribute gigabytes worth of image layers to a large number of machines. For Virtual Machine images, this very problem is addressed using CernVM-FS by the CernVM virtual appliance [4]. We expect to see the same bandwidth and space savings for container images. For instance, in order to run some basic tutorials with the R container on Docker Hub, out of the 1 GB R container only a few tens of megabytes are actually accessed and need to be distributed.

Connecting a container engine to CernVM-FS is trivial if the container image can be provided as a directory on the host, as it is in case of Singularity or the Mesos containerizer. More work is required to connect the widespread Docker container engine with its layer-based image distribution to CernVM-FS. Docker has a central component, the *registry*, that stores layers as content-addressed tarballs together with image manifests that list the layer hashes that are needed to compose the image. In order to run a container, all its uncached layer tarballs need to be downloaded and locally extracted first. Conversely, a tarball can be created from the content of a locally modified container and pushed back to the registry. The component on the Docker host that manages image layers is called a *graph driver*. Since Docker version 1.13, graph drivers are pluggable and open for external extensions.

Ongoing work aims at a CernVM-FS graph driver for Docker. This graph driver still uses the standard registry as a database for available images. However, instead of pushing the actual image contents to the registry, the graph driver instead pushes a *thin image*. The thin image only contains the whereabouts of the layers in CernVM-FS. The actual layer content resides in the uncompressed form in a CernVM-FS directory. Depending on whether the graph driver starts a normal image or a thin image, it can behave like the standard graph driver or it uses directories from a CernVM-FS mount point to provide the container's root file system. This idea for seamless integration with the Docker ecosystem is borrowed from the *Slacker* container image storage system [5]. When pushing thin images back to the registry, a tarball with the local changes is sent to the CernVM-FS server for publication before the thin image itself is pushed to the registry.

3. Conditions Data

Hosting detector conditions data, i.e. data about the state of the detector at the time of data taking, is a use case that is foreseen in the design of CernVM-FS but it is not yet fully exploited. In order to support the full lifecycle of hosting conditions data from data taking to analysis (and its preservation), extensions to CernVM-FS in two areas are in development.

Firstly, we add support for multiple release manager machines that can publish content concurrently as long as they work on different directory subtrees in the repository (see Figure 1).

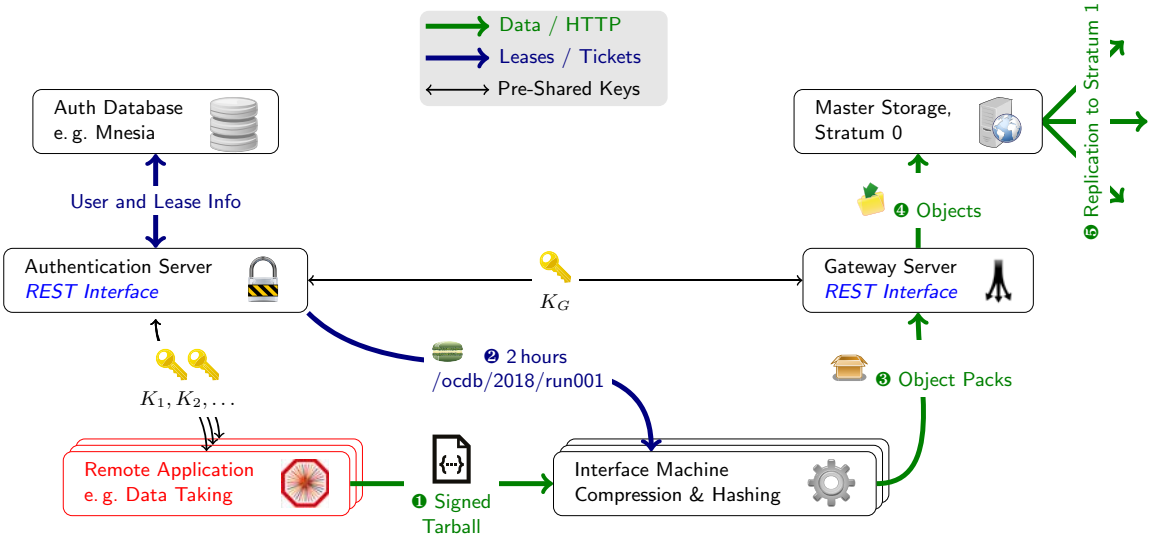

Figure 1. Multiple release manager machines acquire leases for certain subtrees of the directory from a central lease manager. They can then concurrently hash and compress change sets. A gateway service in front of the storage receives the signed object packs from multiple release manager for final committing. The object packs can be reused to speed up replication to Stratum 1 servers.

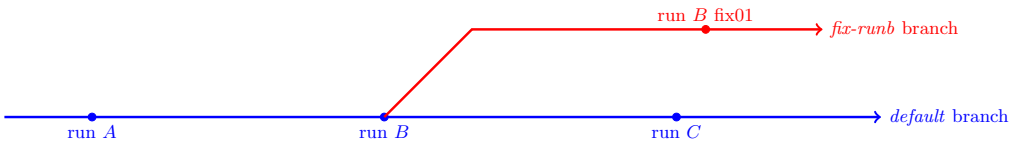

Figure 2. Possible use of file system snapshots and branching for a repository hosting conditions data.

Besides performance improvement, multiple release manager machines can simplify operations. For instance, provided a suitable directory structure, the data stream from data taking can be processed independently from reprocessed conditions data from previous runs.

Secondly, we extend and better expose CernVM-FS' internal versioning. The existing named file system snapshots ("tags") can be used to indicate a meaningful state of the conditions data that should be used for analyses (see Figure 2). In order to scale the number tags beyond 10 000-100 000, we add support for archiving old tags and making them immutable. For swift access to the active tags, support is added for an automatically created, hidden directory `.cvmfs/snapshots`. Every tag becomes a subdirectory of this path and provides access to the corresponding file system snapshot. This interface is analogous to the `.zfs/snapshots` directory in ZFS. Newly added support for *branching* allows to create new file system snapshots based on an arbitrary earlier snapshot, not only the latest snapshot. This provides means for publishing hotfixes to a released file system state.

4. Data Namespace

A combination of contributions to CernVM-FS allow for distributing a namespace of an existing data repository (see Figure 3). In contrast to library based data access tools, clients benefit from CernVM-FS' POSIX interface that allows easy integration with existing applications (for

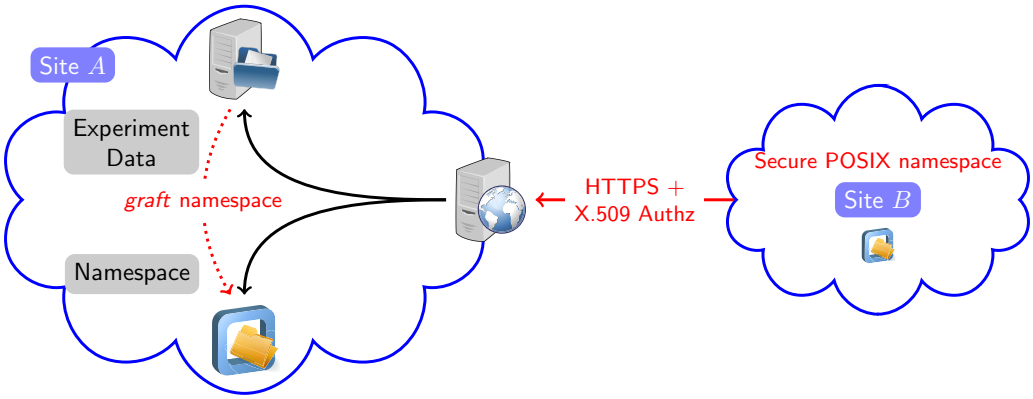

Figure 3. A data namespace on CernVM-FS. Data files are provided on a file server different from the CernVM-FS catalogs. Access is protected through HTTPS with X.509 authorization.

instance: `ls`). Clients can also benefit from CernVM-FS' built-in data integrity verification through content hashes. The added convenience comes at the cost of performance—CernVM-FS is not primarily optimized for high throughput—and more complex configuration.

The new features in support of this use case are:

Grafting. Grafting allows for publications that only define the meta data of files (size, content hash, name, etc.) without actually processing them. Thus data files can be represented in a CernVM-FS directory tree without the need to process them.

Uncompressed files. By default, CernVM-FS uses the content hash of the zlib compressed version of files. In order to avoid compressing existing files, CernVM-FS can now be instructed to store the content hash of the uncompressed version of files in its catalogs.

HTTPS access. HTTPS instead of HTTP access can be used to ensure data confidentially during transfer. CernVM-FS can also present a user's X.509 certificate to authenticate her to the origin web server. HTTPS access implicitly prevents the use of site proxies, i.e. the origin servers need to be properly sized.

Authorization helpers. 3rd party authorization helpers can communicate with a CernVM-FS client on a host through a well-defined protocol. Authorization helpers can locally grant or deny access to a CernVM-FS mount point based on the user and process that tries to access it.

A detailed description of this use case and the deployment in OSG's *StashCache* is subject of a separate publication in these conference proceedings [6]. The contributions help for other use cases, too. For instance, conditions data are usually already compressed and benefit from not compressing them again. Different "views" of existing directory trees can be efficiently created through grafting. For instance, one can create a view that combines binaries from various packages in different directories in a single `bin` directory. The HTTPS access and authorization plugins can be used to distribute non-public software.

5. Cache Plugins

Potentially large (data) files and deployment beyond managed resources (e.g. supercomputers) require flexible cache access by the client. The local cache is a central component in CernVM-FS, all data accesses are routed through the local cache. While a file on CernVM-FS is held open by an application, it has to reside in the local cache. In its default deployment, CernVM-FS uses a local directory as a cache.

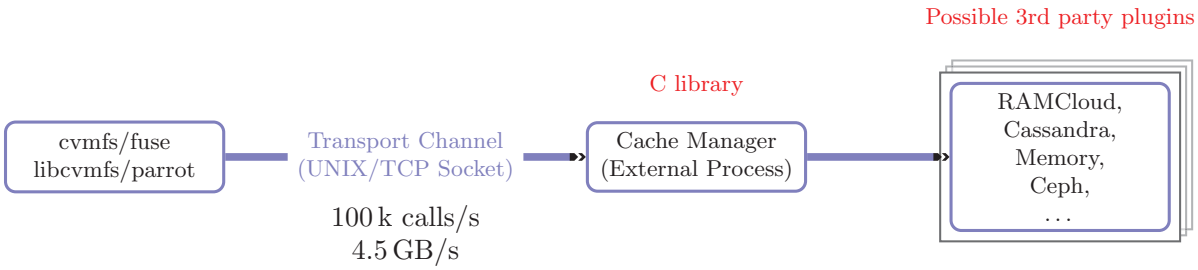

Figure 4. External processes provide the functionality that previously was hard-wired to a (local) cache directory. Cache plugins can be developed and deployed independently of the CernVM-FS client itself.

A new pluggable cache interface allows to connect external cache managers processes to the CernVM-FS client through a well-defined transport protocol (see Figure 4). The external process is connected to the CernVM-FS client either through a local UNIX domain socket or through a TCP port. Through a UNIX domain socket, the transport channel sustains 100 k calls/s and 4.5 GB/s. Writing of external cache plugins is being aided by a C library (see Appendix). Proof-of-concept cache plugins exist for RAMCloud [7] and for a local, in-memory cache. Another planned plugin should use XRootD [8] as a CernVM-FS cache. We are aware of other groups that are working on 3rd party plugins.

In addition to cache plugins, new support for tiered caches allows to combine two cache options. For instance, one could deploy a Ceph cluster cache plugin as a lower layer cache, with a local directory or an in-memory cache plugin as an upper layer. Obviously, many other deployment options are conceivable.

6. Status and Summary

In this contribution, we outlined new use cases and developments in CernVM-FS. In particular, we presented extensions to CernVM-FS' versioning capabilities, the publishing tools, client-side caching, and integration with Container engines. New and recent use cases include hosting conditions data and other potentially large files, non-public files, container distribution, and deployment on supercomputers. Two new plugin interfaces, authorization helpers and cache plugins, aim at increasing the developer community.

As of February 2017, there is a proof-of-concept CernVM-FS graph driver plugin for Docker. Most of the developments for extending the versioning capabilities are merged. Support for cache plugins and a tiered cache is merged, too. All the developments described in section 4 on data namespace support are released.

References

[1] Blomer J, Buncic P, Meusel R, Ganis G, Sfiligoi I and Thain D 2015 *Computing in Science and Engineering* **17** 61–71
[2] Menage P B 2007 *Proc. of the Ottawa Linux Symposium* pp 45–57
[3] Kurtzer G M 2016 Singularity 2.1.2 - Linux application and environment containers for science URL `https://doi.org/10.5281/zenodo.60736`
[4] Blomer J, Berzano D, Buncic P, Charalampidis I, Ganis G, Lestaris G, Meusel R and Nicolaou V 2014 *Journal of Physics: Conference Series* **513**
[5] Harter T, Salmon B, Liu R, Arpaci-Dusseau A C and Arpaci-Dusseau R H 2016 *Proc. 14th USENIX Conference on File and Storage Technologies (FAST'16)*
[6] Bockelman B *et al.* Accessing data federations with cvmfs Talk at this conference

[7] Ousterhout J, Gopalan A, Gupta A, Kejriwal A, Lee C, Montazeri B, Ongaro D, Park S J, Qin H, Rosenblum M, Rumble S, Stutsman R and Yang S 2015 *ACM Transactions on Computer Systems* **33**
[8] Dorigo A, Elmer P, Furano F and Hanushevsky A 2005 *WSEAS Transactions on Computers* **4** 348–353

Appendix

The following listing outlines the callback routines developers for external cache plugins need to implement.

```
// Reading data
int cvmcache_chrefcnt(struct hash object_id, int change_by);
int cvmcache_object_info(struct hash object_id,
                         struct object_info *info);
int cvmcache_pread(struct hash object_id,
                   int offset, int size,
                   void *buffer);

// Transactional writing in fixed-sized parts
int cvmcache_start_txn(struct hash object_id, int txn_id,
                       struct info object_info);
int cvmcache_write_txn(int txn_id, void *buffer, int size);
int cvmcache_abort_txn(int txn_id);
int cvmcache_commit_txn(int txn_id);

// Optional: quota management
int cvmcache_shrink(int shrink_to, int *used);
int cvmcache_listing_begin(...);
int cvmcache_listing_next(int listing_id, ...);
int cvmcache_listing_end(int listing_id);
```