EOS developments

Elvin A Sindrilaru, Andreas J Peters, Geoffray M Adde and Dirk Duellmann
CERN IT, Geneva, Switzerland
E-mail: elvin.alin.sindrilaru@cern.ch andreas.joachim.peters@cern.ch
geoffray.adde@cern.ch dirk.duellmann@cern.ch

Abstract.
CERN has been developing and operating EOS as a disk storage solution successfully for over 6 years. The CERN deployment provides 135 PB and stores 1.2 billion replicas distributed over two computer centres. Deployment includes four LHC instances, a shared instance for smaller experiments and since last year an instance for individual user data as well. The user instance represents the backbone of the CERNBOX service for file sharing. New use cases like synchronisation and sharing, the planned migration to reduce AFS usage at CERN and the continuous growth has brought EOS to new challenges. Recent developments include the integration and evaluation of various technologies to do the transition from a single active in-memory namespace to a scale-out implementation distributed over many meta-data servers. The new architecture aims to separate the data from the application logic and user interface code, thus providing flexibility and scalability to the namespace component.

Another important goal is to provide EOS as a CERN-wide mounted filesystem with strong authentication making it a single storage repository accessible via various services and front-ends (/eos initiative). This required new developments in the security infrastructure of the EOS FUSE implementation. Furthermore, there were a series of improvements targeting the end-user experience like tighter consistency and latency optimisations.

In collaboration with Seagate as Openlab partner, EOS has a complete integration of OpenKinetic object drive cluster as a high-throughput, high-availability, low-cost storage solution. This contribution will discuss these three main development projects and present new performance metrics.

1. Introduction
The EOS[1] project started in 2010 with the goal of providing a disk-only solution for analysis purposes that would guarantee low-latency access to physics data. The architecture of EOS is built around the XRootD[3] framework and profits from the plugin-based design which enabled it to adapt to new requirements and transform itself into the mature systems it is today. The number of EOS instances has been growing steadily throughout the years by handling the data of all major LHC experiments as well as other smaller non-LHC experiments at CERN. Besides the EOS instances which are specifically targeting physics data, there is also an instance storing individual user data acting as a personal storage space for users at CERN. The USER instance represents the backbone of the CERNBOX[2] service for collaboration and file sharing. This represents a completely new type of service that is provided to the CERN users which combines traditional ways of accessing the data using well-known protocols like XRootD, HTTP(S) with
modern synchronization protocols similar to Dropbox.

The excellent performance delivered by the LHC accelerators coupled with the wide adoption of EOS among experiments and users alike, has led to a considerable growth in storage capacity and number of files/directories saved in the namespace. Currently there are around 135 PB of raw storage capacity managed by EOS with the bulk of the increase coming from 2016. In terms of number of entries in the namespace, the EOS USER instance has seen the most dramatic growth over the last year. At this point, the CERNBOX service stores around 170 million entries in the namespace which is roughly the same size as the EOS ATLAS instance. Part of the successful adoption of EOS at CERN was the seamless integration with the existing tools and environments that the users were familiar with. From this perspective, the EOS FUSE module played an important role in making the EOS service popular also with users outside the physics community.

In order to accommodate various use-cases as well as the consistent growth while the service becomes more popular, the EOS architecture needs to evolve. In this paper, we present the main areas in which we are doing active development together with the underlying technical reasons for which a revisit of the initial design decisions is necessary. Most of the challenges that we are about to discuss in the following sections are a direct consequence of the organic growth of the system. In the first part of the paper, we present a new namespace architecture based on a distributed key-value store that aims to overcome limitations of the current in-memory namespace implementation. The second part discusses the EOS FUSE module and the steps taken in order to improve end-user experience by increasing the I/O performance. Last but not least, we present the integration of Kinetic drives which are a novel storage solution into the existing EOS architecture with minimal modifications to the underlying code base.

2. Namespace back-end redesign

The EOS architecture follows the traditional storage model where responsibility between the data and meta-data is split between two types of nodes. The MGM or manager node is responsible for storing the meta-data and the FST (or storage nodes) take care of the actual data. When a client wants to interact with the system, be it for reading or for writing, it needs to first contact the MGM and then gets redirected to one of the FSTs. For the system to be responsive when accessed by many clients, the MGM needs to be fast in satisfying all these requests.

The current namespace implementation has a set of properties that make it particularly suitable for this architecture. The namespace library is single-threaded, written in C++ and provides the API to deal with hierarchical collections of files and directories. Apart from this, when the MGM daemon is running the entire namespace is loaded in memory, therefore providing excellent performance for both look-up and insertion operations. The building blocks of the namespace are the file and container (directory) entries which are enough to fully reconstruct the hierarchical structure. Based on these entries, there are several types of views that can be constructed e.g. hierarchical view, filesystem view, quota view etc. Any modification to a file or directory entry is saved in a changelog file on disk. Therefore, when the namespace is booted it reads the changelog and constructs the file/directory objects in memory together with the corresponding views.

Although this implementation has proven to be an excellent choice so far, it also has some drawbacks. In case of a large namespace, the time to boot-up all the entries in memory can be significant. Moreover, the machine that boots the namespace needs to have enough RAM to store all the information. For example, one of our largest production instances has around 400 million file entries and this translates into roughly 390 GB of RAM. Besides the sheer size in memory there is also the problem of the boot-up time. For the ALICE instance on the recent type of hardware this can take up to 50 minutes.

In order to address these limitations we set out to redesign the namespace part of EOS
while trying to maintain some highly desirable features like fast access, strong consistency and low risk of data loss. The first step was to abstract a namespace interface given the current implementation. This was useful since it allowed us to easily prototype and experiment with different namespace implementations without touching other parts of the EOS code base. We keep full compatibility with the current namespace implementation which now is separated into its own library. The general idea behind the new namespace implementation is to separate the data from the application logic and user interface, thus providing scalability and flexibility. To achieve this we need a distributed key-value storage back-end that can hold the file/directory entries but also information about the different views.

Our first prototype used Redis\cite{4} as key-value back-end and leveraged some of the basic features exposed by Redis like hashes and sets. Redis provides a rich set of data structures but we decided to use only of subset of them. The new namespace implementation Figure 1 is available as a separate library \texttt{libEosOnRedis.so}. Using this approach we managed to reduce the boot-up time of the MGM daemon since now all the data is actually stored in Redis. Unfortunately, this does not solve also the memory size limitation. Redis is able to keep a changelog-like structure for the data it stores, but once it brings a piece of data in memory, it keeps it throughout the lifetime of the process. This small prototype helped us validate the new namespace interface and provided valuable insights for the later stages of development.

At this point we looked into RocksDB\cite{5} which is a persistent key-value store with a particular feature that allows it to move data between RAM and disk depending on the requirements. On the other hand, RocksDB has no knowledge of maps and sets therefore this functionality needs to be implemented on top of it. Therefore, we decided to integrate RocksDB with XRootD while at the same time providing a Redis protocol implementation for XRootD. The protocol implementation supports only a subset of the original Redis protocol, namely the commands concerning map and set objects. The new back-end XRootD daemon supporting the Redis protocol and using RocksDB underneath for storing key-value pairs is called QuarkDB\cite{6}.

Given this new setup, we are now able to address all the limitations of the in-memory namespace implementation.

The fact that EOS has become an important service both within the IT department as well as among the experiments, means that system should be able to handle failures of any component and remain available as much as possible. In this new architecture of the namespace, the back-end becomes a single point of failure. To avoid such an undesirable property, we decided to ensure high-availability by using the \textbf{Raft consensus algorithm}\cite{7}. The new deployment model, consists of several machines begin part of the back-end cluster, each having a copy of the original data as shown in Figure 2. At any point in time, there can be only one master node which handles all the incoming requests and also replicates all the writes to at least the majority of nodes in the cluster. By using this model, we can ensure that if the master node dies, then one of the other nodes in the cluster will be elected as the master and will continue serving requests without any data loss.

The first benchmark tests using the new system show, as expected, a considerable drop in
3. EOS FUSE improvements

Ever since it was first presented to the users, the EOS FUSE module has become a major feature of the EOS service due to its familiar interface and ease of use. Lately, the FUSE module has undergone some important changes with the undeclared goal of becoming a viable alternative to the current AFS system which is in place at CERN since decades. What this means, in practical terms, is faster access to the data and better support for some operations that users normally do on a traditional filesystem e.g archiving, compiling code, doing a git checkout of a project etc.

The first such improvement represents the caching of meta-data on the client side by using the Linux Kernel buffer cache. But caching has its limitations, since the client would still need to contact the MGM node for update modifications and also while filling the cache. For this reason, some operations that normally took several round-trips to the MGM node were heavily optimised by moving to bulk requests and responses. One such example is the directory listing which now is done using bulk meta-data queries rather than the usual sequence of opendir, read entry and closedir.

Another important change, more on the deployment side this time, is the move from individual FUSE mounts to multi-user FUSE mounts (Figure 3). The new deployment model has a series of benefits like better resource allocation, better cache utilization and lower operational costs. The multi-user EOS FUSE mount leverages the XRootD capability of using different authentication protocols and credentials for each user and operation. Therefore, the EOS FUSE mount can now handle several users at the same time, each of them having a dedicated and authenticated connection to the EOS instance. The user has the possibility to choose what authentication protocol (Kerberos 5 or Grid Certificate) as well as what credentials he would like to use when interacting with the FUSE mount. Apart from this, the integration with automount represents another step towards improving the user experience and making it as easy as possible to use the FUSE mount.
Reducing the communication between the FUSE mount and the EOS instance to the minimum, represents the only way of achieving high performance when doing interactive operations. The next area to undergo performance tuning is the interaction with the actual dataserver in EOS. The XRootD protocol heavily relies on the concept of redirection to guide the client to the data, but this way of discovering and placing data is at odds with some common operations performed by users. When opening a file for reading or writing the client goes to the MGM and then is redirected to the FST. But many applications open files without doing any I/O operation immediately after the open. Therefore, the application needs to pay the penalty of the redirection for each open operation even if it uses the file or not.

To compensate for this, we introduce the concept of lazy-open Figure 4. What this means in practice, is that we separate the meta-data and the data part when a client interacts with the EOS service. When requested to do an open, the FUSE mount will do the open only at the MGM node making sure the meta-data information is present. After a successful open at the MGM, the FUSE client will report success to the user without actually opening the file on the dataserver. This operation now happens asynchronously and when an actual I/O request comes from the user, chances are the file open on the dataserver is already completed. Therefore, in the best case scenario, the asynchronous open on the dataserver completely hides the latency seen by the client. In the worst case, we are no better off than we were before when using redirection.

To limit the impact of many small operations to the backend we implemented a write-back cache in the FUSE layer that is able to aggregate small write requests and send them asynchronously once a certain amount of data has accumulated. Depending on the type of
Figure 4. EOS FUSE lazy-open mechanism

application and its interaction with the storage layer, this optimisation can have a considerable impact on the overall performance observed by the user. For example, the time to compile the XRootD project on a EOS FUSE mount has dropped from around 900 to 33 seconds. This is comparable with the performance achieved on a local file system with warm cache that takes around 22 seconds.

4. Kinetic Storage integration

Due to the ever increasing demand as well as requirements for better fault-tolerance and availability, the storage industry needs to continuously adapt and experiment with new concepts and designs. One such initiative is the Kinetic Open Storage Project\cite{8} that proposes a new way of storing and accessing the data. The Kinetic drive architecture provides a key-value interface with Ethernet connectivity and tries to be a drop-in storage solution for current datacentres.

The key advantage from the operational point of view is that if one of the drivers becomes unavailable then the technician only needs to replace that drive without affecting the performance or availability of the entire cluster. Moreover, a number of functionalities that normally are implemented in higher levels of the software stack, can now be directly implemented at the protocol or cluster level. For example, support for checksums is embedded into the protocol and together with compression and replication capabilities represent a feature rich offering for a storage system. Last but not least, the simple abstract interface that it exposes to users makes it simple to develop applications on top of it and also easy to reason about the behaviour of the system.

Besides the technical specifications, the Kinetic Open Storage Project is attractive also for the reduced cost of ownership that it promises. Given all these nice features and the excellent collaboration that we have with Seagate within the CERN Openlab Programme we decided to add support for Kinetic clusters to EOS. There are two ways of attaching a Kinetic cluster behind EOS. The easier one that requires only minimal modifications to the code base, connects each Kinetic cluster to only one individual dataserver (FST node in EOS terminology). To achieve this, we added a new I/O plugin called KineticIO that uses the Kinetic protocol to send and receive data. In this configuration, the rest of the EOS service is completely agnostic to the underlying I/O access type.

While this was relatively easy to implement and deploy, it has the disadvantage that the data coming from the local storage attached to the dataserver and the data coming from the Kinetic cluster compete for the same network bandwidth. On top of this, if the network connection to the dataserver is broken, then the entire data stored in the Kinetic cluster also becomes
 unavailable. In order to address such concerns, we present a different architecture in which each Kinetic cluster is connected to several dataservers. This new EOS Kinetic multi-path configuration (Figure 5) requires more subtle modifications to the scheduling mechanism as well as extra logic to deal with concurrency resolution. All these have been integrated in the new Kinetic-aware scheduling component that takes care of cache utilization and load balancing.

Being able to incorporate Kinetic clusters into the EOS architecture, means that our current dataservers can provide a higher storage capacity without any additional storage purchase. In this model the dataservers become gateways to the Kinetic cluster and by using the multi-path access we manage to fully utilize the combined dataserver network capacity. We currently have one Kinetic cluster of 1PB deployed on a testing instance and we expect a new delivery this year consisting of a Kinetic system with 2nd generation Seagate drives.

5. Conclusions
In recent years the EOS system has seen considerable growth both in terms of storage size but also in the number of instances deployed at CERN and other institutes. Keeping up with user demands as well as scalability challenges required some core changes to the namespace architecture and also important performance tuning on the client side. The new namespace uses a scalable distributed key-value store and tries to efficiently utilise caching techniques to hide latency. Using the persistent key-value store implemented in-house on top of proved technology such as RocksDB and the Redis protocol provides us with the necessary tools to easily customize the behaviour of the system to our particular needs as well as overcoming the limitations of the previous implementation.

The continuous improvements done to the FUSE module represent a proof of our commitment to address the specific needs of the physics community by seamlessly integrating EOS with the environment users are most familiar with. Last but not least, we try to experiment and evaluate different storage technologies that are emerging on the market in order to provide an easy to manage, cost-effective and performant system.
6. Acknowledgements
We thank all our collaborators for their valuable feedback and particularly Paul H. Lensing from Seagate Technologies for his contribution concerning Kinetic integration with EOS.

References