# Scaling the EOS namespace – new developments, and performance optimizations

*Georgios* Bitzes[1,*], *Elvin Alin* Sindrilaru[1,**], and *Andreas Joachim* Peters[1,***]

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

**Abstract.**
EOS is the distributed storage solution being developed and deployed at CERN with the primary goal of fulfilling the data needs of the LHC and its various experiments. Being in production since 2011, EOS currently manages around 256 petabytes of raw disk space and 3.4 billion files across several instances.
Nowadays, EOS is increasingly being used as a distributed filesystem and file sharing platform, which poses scalability challenges on its legacy namespace subsystem, tasked with keeping track of all file and directory metadata on a particular instance.
In this paper we discuss said challenges, and present our solution which has recently entered production. We made several architectural improvements to the overall system design, the most important of which was introducing QuarkDB, a highly-available datastore capable of serving as the metadata backend for EOS, tailored to the needs of the namespace.
We also describe our efforts in providing comparable latency and performance to the legacy in-memory implementation, both when reading through the use of extensive caching and prefetching, and when writing through the use of latency-hiding techniques involving a persistent, back-pressured local queue for batching writes towards the QuarkDB backend.

## 1 Introduction

EOS is a distributed storage system being developed at CERN since 2011 [1], with the aim of addressing the demanding data needs of the LHC and its various experiments. It's built upon the XRootD [2] client-server framework, supports several data access protocols (XRootD, gsiftp, WebDAV, S3), and currently manages around 256 petabytes of raw disk space and 3.4 billion files across several instances.

Recently, the scope of EOS has been expanded to additionally serve as the backend for user home directories and a file syncing service, CERNBox [3], as the future replacement to the AFS [4] service provided by CERN IT.

Even though individual EOS instances routinely manage several hundreds of disk servers, users access the contents through a unified hierarchical namespace, giving the illusion of coherency, and simplifying file management. The namespace is exposed by the EOS head

[*]e-mail: georgios.bitzes@cern.ch
[**]e-mail: elvin.alin.sindrilaru@cern.ch
[***]e-mail: andreas.joachim.peters@cern.ch

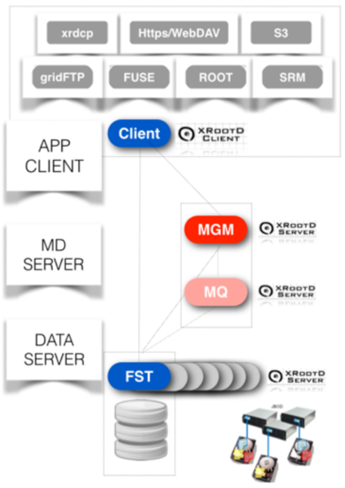

**Figure 1.** High-level view of the EOS architecture. FSTs are connected to and are responsible for the physical disks, while the MGM acts as the initial point of contact for external clients.

node (MGM), which stores the metadata of all files and directories stored on a particular instance.

The legacy namespace implementation keeps all metadata in-memory to permit low-latency access, backed by on-disk changelogs which are replayed during boot time. However, during the last few years, the number of files stored on EOS has been continuously growing. The legacy namespace subsystem has reached its scalability limits, presenting an urgent need for replacement. The limitations inherent to the old system mean that as the number of metadata entries grow, so does the memory consumption and reboot time of the MGM process.

We designed and implemented a new namespace subsystem, which is based on a key-value datastore backend, with in-memory caching on the MGM for frequently accessed entries. Therefore, the MGM now acts as a stateless write-through cache, with all metadata persisted on the backend. This paper is a continuation of our earlier work in 2017 [5] – the new namespace implementation has been improved further and recently entered production. The rest of this text is structured as follows:

- Section 2 presents a high-level architectural view of EOS, with a highlight on the changes brought by the introduction of QuarkDB.
- Section 3 offers more details on QuarkDB, including its design, development, and heavy focus on software testing.
- Section 4 describes our experience integrating the new namespace, and work done on performance tuning and latency hiding.
- Section 5 summarizes, and points to areas for improvement in the future.

## 2 Architectural overview

From a high-level view, the purpose of EOS is making large fleets of hard drives manageable, and presenting them to users in a convenient and coherent way, as well as enforcing access control. The architecture consists of the following components:

- The File Storage nodes (FSTs) are responsible for handling the physical storage — in a typical deployment, each FST manages several tens of hard drives.
- The Metadata Manager (MGM) is the initial point of contact for external clients, handles authentication and authorization, and redirects clients to the appropriate FSTs on both reading and writing.
- The Message Queue (MQ) handles inter-cluster communication between the MGM and the FSTs, delivering messages such as heartbeats and configuration changes.

The namespace subsystem is a critical part of the MGM, whose major purpose is translating logical paths to the physical locations where the files reside within the cluster. Among other things, the MGM is also responsible for background tasks, such as balancing, draining, detection of failed hard drives or corrupted files, and general management of the FSTs.

### 2.1 Legacy in-memory namespace implementation

The initial design of EOS foresaw each instance storing a rather limited amount of files, not exceeding a hundred million. Based on this assumption, a simple namespace implementation was chosen with a focus on performance and the following characteristics:

- The MGM holds the entire namespace contents in-memory to provide clients with fast access latency. No I/O or network requests to external services are necessary during path resolution, providing both low latency and high throughput under heavy load.
- Updates to the in-memory contents are backed by on-disk changelogs, which are periodically compacted. There is one changelog for file metadata, and one for directories.
- Whenever the MGM process needs to reboot, the entire in-memory contents are reconstructed by replaying the changelogs.

While the above scheme has proven to be simple and reliable, it comes with severe limitations once the number of files stored on an instance grows past a few hundred millions:

- Each metadata entry consumes around 1kb of memory on the MGM node, imposing a requirement for unreasonable amounts of physical RAM once the number of files grows past a certain number.
- Replaying the changelogs on reboot is a producedure which takes considerable amount of time, directly proportional to the number of files on an instance. This amplifies greatly the impact of crashes due to bugs on the MGM, as it can take a long time before the instance is ready to serve requests after a crash.

As an example, the most worrisome EOS instance at CERN which contains more than 600 million files requires an MGM node with 1 TB of physical RAM, while a restart of the server daemon takes around an hour to complete.

### 2.2 The need for a new namespace implementation

During the past few years, several ideas were considered for designing a better, more scalable namespace:

- Storing all metadata as objects in a highly available RADOS [6] instance, and communicating through librados. After a few short experiments, it was found to have unacceptably high write latency for our use-case.

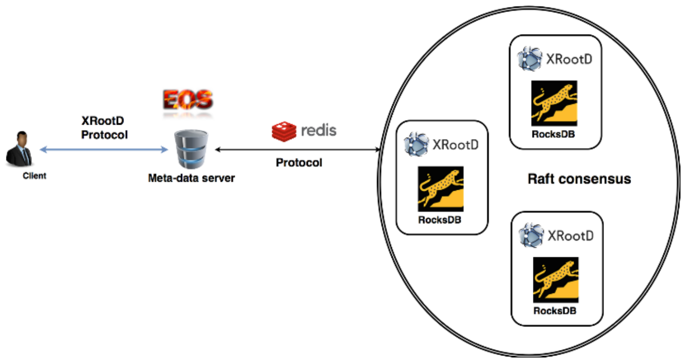

**Figure 2.** Metadata handling in the new namespace: QuarkDB acts as persistent storage for all file and directory metadata, with the MGM caching hot entries to limit amount of necessary network roundtrips, thus improving performance and responsiveness.

- Using a traditional relational database, for example MySQL or PostgreSQL. The concern was scalability and latency: RDBMS tend to perform poorly as the number of rows in a given table increase into the billions, and present early the need for complicated sharding schemes, which would significantly increase the complexity and operational cost of running an EOS instance. Additionally, having a performant, low-latency RDBMS would require a comparable amount of memory to the legacy namespace, and high-performance SSDs.
- Using a simple, non-relational datastore such as redis [7], well-known for its high performance and flexibility. What makes redis unsuitable for our use-case is the requirement that the entire dataset is memory-resident. The demand for physical RAM would still scale with the number of files, even if a small fraction of those are being accessed regularly.

The solution we chose is based on RocksDB [8], a highly-performant and stable embeddable key-value store. RocksDB implements a log-structured merge-tree (LSM) data structure, which retains good performance characteristics even when the number of key-value pairs stored within reaches the tens of billions, an important consideration in our use-case.

Since RocksDB is simply a library, we decided to develop a server around it which speaks the same protocol as redis. The simplicity of using redis from previous experiments made a good impression on us, so we decided to keep the same simple protocol and command-set, but substitute the storage backend with RocksDB, thus creating QuarkDB.

## 3 QuarkDB, a highly available datastore

Even though QuarkDB was designed specifically for storing the EOS namespace, and tailored to its needs, the functionality offered is generic. At its core, QuarkDB is a translation layer which converts redis commands into the appropriate RocksDB key-value transactions, using an internal encoding scheme.

QuarkDB does not currently implement the entire, extensive command-set of the official redis implementation, but only the small subset needed for EOS, thus limiting the complexity of the codebase.

An important consideration when designing QuarkDB has been high-availability. We aim in reducing any single points of failure existing in our software architecture, since EOS has become critical for data at CERN and unavailability has the potential of causing major disruption.

We decided that QuarkDB should be highly-available from the beginning, and implemented support for native quorum-consensus replication based on the Raft [9] algorithm. While a full description of Raft is outside the scope of this article, high-availability in QuarkDB works as follows:

- The QuarkDB cluster is able to tolerate losing some nodes without any impact on availability, provided that a majority (or *quorum*) remain online. In a typical deployment with 3 replica nodes, as long as at least 2 out of 3 nodes are alive and connected to each other, the cluster is fully operational.
- Replication is semi-synchronous, meaning that clients receive an acknowledgement to a write as soon as it has been replicated to a quorum of nodes.

### 3.1 Testing strategy

Ensuring correctness of QuarkDB and eliminating the likelihood of bugs remains a high priority when making changes to the codebase. Mild bugs have the potential of causing widespread disruption, given that QuarkDB is poised to become the index for hundreds of petabytes and billions of files at CERN. In addition, consensus algorithms are notoriously difficult to implement correctly, and are often plagued by subtle race conditions which are extremely hard to reproduce.

For the aforementioned reasons, QuarkDB is being developed in the spirit of Test Driven Development (TDD), which has resulted in a large suite of unit, functional, and stress tests, covering 91% of the codebase as of the time of writing. There are several end-to-end tests which actively attempt to trigger bugs and inconsistencies by randomly restarting cluster nodes under heavy read and write load.

The idea is to subject QuarkDB to far harsher conditions than it would typically encounter in a production environment. Of course, even such kind of testing does not guarantee the complete absence of bugs, but drives code quality up, and reduces the chance of regressions being able to survive undetected.

Experience in production has been very positive thus far, with QuarkDB routinely achieving continuous uptime of several months, usually interrupted just to update to a new version.

## 4 Integration work and performance optimizations

### 4.1 Impedance mismatch in the existing namespace API

Even though the namespace subsystem is cleanly delineated from the rest of the MGM code, with it being a separate loadable plugin, a large amount of integration work had to be completed before the new namespace was ready for production.

The major source of issues was the assumption made throughout the entire MGM code that namespace and path resolution requests were always low-latency, in-memory operations. This is no longer the case when using QuarkDB as a namespace backend, since an operation might now require one or more network roundtrips to complete. The difference in latency can be in the several orders of magnitude, even if all participating MGM and QuarkDB daemons are colocated in the same datacenter.

The above is exacerbated by the internal namespace API, used throughout the MGM. All namespace reads and writes occur under the global namespace read-write mutex – if an operation waits on a network roundtrip inside that lock, all other clients could stall for an unacceptable duration of time, negatively impacting the MGM responsiveness. The effects are especially visible during write operations, as only a single thread can acquire the mutex in write mode – multiple read operations could make progress in parallel.

While changing the API and using finer-grained locks would be great, the amount of effort and time needed for such a code refactoring would take too long, considering the urgency of deprecating the old namespace. Thus, the overarching goal has been to eliminate, where possible, having to pay network roundtrip penalties within the global namespace lock. Improving the namespace API is a planned future task.

### 4.2 In-memory write-through cache

The first method for combating network latency was introducing an in-memory metadata cache on the MGM. The namespace subsystem is capable of transparently fetching requested metadata either from its local cache depending on availability, or from the QuarkDB backend, substantially reducing the number of roundtrips paid for frequently accessed metadata.

Only a small fraction of total metadata entries need to be accessed frequently. The cache makes most read operations entirely local, providing equivalent latency as the legacy in-memory namespace. Entries are evicted from the cache on a Least-Recently-Used basis.

### 4.3 Persistent, backpressured queue for writes

While being extremely useful for read operations, caching does little to help writes in this case. There is a dilemma when issuing write requests to QuarkDB: Should we wait for an acknowledgement, or “fire and forget” assuming all requests succeed?

The first choice is certainly the safest, but imposes a severe performance penalty: Any namespace modification would involve waiting on a network request while holding the global namespace lock, preventing other clients from interacting with the namespace. Given that EOS instances typically serve thousands of clients at any given moment, such an approach would quickly make an instance unusable, limiting metadata modification throughput on roundtrip latency between the MGM and QuarkDB.

On the other hand, “fire and forget” avoids paying any network penalty, but is obviously unsafe as writes could be silently lost even if an end-user receives acknolwedgement for some operation.

We chose an approach which is the best of both worlds: All metadata modifications are recorded into a local, persistent queue, which is replayed towards QuarkDB transparently in the background. The queue keeps track of which items have been acknowledged, and are thus safe to discard. In the event of network instabilities, requests will be retried until they succeed, while ensuring they are issued in the correct order.

This way, issuing acknolwedgements to end-user clients before receiving confirmation from QuarkDB becomes safe: All unconfirmed requests will be retried even if the MGM crashes, ensuring consistency. After a crash, the MGM will not process any end-user requests until any potential, unconfirmed operations from the previous run have been applied on QuarkDB. Therefore, a namespace modification will never wait on a network request. It’s enough to record into the local queue, and appropriately update the contents of the cache.

The above creates the possibility of the queue growing out of control in case of widespread network disruption, or QuarkDB unavailability. To avoid such a situation, backpressure built into the queue stops the addition of further requests once the number of pending ones reaches a certain, configurable number.

### 4.4 Prefetching entries outside the namespace lock

Consider for a moment what happens when listing the contents of a directory with a million files. The MGM acquires the global namespace lock, and synchronously retrieves one-by-one all metadata entries contained within. Provided the contents are not in the cache, the above would cost a million *network roundtrips* between the MGM and QuarkDB, essentially making the instance unavailable for several minutes.

The above limitation arises, once again, due to the assumption that all metadata is local, and available in-memory. We implemented the following mitigations:

- Addition of an asynchronous version for many operations in the internal namespace API. Instead of waiting for a response from the server, a future object is returned and the request is processed asynchronously in the background. This way, the calling thread is free to pipeline metadata requests, and only really pay the network latency penalty once.

  In our example, a million requests will still happen, but those will be streamed, not "ping-ponged". The overall latency for the completion of all one million requests drops drastically.

- Ability to prefetch metadata into the cache outside the namespace lock. Even before acquiring the lock, it's already clear which metadata entries will be needed to complete an operation: We instruct the prefetcher class to ensure all necessary entries are present in the cache. Once we acquire the lock, it's guaranteed no network requests will occur, as all metadata needed is in-memory already. This drastically reduces the amount of time any client has to hold onto the global namespace lock.

  In our example, this ensures all million metadata entries are fetched while not holding the namespace lock – other clients are free to use the instance in parallel.

### 4.5 Cache-bypass for streaming operations

Certain operations do not particularly benefit from the use of in-memory metadata caching, such as those scanning through a large number of metadata entries. A notable example is the "find" command, which recursively traverses a given path and reports all files contained within.

In such case, network roundtrip latency is amortized by pipelining metadata requests – caching will not make much of a difference. Moreover, the scanned contents are not likely to be among the most frequently used entries, thus thrashing the memory-limited cache and potentially evicting frequently accessed entries. Therefore, bulk operations such as "find" bypass the cache and the global namespace mutex.

## 5 Conclusions and future work

We set out to improve the scalability shortcomings in the original design of the EOS namespace, and implemented a highly available datastore to serve as the metadata backend. The overall system was optimized to offer comparable levels of performance and latency as the in-memory approach. We believe the new namespace implementation is capable of offering the next order of magnitude of scaling for EOS, ready to meet the data needs of the LHC experiments and CERN as a whole:

- Booting time of the MGM, which required *1 hour* for instances with 600 million files, now takes less than 30 seconds.

- Our test instance on the new namespace has been tested with up to 4.6 billion files, which is more than all other EOS instances combined at the time of writing. Even at such size, namespace operations remain fast and responsive, with boot time not exceeding a few seconds.
- Memory consumption is mostly dictated by the size of the cache, which is configurable, and not by the total size of the entire namespace. While we did not yet need to perform extensive evaluations on the optimal cache size, experience shows only a small fraction of the namespace is accessed frequently, and thus benefits from caching.

Future work could improve on the design in several areas:

- Ability to use additional MGMs as read-only secondaries. This would help in absorbing read load for use-cases which tolerate eventual namespace consistency.
- Gradual removal of the global namespace lock, introduction of finer-grained locks in its stead.
- Improvement and rationalization of the namespace API.
- QuarkDB could be made to additionally serve as a highly available message queue, replacing the current one and removing an additional architectural single point of failure in EOS.

## References

[1] Peters, Andreas J., Janyst, L.: Exabyte scale storage at CERN. Journal of Physics: Conference Series. Vol. 331. No. 5. IOP Publishing (2011)

[2] Dorigo, Alvise, et al.: XROOTD-A Highly scalable architecture for data access. WSEAS Transactions on Computers 1.4.3 (2005)

[3] Mascetti, L., et al.: CERNBox+ EOS: end-user storage for science. Journal of Physics: Conference Series. Vol. 664. No. 6. IOP Publishing (2015)

[4] Howard, John H.: An overview of the andrew file system. Carnegie Mellon University, Information Technology Center (1988)

[5] Peters, Andreas J and Sindrilaru, Elvin A and Bitzes, Georgios: Scaling the EOS namespace. International Conference on High Performance Computing, Springer (2017)

[6] Weil, Sage A., et al: Rados: a scalable, reliable storage service for petabyte-scale storage clusters. Proceedings of the 2nd international workshop on Petascale data storage: held in conjunction with Supercomputing'07, ACM (2007)

[7] Sanfilippo, Salvatore, and Pieter Noordhuis.: Redis (2009)

[8] Borthakur, Dhruba.: Under the Hood: Building and open-sourcing RocksDB. Facebook Engineering Notes (2013)

[9] Ongaro, Diego, and John K. Ousterhout.: In Search of an Understandable Consensus Algorithm. USENIX Annual Technical Conference (2014)