EPJ Web of Conferences **214**, 03036 (2019) https://doi.org/10.1051/epjconf/201921403036
CHEP 2018

Towards a responsive CernVM-FS architecture

Radu Popescu[1,*], *Jakob* Blomer[1,†], and *Gerardo* Ganis[1,‡]

[1]CERN, Esplanade des Particules 1, 1217 Meyrin, Switzerland

Abstract. The CernVM File System (CernVM-FS) provides a scalable and reliable software distribution service implemented as a POSIX read-only filesystem in user space (FUSE). It was originally developed at CERN to assist High Energy Physics (HEP) collaborations in deploying software on the worldwide distributed computing infrastructure for data processing applications. Files are stored remotely as content-addressed blocks on standard web servers and are retrieved and cached on-demand through outgoing HTTP connections only. Repository metadata is recorded in SQLite catalogs, which represent implicit Merkle tree encodings of the repository state. For writing, CernVM-FS follows a publish-subscribe pattern with a single source of new content that is propagated to a large number of readers. This paper focuses on the work to move the CernVM-FS architecture in the direction of a responsive data distribution system. A new distributed publication backend allows scaling out large publication tasks across multiple machines, reducing the time to publish. For the faster propagation of new published content, the addition of a notification system allows clients to subscribe to messages about changes in the repository and to request new root catalogs as soon as they become available. These developments make CernVM-FS more responsive and are particularly relevant for use cases where a short propagation delay from repository down to individual clients is important, such as using CernVM-FS as an AFS replacement for distributing software stacks. Additionally, they permit the implementation of more complex workflows, with producer-consumer pipelines, as for example in the ALICE analysis trains system.

1 Introduction

The CernVM File System (CernVM-FS) is the main software delivery system bringing the high-energy physics (HEP) experiment software stacks to the world-wide LHC computing grid (WLCG) [1]. CernVM-FS is implemented as a file system in userspace (FUSE) module which provides a read-only interface to a remote repository containing files and directories. A few concepts are fundamental to the design of CernVM-FS. First, files are stored in the repository as immutable content-addressed chunks. Second, the transfer of file contents is done lazily, on a chunk-by-chunk level, and all transfers use HTTP connections initiated by the client. Finally, the repository metadata is stored in an SQLite catalog, which is itself stored as a content-addressed object in the repository. For scalability reasons, there can be

*e-mail: radu.popescu@cern.ch
†e-mail: jakob.blomer@cern.ch
‡e-mail: gerardo.ganis@cern.ch

multiple catalogs describing a repository, each catalog assigned to a specific directory subtree. In addition to storing metadata, a catalog references the content-addressed chunks of all the files in the repository. As a result, the hash of the root catalog represents the root of an implicit Merkle tree, encoding the entire state of the repository at a given point in time. The entry point into the repository is the manifest, a text file located in the top-level directory of the repository, containing the current root hash of the repository, the revision number and other repository-level metadata.

The classic publication architecture of CernVM-FS involves a single publisher per repository. The publisher, called "release manager" in CernVM-FS terminology, is a machine with write access to the authoritative storage of a repository. On this machine, the repository contents are updated using a transactional approach: once a transaction is opened for a repository, new content can be written into a temporary staging area. Committing the transaction involves hashing and compressing the new or modified content, building new metadata catalogs, and writing the new data and metadata into the repository storage. CernVM-FS clients download the repository manifest and cache it with a time-to-live (TTL) value, typically a few minutes. When the TTL expires, clients download the latest version of the manifest, making the updated content of the repository available. This architecture is robust and efficient, and has been proven to work for the most common use cases [2].

On the release manager, the CernVM-FS server tool can take advantage of multiple CPU cores and benefits greatly from a high throughput in terms of I/O operations, as is realized in solid-state drives and through parallel streams with the Amazon S3 compatible repository storage backend [3]. However, the existence of a single release manager per repository can eventually become a bottleneck, when the change sets being published are large enough. Most CernVM-FS use cases are not sensitive to the propagation delay of changes from repository to clients; however, in cases where this delay does need to be minimized, the TTL-based approach introduces a lower bound for this delay. Additionally, relying solely on TTL expiration for propagating changesets makes building complex publication pipelines both difficult and fragile.

This paper describes work done to make the CernVM-FS publication architecture more responsive; removing the limitations listed here allows horizonal scaling to accommodate an increase in the size of publication payloads as well as implementing new use cases previously prohibited by the TTL-based change propagation.

2 Multiple release managers per repository

The "single-writer" approach has some benefits: a single machine is easier to operate, and, unlike a distributed system, there is no risk of losing consistency between the different parts of the system [4]. The downside is the limited scalability of the CernVM-FS publication process: faster CPUs with more cores and faster I/O devices cannot cope with a potentially unbounded increase in publication workloads. In such a situation, multiple release manager machines, able to publish concurrently to a repository, are needed.

The effective parallelisation of the publication process relies on distributing as much of the work as possible across multiple release manager machines. The most time-consuming part of each publication is processing any new or modified files. It involves dividing the files into chunks, which are then compressed, hashed and saved to the repository storage. Individual files can be processed independently from each other, requiring no synchronisation.

The final part of the publication involves constructing new metadata catalogs for the updated content of the repository. Merging the changes from two concurrently modified subtrees represents a critical section in the parallelisation of the processing, since any modification of the internal state of the repository propagates upwards in the tree of catalogs, resulting in

a new root catalog. An advantage is that concurrently modified subtrees that happen to be described by separate catalogs are merged instantaneously and do not require any additional directory traversal into the sub catalogs.

The separation between parallelisable and serial work informs the design of an improved publication architecture (Fig 1). The multiple release manager machines are mostly unchanged, with respect to the classic architecture, except for the fact that they no longer write directly to the repository storage. Instead, there is a new component in the system, named the repository gateway, which mediates all the interactions between the release managers and the repository storage.

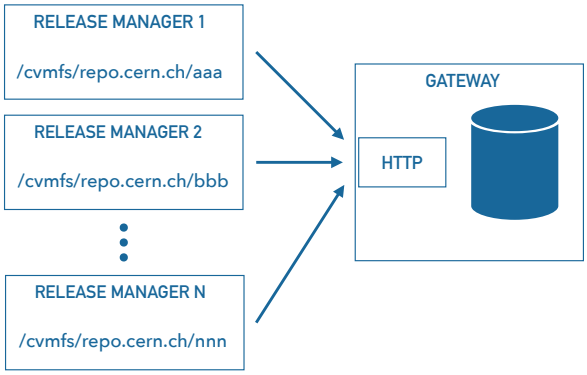

Figure 1. Multiple release managers can publish concurrently to a single CernVM-FS repository through a common repository gateway.

2.1 The repository gateway

The repository gateway is the only component in the new publication architecture with direct access to the repository storage. It receives the processed chunks from the different release manager machines and writes them into the repository. To maintain the consistence of the repository, the gateway prevents different release managers from updating the same paths at the same time. To this end, the gateway serves as an exclusive lock manager for the repository, handing out to each release manager exclusive leases for different repository subpaths. Release managers acquire a lease at the beginning of each transaction, and keep the lease until the publication is finished or the transaction is rolled back. The gateway will only hand out a lease if the requested path is not already covered by an existing lease. For example, the paths `/a/b/c` and `/a/b/d` can be simultaneously locked by the gateway, but only one of `/a/b/c` and `a/b` can be locked at a time.

The gateway exposes an HTTP API that release managers consume. For security, API requests need to be signed with a secret key that is shared by the release managers and the gateway. Each release manager can be configured with a separate key, and keys can be restricted to specific paths inside the repository. This security mechanism can be used to implement shared user repositories, where each user can only write into a personal area of the repository, but can read the personal areas of all the other users.

Scalability and robustness are essential factors for the operation of the gateway, and its implementation reflects this (Fig 2). The gateway is implemented in Erlang [5], a programming

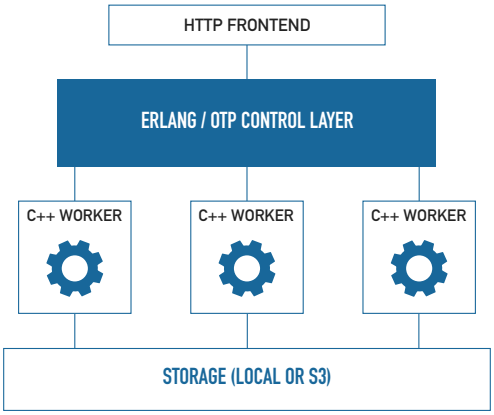

Figure 2. The repository gateway is composed of an Erlang/OTP control layer driving multiple C++ worker processes.

language originating from the telecommunications industry, designed for implementing scalable, fault-tolerant systems that need to offer very high levels of availability. Erlang comes with a standard library, the Open Telecom Platform (OTP) [6], containing reusable application components such as generic request/reply services, finite state machines, event managers and process supervisors. Erlang is used for the HTTP front-end and the control layer of the gateway application. The Erlang control layer drives a number of worker processes implemented in C++. This programming language was chosen for the worker processes to maximise the performance of certain critical functions, but also to allow code reuse between the gateway and release manager components.

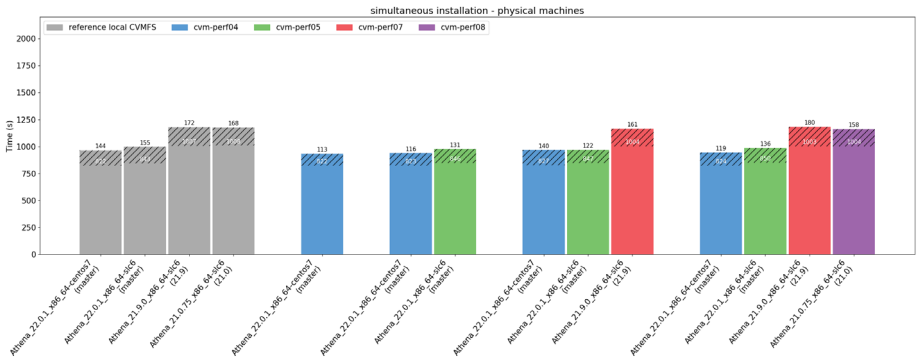

Figure 3. Parallel publication of the Athena framework with a different number of release managers connected to a single repository gateway. The grey bars represent the time to build and publish each configuration of Athena on a single release manager without a gateway. The three sets of coloured bars show the time to build and publish multiple configurations of Athena, in parallel, on multiple release manager machines, with one configuration assigned per machine.

An initial benchmark was set up to measure the scalability of the publication architecture: four release manager machines were provisioned for concurrent publication into a CernVM-

FS repository. Multiple configurations of the ATLAS Athena analysis framework [7] were built and published in parallel on the four release manager machines. Results show that the parallel publication architecture achieves *weak scaling* (Fig 3): publishing a payload of size $S_N = NS_S$ using N release managers takes the same amount of time as publishing a payload of size S_1 on a single release manager [8]. This indicates the new parallel publication architecture is able to scale out horizontally to meet an increase in publication workload, meaning the volume of publications is now limited by the bandwidth into the repository gateway node or the bandwidth of the storage backend. As each repository gateway node is expected to serve at most a few tens of concurrent publishers, the bandwidth limitation should not pose problems in practice.

3 Repository activity notification system

CernVM-FS clients request updated copies of repository manifests when their TTL value expires. This pull-based approach is very robust, since there is no central point of failure preventing the propagation of changes to clients, as in the case of a push-based system. Additionally, due to the spacing in time of requests coming from different clients, the content distribution infrastructure is protected from any "thundering herd" situation.

The inability to know precisely when a repository has been updated is not a problem for most CernVM-FS applications, which are not sensitive to a propagation delay below a few minutes. We have identified, however, at least two use cases better served by a more precise propagation system. The first is the distribution of conditions databases, which during data taking change at a much higher rate than software repositories. In a conditions data workflow, it is desired to process new data samples as soon as they are available, to avoid having a pileup of new samples. The second case is the construction of a complex software build and test pipeline, where later stages of the pipeline depend on artifacts published at earlier stages of the pipeline already being available on replicas of the repository. From the point of view of the pipeline, the replication of a repository to stratum-1 servers is an asynchronous process; the replication status can only be learned through polling.

To address this type of use case, a new notification system was designed, centered around a message broker relaying activity messages between interested parties (see Fig 4).

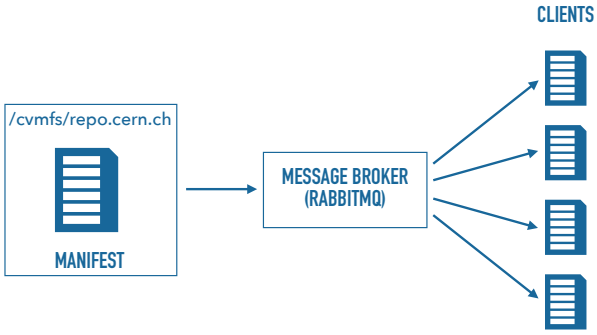

Figure 4. A RabbitMQ message broker carries repository activity messages to interested clients.

3.1 Message protocol

This activity message informs the consumers of the notification system that a given repository has changed state. Messages are encoded as JSON objects; the structure of the message is

shown in Fig 5. The protocol version is specified for forward compatibility reasons, and each message is tagged with a unique identifier, which could be used to implement at-most-once delivery semantics, if needed. The name of the repository is included in the message, together with the Base64 encoding of the new signed manifest of the repository. The message broker checks the signature of the manifest, only forwarding to consumers messages with a valid signature.

```
{
  "version"    : <protocol version>,
  "uuid"       : <unique identifier of the message>,
  "timestamp"  : <time when the message was constructed>,
  "type"       : "activity",
  "repository" : <name of the repository>,
  "manifest"   : <base64 encoding of manifest>
}
```

Figure 5. Structure of a repository activity message

3.2 The message broker

RabbitMQ [9], an open-source, high-performance, message broker was chosen to drive the notification system. This choice was motivated by the excellent performance of RabbitMQ, its extensive documentation and ease of use, and its compatibility with different programming languages. The ability of the message broker to handle a large number of connections is also important. The message frequency is expected to be low, but a single broker should be able to handle $O(10^5)$ simultaneous clients. RabbitMQ is implemented in Erlang/OTP and is able to scale to a very large number of clients on a single machine. Indeed, preliminary tests show that the target maximum number of connections can be handled by a single commodity server (64 GB RAM, 16 CPUs).

The broker implements a publish/subscribe communication pattern: each activity message for a certain repository is delivered to all consumers stating their interest in the repository. Upon connecting, clients immediately receive the last message published for the target repository.

By default, a username and a password are used to connect to a RabbitMQ instance, although multiple users with different permission levels can be configured. The message broker should be freely accessible by a large number of message consumers, making it undesirable to distribute the connection credentials with the CernVM-FS client packages. This led us to develop a front-end for the message broker, also using Erlang/OTP, to serve as an adapter between client and the broker (Fig 6).

The front-end accepts new activity messages over HTTP and verifies their signature, forwarding valid messages to the message broker over AMQP [10]. Clients establish long-running connections to the front-end over Websockets, receiving notifications as they become available. The front-end establishes a single AMQP consumer connection to the message broker for each repository and forwards messages to the relavant Websocket connections it maintains to the clients. This separation of concerns means that the front-end is responsible for the authentication of messages and for maintaining connections from a potentially large number of clients. It is also the only participant with the credentials to connect to the message broker. The message broker is reponsible for maintaining the message queues, routing messages based on topic and ensuring queue persistance in the case of failures.

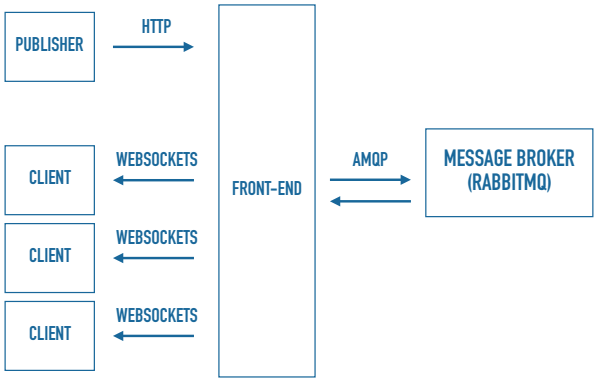

Figure 6. A front-end for the RabbitMQ message broker handles connections over Websockets from a large number of clients and receives messages over HTTP, checking their signatures.

3.3 Messaging tools

The CernVM-FS server toolset was extended with a new command-line tool for interacting with the notification system. In publisher mode, the tool is given the URL of a CernVM-FS repository and the URL of a notification system front-end. It will download the repository manifest and submit it as a repository activity message to the notification system. In consumer mode, the tool is given the same URL as in publisher mode, together with a repository name, and will establish a consumer connection to the notification system. There is the option to quit or to keep the connection active upon receiving an activity message.

In addition to the command-line tool which can be used for various scripting use-cases, the CernVM-FS FUSE client has also been extended to make optional use of the notification system. The FUSE client can be configured with the URL of a notification system, in which case it will establish a connection to the notification system and consume repository activity messages. Upon receiving such a message, the mount point is explicitly updated and the latest repository manifest is loaded. This functionality works in tandem with the TTL-based mechanism: in the absence of notifications from the server, the FUSE client still requests an updated manifest when the TTL of the cached manifest copy expires.

4 Summary

In this paper we described two new lines of development in the CernVM-FS project, with the aim of making the file distribution system more flexible and responsive. First, the publication architecture is extended beyond the original single-writer design, with the ability of publishing payloads into a repository from multiple release managers simultaneously. The consistency of the repository is ensured by a new server component, the repository gateway, handing out exclusive leases to different repository subpaths and acting as a mediator between the release managers and the storage backend of the repository. This new architecture is horizontally scalable to handle publication volumes which are expected to grow. The improved access control implemented in the gateway also permits the creation of shared repositories, that restrict writing access such that users can only publish to their own (publicly readable) sub path.

Second, a new notification system is introduced to propagate changes from CernVM-FS repository to clients with very low delays. This system allows interested clients to be notified

of changes in repositories as soon as they happen, opposed to the existing poll-based mechanism. In addition to operating FUSE clients with low propagation delay, the notification system facilitates the construction of complex automated pipelines which react to CernVM-FS repository activity.

We would like to acknowledge the work of Tomáš Stefan who benchmarked the publication of the ATLAS Athena framework with multiple CernVM-FS release managers.

References

[1] J. Blomer, C. Aguado-Sánchez, P. Buncic, A. Harutyunyan, *Distributing LHC application software and conditions databases using the CernVM file system*, in *Journal of Physics: Conference Series* (IOP Publishing, 2011), Vol. 331, p. 042003

[2] J. Blomer, P. Buncic, R. Meusel, G. Ganis, I. Sfiligoi, D. Thain, *Computing in Science & Engineering* **17**, 61 (2015)

[3] M. Arsuaga-Ríos, S.S. Heikkilä, D. Duellmann, R. Meusel, J. Blomer, B. Couturier, *Using S3 cloud storage with ROOT and CVMFS*, in *Journal of Physics: Conference Series* (IOP Publishing, 2015), Vol. 664, p. 022001

[4] E.A. Brewer, *Towards robust distributed systems*, in *PODC* (2000), Vol. 7

[5] J. Armstrong, *Programming Erlang: Software for a concurrent world* (Pragmatic Bookshelf, 2013)

[6] S. Torstendahl, *Ericsson Review (English Edition)* **74**, 14 (1997)

[7] K. Cranmer, *The ATLAS Analysis Architecture* (2007), `https://cds.cern.ch/record/1047631`

[8] T. Stefan, *Testing and Improving Deployment of ATLAS Releases to CVMFS* (2018), `http://cds.cern.ch/record/2641384`

[9] RabbitMQ Contributors, *RabbitMQ: Messaging that just works*, `http://www.rabbitmq.com` (2008–2018)

[10] S. Vinoski, *IEEE Internet Computing* **10** (2006)