Scaling the EOS Namespace

Andreas J. Peters, Elvin A. Sindrilaru, and Georgios Bitzes(✉)

CERN IT, Geneva, Switzerland
Georgios.bitzes@cern.ch

Abstract. EOS is the distributed storage system being developed at CERN with the aim of fulfilling a wide range of data storage needs, ranging from physics data to user home directories. Being in production since 2011, EOS currently manages around 224 petabytes of disk space and 1.4 billion files across several instances.

Even though individual EOS instances routinely manage hundreds of disk servers, users access the contents through a single, unified namespace which is exposed by the head node (MGM), and contains the metadata of all files stored on that instance.

The legacy implementation keeps the entire namespace in-memory. Modifications are appended to a persistent, on-disk changelog; this way, the in-memory contents can be reconstructed after every reboot by replaying the changelog.

While this solution has proven reliable and effective, we are quickly approaching the limits of its scalability. In this paper, we present our new implementation which is currently in testing. We have designed and implemented QuarkDB, a highly available, strongly consistent distributed database which exposes a subset of the redis command set, and serves as the namespace storage backend.

Using this design, the MGM now acts as a stateless write-through cache, with all metadata persisted in QuarkDB. Scalability is achieved by having multiple MGMs, each assigned to a subtree of the namespace, with clients being automatically redirected to the appropriate one.

1 Introduction

The EOS project [1] started in 2011 to fulfill the data storage needs of CERN, and in particular storing and making available the physics data produced by the LHC experiments. Being developed and in production since 2011, EOS is built upon the XRootD [8] client-server framework, supports several data access protocols (XRootD, gsiftp, WebDAV, S3), and currently manages around 224 petabytes of disk space with 1.4 billion files across several instances.

Recently, the scope of EOS has 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 [2] service provided by CERN IT.

The gradual growth in the total number of files stored on EOS has revealed certain scalability limitations in the original design of the namespace subsystem. In this paper, we describe the legacy implementation and its shortcomings, discuss our new implementation based on a separate highly-available metadata store exposing a redis-like interface, and present some preliminary performance measurements.

2 Architectural Overview

An EOS instance is composed of several distinct 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 component this paper focuses on is the MGM, and in particular its namespace subsystem which stores all file metadata and among other things is responsible for translating logical paths to the physical locations where the files reside within the cluster.

Example 1. Sample namespace entry representing `/eos/somedir/filename`.
Inode number: `134563`
Name: `filename`
Parent directory inode: `1234`, meaning `eos/somedir`
Size: `19183 bytes`
File layout: `2 replicas`
Physical replica 1: `Filesystem #23`, meaning `fst-1.cern.ch:/mnt34/`
Physical replica 2: `Filesystem #45`, meaning `fst-2.cern.ch:/mnt11/`
Checksum: `md5-567c100888518c1163b3462993de7d47`

In the process of developing a better namespace implementation, and for the sake of being able to run experiments and measurements easily, we moved the namespace subsystem into a separate plugin. The rest of the MGM code uses a standard interface to talk to it, thus facilitating an easy way of replacing it without affecting the rest of the code.

Making the namespace more scalable involved some changes to the above architecture; namely, the addition of a new highly-available database component, as well as enabling the use of multiple MGMs for load-balancing. These changes are described in more detail in later sections.

3 The Legacy In-Memory Namespace

One of the primary goals of EOS from the beginning has been to deliver good and consistent performance. This includes being able to fully exploit the underlying hardware in terms of I/O and network performance of the FSTs, as well as perform low-latency metadata operations on the MGM.

The initial design includes a namespace implementation where all metadata lives in-memory on the MGM, and is persisted on-disk in the form of a changelog. In more detail:

– During MGM boot, the entire namespace is reconstructed in-memory by replaying the on-disk changelog.
– File lookups require no I/O operations, as all metadata is retrieved from memory. Entries are stored in a dense hash map (provided by the Google SparseHash library), keyed by the inode number, and consume approximately 1kb of memory each.
– For metadata updates, the memory contents are modified and an entry is appended to the changelog, which is fsynced periodically.
– A background thread compacts the changelog on regular intervals, thus purging out-of-date entries which have been superseded by newer ones. This process ensures the size of the changelog remains under control, and stays proportional to the total size of the instance, and not to the entire history of operations on it.

While this solution has proven reliable and effective, it has several important limitations:

– The total size to store the entire namespace of an instance cannot exceed the physical RAM available on the head node, since everything is stored in-memory.
– Replaying the changelog after a reboot can take a long time, upwards to one hour for some of our larger instances.
– The use of a single head node represents a scalability bottleneck, as well as a single point of failure.

The effects of long boot time can however be mitigated by employing optional active-passive replication, through which is possible to have a slave MGM on hot standby that can be manually promoted to master, in case the current one fails. During normal operation, the master MGM performs continuous one-way synchronization of its changelog towards any configured slaves.

4 The New, Scalable Namespace

One of the more promising ideas for replacing the legacy namespace has been to store all metadata on a redis [4] instance, a datastore well-known for its high performance and flexibility. We implemented a namespace plugin which used redis for metadata persistence — what made it unsuitable for our use-case was the need to accommodate very large datasets. Redis poses the requirement that the total data stored is smaller than the physical RAM of the machine hosting it.

A different idea has been to use an embeddable, on-disk key-value store such as RocksDB [5] directly on the MGM. This solves both major issues of the legacy design:

– No need for unreasonable amounts of RAM on the MGM, since the contents can be retrieved from disk when needed. This is certainly much slower than a memory lookup, but we can mitigate the effects by adding a caching layer for hot entries.
– Initialization time is nearly instantaneous even for datasets spanning several terabytes.

While such a design would solve all immediate problems we faced, an important downside remained. The MGM would still represent a scalability bottleneck and single point of failure, and losing it would result in the entire cluster becoming unavailable, requiring manual intervention.

Our final design combines the two ideas above. We implemented a highly available distributed datastore, QuarkDB, which supports and exposes a small subset of the redis command set, using RocksDB as the storage backend and translating all redis commands into equivalent RocksDB key-value transactions.

The MGM encodes all metadata in a redis-compatible format using a combination of `STRING`, `HASH`, and `SET` redis commands, serialized with protocol buffers [6].

To minimize the impact of an extra network roundtrip between the MGM and QuarkDB for most metadata operations, the MGM caches hot entries locally under a Least-Recently-Used eviction policy. As we shall see later in the measurements, there is no performance loss compared to the in-memory implementation for cached read operations, which is usually the dominating access pattern in terms of frequency.

Using the above architecture, it now becomes possible to spread the client load by employing multiple MGMs, having each responsible for a subtree of the namespace. In this regard, each MGM essentially acts as a write-through cache for all metadata which is persisted on QuarkDB. To simplify the management and deployment of multiple MGMs, the configuration setup moves from being stored in files locally on an MGM, to being centrally managed in QuarkDB.

5 Designing QuarkDB, a Highly Available Datastore

5.1 Choosing a Storage Backend and Access Protocol

As mentioned earlier, the goal of QuarkDB is to serve as the namespace metadata backend for EOS. In order to avoid the time-consuming task of re-implementing the low-level details of a database, we leverage the RocksDB library, a highly-performant embeddable datastore based on the log-structured merge-tree data structure. We made this choice based on the fact that RocksDB is open source, actively maintained, and used across several important projects already.

We chose the Redis Serialization Protocol, the same one used in the official redis server, based on its simplicity of use and implementation and the fact that there already exist tools compatible with it. (e.g. `redis-cli`, `redis-benchmark`).

5.2 Redis Data Structures Stored in RocksDB

The next step was to decide on a way to translate between redis operations and RocksDB key-value transactions. We implemented the following simple encoding scheme:

- Each redis key is associated to a key descriptor stored in RocksDB, which is its name prefixed by the letter "d", containing its type (whether a `STRING`, a `HASH`, or `SET`) and size. This allows to detect errors, for example when the user attempts to use an existing `HASH` like a `SET`.
- The contents of a `STRING` are stored in a key containing its name prefixed by the letter "a".
- Each element in a `HASH` or `SET` is stored in its own RocksDB key: the key name concatenated with the symbol "#" plus the element name, prefixed by the letter "b" or "c", depending on whether it's a `HASH` or a `SET`.

To make things more clear, the following example shows the steps performed during a lookup using the `HGET` redis command.

1. The client issues `HGET mykey myelement`. In this context, the client is an EOS MGM.
2. The key descriptor for `mykey` is retrieved by looking up `dmykey` in RocksDB.
3. If the key descriptor does not exist, an empty reply is returned. If the key descriptor is not associated to a `HASH` but some other data structure, an error is returned.
4. A lookup for `bmykey#myelement` is done in RocksDB, and the contents are returned to the client. If the lookup finds no result, it means there's no `myelement` within `mykey`, and an empty reply is returned.

Listing the contents of a container works in a similar way – after retrieving the key descriptor, a range scan is performed with the appropriate prefix, which returns all elements in the corresponding hash or set.

5.3 Introducing High Availability

To prevent QuarkDB from becoming the single point of failure, we implemented native quorum-consensus replication based on the Raft [7] consensus algorithm:

- 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.

Raft works by essentially replaying a series of operations towards a database (called the *state machine* in Raft terminology), ensuring they are identical across all nodes and are applied with an identical order. In our case, the state machine is represented by the class which translates redis commands into RocksDB transactions, and is mostly separate from the consensus logic – this way, QuarkDB can be run in standalone mode as well, without having to pay for the high overhead of consensus in case high availability is not needed.

5.4 Ensuring Correctness

Since QuarkDB is to become a critical component of every EOS instance, ensuring correctness has been of paramount importance, especially given that implementing distributed consensus correctly can be quite tricky. QuarkDB is being written following the spirit of Test Driven Development (TDD), which has resulted in a large suite of unit, functional, and stress tests. This is in addition to several internal assertions that detect possible inconsistencies between the nodes, so as to further reduce the risk of the replicas getting out of sync and opting to crash early instead.

6 Preliminary Measurements

6.1 Test Setup

We used three identical bare-metal machines running CERN CentOS 7 with dual-socket Intel Xeon E5-2650 v2 at 2.60 GHz, providing a total of 32 cores on each machine.

6.2 QuarkDB Performance

Performance depends heavily on whether pipelining is used, that is if the client sends multiple commands at the same time without waiting for an acknowledgement of the previous ones. This amortizes the roundtrip latency and allows for certain optimizations, such as batching several responses to the client using a single `write()` system call.

All measurements were taken using the `redis-benchmark` tool with keyspacelen of 10 million – this ensures the load is spread over a large set of keys.

Ping Throughput. `PING` is a redis command to which the server simply replies with `PONG`. This test is useful to verify that the machinery handling network sockets and threads is efficient: QuarkDB is able to reach a peak of 1.6 million pings per second (Fig. 1).

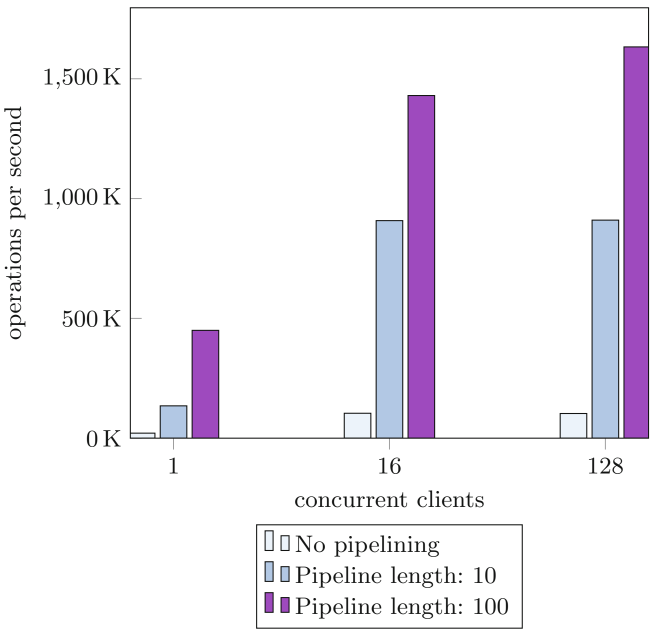

Fig. 1. QuarkDB PING throughput

Standalone Mode. Write performance reaches a peak of around 105 thousand operations per second using the `SET` command (Fig. 2). Each write was 200 bytes in size. Read performance reaches a peak of 320 thousand operations per second (Fig. 3).

Replicated Mode with Raft Consensus. Write performance reaches a peak of 9000 operations per second – the major limiting factor here is the Raft journal into which all write operations must be serialized (Fig. 4). Read performance is identical as in standalone mode (Fig. 3), since in our implementation reads go directly to the state machine, without passing through the raft journal.

6.3 EOS Measurements

There is currently a major limitation in how EOS handles writes into the namespace: certain locks prevent multiple clients from performing concurrent updates, resulting in low parallelism and limited use of pipelining towards QuarkDB.

We are in the process of fixing this limitation – even so, the performance achieved in replicated mode is still several times higher than what we currently see in production (20 Hz file creation rate). The goal is to eventually reach and surpass the rates achieved by the in-memory namespace.

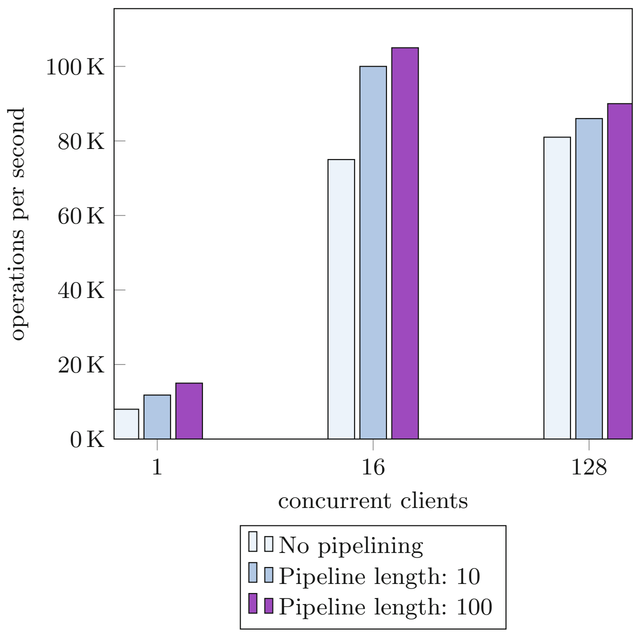

Fig. 2. QuarkDB write performance in standalone mode, SET command, 200 bytes

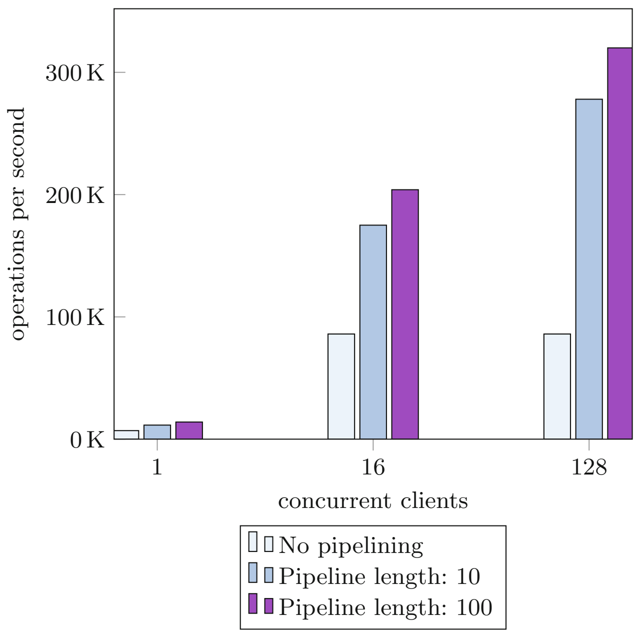

Fig. 3. QuarkDB read performance, GET command

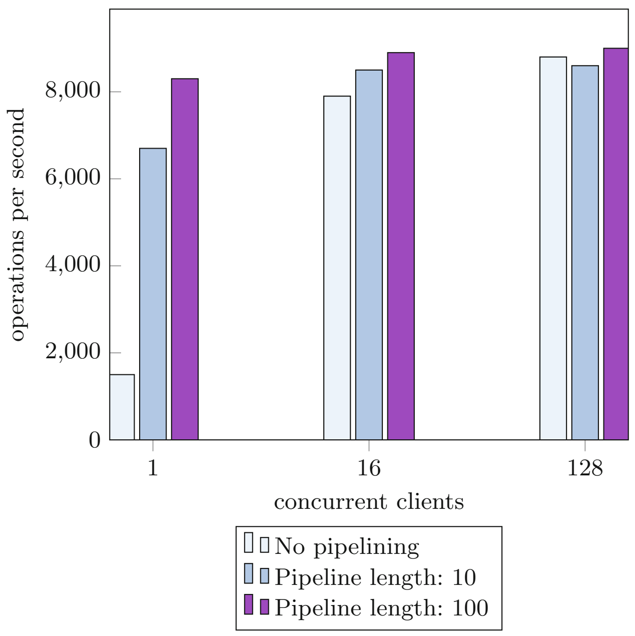

Fig. 4. QuarkDB write performance in Raft mode, SET command, 200 bytes

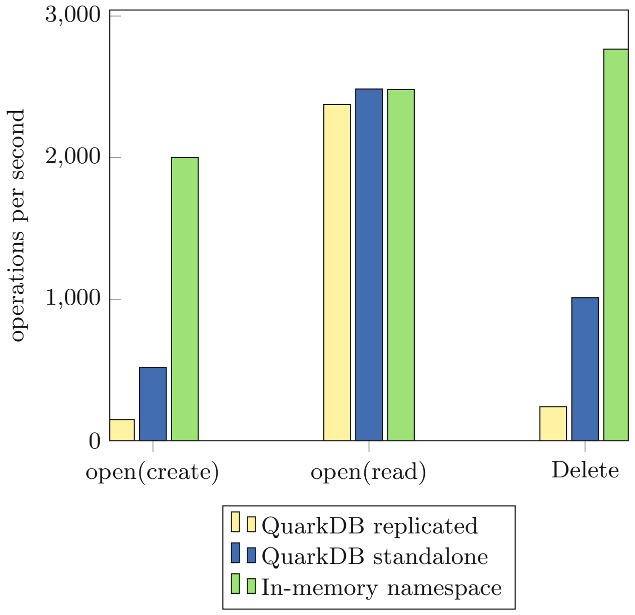

Fig. 5. End-to-end operations towards EOS

Measurements were taken using our custom load-testing tool through the XRootD file access protocol. It's important to note that operations such as file creation and deletion result in several key writes towards QuarkDB, which is why these measurements are presented separately (Fig. 5).

7 Conclusions and Future Work

We set out to improve the scalability shortcomings in the original design of the EOS namespace. We implemented a highly available metadata server component based on the redis serialization protocol, the RocksDB embeddable key-value store, and the Raft consensus algorithm.

Our measurements show that 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. Future work could improve on the design in several areas:

– Implementing automatic sharding in QuarkDB to overcome the inherent bottleneck imposed by the serial Raft log.
– Adding automatic and transparent failover to the MGM layer. An MGM failure could be made detectable by the rest, thus transferring its responsibilities and assigned namespace subtree to a different node, automatically and with minimal impact on availability.
– QuarkDB could be made to additionally serve as a highly available message queue, replacing the current one and removing one of the few remaining single points of failure in EOS.

References

1. Peters, A.J., Janyst, L.: Exabyte scale storage at CERN. J. Phys. Conf. Ser. **331**(5), 052015 (2011). IOP Publishing
2. Howard, J.H.: An overview of the Andrew file system. Carnegie Mellon University, Information Technology Center (1988)
3. Mascetti, L., et al.: CERNBox+ EOS: end-user storage for science. J. Phys. Conf. Ser. **664**(6), 062037 (2015). IOP Publishing
4. Sanfilippo, S., Noordhuis, P.: Redis (2009)
5. Borthakur, D.: Under the Hood: Building and Open-Sourcing RocksDB. Facebook Engineering Notes (2013)
6. Varda, K.: Protocol Buffers. Google Open Source Blog (2008)
7. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: USENIX Annual Technical Conference (2014)
8. Dorigo, A., et al.: XROOTD-A highly scalable architecture for data access. WSEAS Trans. Comput. **1**(4.3), 348–353 (2005)