Abstract. EOS is an open source distributed filesystem developed and used mainly at CERN. It provides low latency, high availability, strong authentication, multiple replication schemas as well as multiple access protocols and features. Deployment and operations remain simple and EOS is currently being used by multiple experiments at CERN providing a total raw storage space of 86PB. A brief overview of EOS’s architecture is given then its main and latest features are reviewed and some operations facts are reported. Finally, emphasis is laid on the new infrastructure aware file placement and access engine.

1 Introduction
EOS is an open source distributed filesystem (DFS) developed at CERN. It follows some simplified POSIX semantics. It runs on commodity hardware and it is mainly focussed on low latency, high availability, ease of operation and low total cost of ownership. It is built on top of the XRootD framework and thus offers a native XRoot access to the data. Some of the closest major DFS’s to EOS are DPM, GPFS, Lustre and CephFS. First, only the three latter are strictly POSIX compliant. Compared to GPFS and CephFS, EOS proposes a simpler architecture which makes it easier to set up and to operate. It also reduces the latency as no blocking inter-component communication is required to serve requests. As a consequence of this architecture, the hardware of the head node is the scalability bottleneck of the system. GPFS is commercial software, which in many cases, is a major distinction per se. GPFS relies mainly on RAID storage whereas EOS is based exclusively on RAIN storage. CephFS is a filesystem implemented on top of the Ceph object storage system. This additional layer can also provide block device access. EOS does not propose such a unifying approach and remains simpler. Lustre does not provide data replication of any type. This is delegated to dedicated hardware/subsystem. DPM is more focussed on small instances and less oriented toward low latency access.

EOS uses an agile development cycle which leads to numerous releases. In the first part of this paper, we expose the latest evolution of EOS, including new features that are already in operation and those which will be used soon. Recently, the environment in which EOS is operated at CERN went through a major change. A second computer centre was opened in Hungary, and now, EOS runs over a WAN. In the
second part of this document, we make a detailed focus on an evolution which was driven by this change: the new infrastructure-aware file placement and access engine\(^1\).

2 Architecture

EOS is an open source project developed at CERN in the Data Storage Services group of the IT department. It is written in C++ and released under GPLv3 license. It is exclusively based on open source technologies and does not have any dependency on commercial software including database.

EOS’s architecture has been designed and kept simple in order to allow easy administration of the instances and to keep the latency low. An instance is merely composed of three types of components. These software components are just plug-ins into the same XRootD server daemon thus being given different roles.

![Architecture of EOS](image)

**Figure 1. Architecture of EOS**

**MGM**: This is the management component. This component usually runs on a dedicated machine (optionally two with one in slave mode for fast fail-over). This machine is the head node of the instance. This component provides:

- authentication and identity management
- permissions and quota accounting
- namespace and file placement/access
- instance management
- node balancing and draining

This component is not involved in any data transfer.

**MQ**: This is the message queuing component. It is actually a message broker that allows the coordination of the asynchronous messaging in the instance. Commonly, it runs on the same machine as the MGM.

**FST**: This is the file storage component. It physically stores the files (replicas or stripes). It transfers data to and from the clients. It is also in charge of checksumming and disk verification. This component runs on many servers - currently, up to a few hundred.

Figure 1 gives a schematic view of how these components interact in a typical file operation from a client.

\(^1\) In this paper, *file placement and access* is also referred as *file scheduling*
3 Operations
EOS has been in operations at CERN since 2011. Today there are 6 instances running. They sum-up to the following facts:

- 195M files
- 86PB of raw space
- 35000 disk drives
- 1 lost files per million and per year over the two last years

The biggest instance has 70 million files and 26PB of raw storage in 10800 drives scattered in 421 nodes. Peak reading rate in operations was reached at 15GB/s. This is far from the limits of the instance - most of nodes having a 10Gb/s link - but this is constrained by the physics analysis use case. Some scalability tests have been run and it has been reported that within the limits of the current hardware, this instance can double in size without hitting any scalability issue.

4 Features and Evolutions
EOS targets mainly the physics analysis use case at CERN. In average, it is defined by

- write once, read many
- possibly highly concurrent access of a bunch of hot files
- simplified POSIX access including scattered short reads, mainly through ROOT.

It drives most of its evolutions. In this section, we give an overview of the main features of EOS and of their latest developments.

4.1 Authentication, user accounting and data access
EOS provides strong authentication via Kerberos 5, X509 and Simple Shared Secret. User/group mapping comes together with UNIX permissions and ACLs. An advanced quota system runs along it and provides user and group quotas. Lately, a new project quota has been added. It is attached to a subtree of the namespace. A configurable recycle bin feature has also been added and now provides an additional data safety measure in operation especially against erroneous client-side individual or bulk deletions.

4.2 Core features
EOS has two strong traits in its architecture.

The first one is low latency. It is helped by a fast in-memory namespace whose memory footprint has been reduced as well as its boot-up loading time. Dramatic improvements in case of scattered IO have been brought by the implementation of vector reads; they should hit operations soon.

The second trait is tunable physical layout. In EOS, the storage unit is a file and data placement/replication is managed at this level. While the standard replica layout is still the most widely used, newer RAIN layouts using erasure encoding allow to set more accurately the desired trade-off between file read availability, write speed and redundancy. As the access and replication requirements about a file are likely to evolve, a converter engine has been implemented and is now used in operations. It allows layout changes to be carried out upon individual requests or under the supervision of the new LRU engine that watches the last access time stamps of files. Physical placement and access of the stripes/replicas has recently received infrastructure awareness. Taking into account networking, racks and locations increases speed and availability of the whole instance while it decreases internal data transfers.

EOS’s potential of scalability has recently been levered by allowing delegation of the authentication to dedicated servers. It can offload this burden from the MGM node as it has
been identified as one of the most CPU consuming tasks. It is ready to use but not yet in production.

4.3 Protocols and client features
EOS is being operated in multiple ecosystems. To that end, it offers native access to the data through XRootD and HTTP(S)/WebDav.

Interfaces are provided for SRM and more recently for S3 through HTTP. A Dropbox-like client has also been added. A GridFTP DSI plug-in is also provided to run gateways between GridFTP and EOS. Lately, it has been rewritten to allow interaction with latest versions of XRootD and Globus software.

EOS features a multi-user FUSE client which allows to mount EOS instances onto a client file tree. It has been rewritten. It is more stable and now provides better caching, usage of vector reads as well as parallel streams for RAIN layouts.

Finally, a ROOT integration is provided. It has been vastly improved thanks to caching optimization and vector reads.

4.4 Availability, Sustainability and Operating Costs
In this field, EOS keeps evolving towards higher availability and simpler operations.

EOS has the ability to run in dual master/slave MGM node configuration. The slave MGM then follows all the operations and takes over in case of failure of the master MGM. The fail-over mechanism has been reimplemented and is now instantaneous. Being based on XRootD 4, latest versions of EOS benefit from the updates of the framework, including IPv6 compliance.

Everyday operations are eased by automatic disk/node draining/balancing for which policies have been improved. These work in close collaboration with a background disks scrubbing process and a real-time filesystem check feature. In addition, a new periodic direct-IO file checksum scan has been implemented to check the integrity of the data actually stored on disk. It features an optional auto-repair mechanism able to recreate replica/stripes if enough data is still available.

5 New feature closeup : infrastructure aware file scheduling
5.1 Previous Architecture
So far, EOS has been aware of the infrastructure it runs on. This information has mainly been incorporated in the way scheduling groups are designed. A scheduling group is a bunch of machines and a scheduling subgroup is a list of filesystems, each one being taken from each machine in the group (fig. 2). Each replica/stripe being allocated in the same subgroup, it avoids collocation of any stripe/replica of the same file and then avoids a common point of failure (CPOF) related to files having two replicas/stripes hosted by the same machine. In this context, file placement is carried out by a Round-Robin selection of the subgroup followed by a random selection of the target remote filesystems, the sampling being weighted with the network and disk activity of each remote FS. The file access is also done using a weighted random selection but among the remote FS hosting a replica/stripe of the file being accessed.

5.2 New Architecture
Bringing infrastructure awareness to EOS (fig. 3) brings three major types of improvement to this initial design.

- skip many more types of CPOF like racks, routers, rooms and sites.
- optimize file access by directing the client to the closest available source.
- optimize internal data moves triggered by balancing and draining.
These assets are especially important when operating in a WAN.

**Figure 2.** an EOS Scheduling group and its subgroups (horizontal slices)  

**Figure 3.** Scheduling tree for a scheduling subgroup

In this new design, the concepts of scheduling groups and subgroups (fig. 2) are still valid. The subgroup gets enhanced from a flat vector of remote FS to a tree structure of the underlying infrastructure (fig. 3).

File placement starts the same way with round-robin selection of a subgroup. Selecting the target FS can then be carried out following one of multiple policies.

- spread out replicas/stripes as much as possible in the tree to strengthen data safety/availability by minimizing the number of CPOF
- concentrate replicas/stripes as much as possible around a given node in the tree to maximize the file access capacity for a given group of client machines.
- any kind of intermediate policy between the two above, possibly specific to the physical layout of the file.
- any policy involving any kind of distinction in the branches of the tree (e.g. remote site, SSD rather than HDD).

File access algorithm is also modified. First, a copy of the scheduling tree is populated with the available stripes/replicas for the file being accessed. Then, in the scheduling tree, the closest node to the client is computed. The closest replica/stripe to that node is then selected.

Other types of information can be integrated to the selection on the remote FS in the tree. It can lead to convenient behaviours like online balancing of the storage nodes, network traffic management, etc. For instance, the current implementation skips saturated nodes as much as possible and provides an optional inline storage balancing feature.

**5.3 Implementation**

Compared to the previous one, the new architecture is "stateful" because trees and internal structures need to be kept up-to-date. It was implemented with two major constraints in mind:

- No significant latency should be added to the file placement/access operations.  
  This was carried out by using two distinct types of structures: trees that are flexible and easy to update and snapshots that are compact and very fast (typically Mops/s). The latter retain the structure of a tree and some data attached to it and can be efficiently copied and used as a support for further computations.
- The new subsystem should scale linearly with the number of threads. 
  To ensure this, the implementation makes sure that only a small amount of data needs to be concurrently updated by the serving threads - which can be many. When these updates are needed, they are processed using atomic instructions without any mutexing.
Figure 4. Infrastructure-aware implementation overview

Figure 4 shows an overview of the new architecture. The continuous update of the scheduling tree is fed with asynchronous messages from the FST nodes periodically reporting their state. A reference snapshot of the tree is also made periodically and then used to carry out all the placement/access requests. As a result, there is a latency in the state of the instance as it is seen by the system. This could lead to overloading some FSTs because they are seen as idle while they actually have been selected to serve many requests. To bypass this latency, every time a request is served, the reference tree is amended with an approximation of the overhead caused by the operation. To avoid mutexing, this is made using atomic instructions. As a side-effect of this design, a tiny part of these updates is also lost on the way. It is not an issue as we already deal with an approximation of the correction of the instance’s state. This approximation is not extremely precise and does not need to be as errors don’t get accumulated.

Tests were run on a small instance with 8 FSTs with 8 FSs each. The MGM node had 4 core i7 2.2Ghz cores and 8GB of RAM. Concurrent read and write of small files (a few bytes) were issued to/from the instances. Half of them with the former algorithm, half of them with the new algorithm. Execution time of the placement/access procedure were recorded and averaged (fig. 5).

|                | 2 replicas | 3 replicas | raid6     |
|----------------|------------|------------|-----------|
| placement      | 250/150    | 270/155    | 320/160   |
| access         | 100/135    | 140/135    | 270/140   |

Figure 5. Execution time in µs. Flat alg./Tree alg.

In most cases, latency is reduced. The order of magnitude remains the same: 0.1ms per operation. Given the architecture and these measurements, the derived processing capacity is $O(10k$ schedulings/thread/s) scaling linearly with the number of cores.

6 Conclusion
Latest enhancements of EOS have been reviewed. The new tree-based file placement and access algorithm has been presented. It offers several assets to optimize EOS operations in a WAN. This system will likely be introduced into the balancing and draining algorithms. The whole system will become operational in a near future.