Journal of Physics: Conference Series **331** (2011) 052015 doi:10.1088/1742-6596/331/5/052015

Exabyte Scale Storage at CERN

Andreas J. Peters
CERN IT-DSS, Geneva, Switzerland
E-mail: andreas.joachim.peters@cern.ch

Lukasz Janyst
CERN IT-DSS, Geneva, Switzerland
E-mail: lukasz.janyst@cern.ch

Abstract. The future of data management for LHC at CERN brings new requirements for scalability and a change of scheduling and data handling compared to the HSM mass storage system in use today. A forecast for disk based storage volume at CERN in 2015 is on the Exabyte scale with hundreds of millions of files.
A new CERN storage architecture is represented as a storage cluster with an analysis, archive and tape pool with container based data movements and decoupled namespaces.
Main assets of a new system is high-availability and life cycle management for large storage installations. Today this is one of the major issues at the CERN computer centre with more than 1,000 disk servers and continuous hardware replacement. Another key point is distributed meta data handling with in-memory caching and persistent key-value stores to reduce latencies and operational complexity.
Focus of this paper will be on the analysis pool implementation providing low-latency, non-sequential file access and a hierarchical namespace. A summary of performance indicators and first operational experiences will be reported.

1. Introduction

1.1. LHC Storage Era

There is a growing demand for data storage and analysis caused by the data taking of the Large Hadron Collider LHC [2]. Today CERN stores 12 million new files per month in the CASTOR storage system [1]. The volume is expected to increase three times until 2015 reaching 0.125 EB. Around 2.000 storage servers are used to store experiment and user data. As a result life cycle management, file system losses and corruptions are becoming a major issue for daily operations. However, there is a huge potential in the existing computing and storage infrastructure at CERN, which can be exploited further. Adapted software and storage usage can help to use existing resources in a more efficient way and to lower operational costs. The presented storage software prototype is trying to achieve these goals.

1.2. Transition from a hierarchical to a Tier model

The CERN storage system is a hierarchical storage management system designed to meet the requirements of central data recording: data files produced by LHC experiments are transparently

moved back and forth from a disk cache to tape storage based on defined policies. LHC experiments have integrated the CERN storage system into their GRID frameworks [3]. However, for several use cases the preferred mode of operation is to disable automatic garbage collection on the disk cache to be able to manage the currently interesting data from experiment workflow systems. These GRID middleware tools provide high level storage management and data movements between so-called storage elements. To schedule and run efficiently dataset oriented bulk transfers it is desirable for all the files within a storage element to have a similar access latency.

Figure 1 reflects a new tier storage model following these ideas. The storage system is divided into

- a low latency disk-based analysis storage supporting both random and sequential **rw** access
- a medium latency disk-based archive storage for sequential **read** and **write-once access**
- a high latency tape storage for sequential **read** and **write-once** access to data containers (datasets)

The medium latency storage is optional and it's effectiveness is dependent on cost considerations of disk and tape storage media.

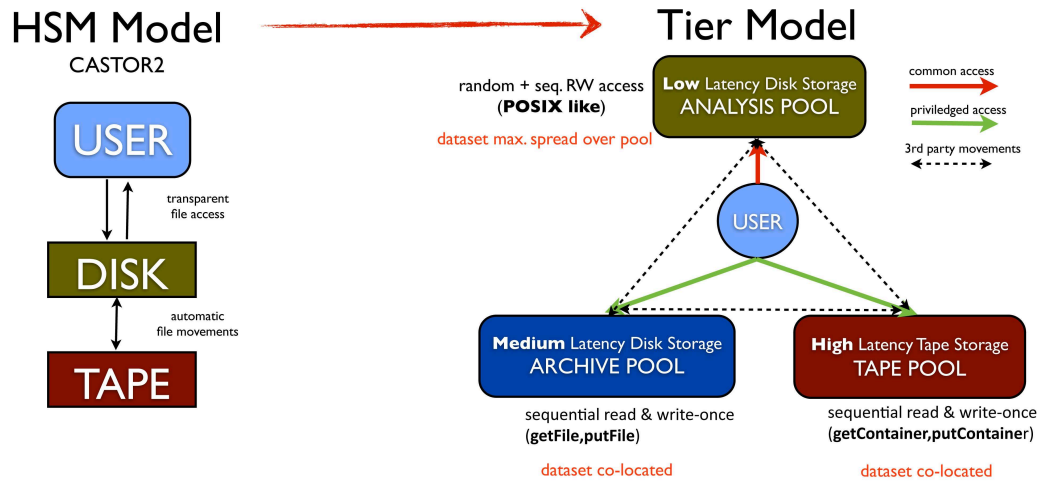

Figure 1. Storage Model Transition

In the analysis pool a dataset is maximally spread over the available nodes to provide highest possible throughput. A dataset (also called container or directory) defines how files can be grouped for bulk operations. In the archive and tape pool datasets are co-located to minimize the impact on the completeness of datasets in case of hardware failures e.g. if a node fails only a small fraction of datasets is unavailable (in contrast to all datasets being incomplete).

1.3. Exabyte Scale

Table 1.3 shows access latency and a conservative calculation for the maximum number of files, containers and total volume in the three described storage tiers based on an average file size of 1 MB. For comparison: the current average file size in the CERN storage system CASTOR is approximately 200 MB which would downscale the number of files and containers by a factor 200.

In the following part of the document we describe the development work done on the EOS prototype of an analysis pool.

	Access latency [s]	Files [#]	File Container [#]	Volume [bytes]
Analysis Pool	$10^{-3} - 10^{-2}$	$10^{9} - 10^{10}$	$10^{6} - 10^{7}$	$10^{15} - 10^{16}$
Archive Pool	$10^{-2} - 10^{-1}$	$10^{11} - 10^{12}$	$10^{8} - 10^{9}$	$10^{17} - 10^{18}$
Tape Pool	$10^{1} - 10^{3}$	$10^{11} - 10^{12}$	$10^{8} - 10^{9}$	$10^{17} - 10^{18}$

2. EOS - Analysis Pool

The EOS disk pool project was started with a project mandate in April 2010 in the IT-DSS group at CERN based on the following requirements:

- POSIX-like **rw** file access (random + sequential + update)
- hierarchical namespace
  - 10^{9} files
  - $10^{6} - 10^{7}$ directories
- strong authentication (Kerberos5 [7], GSI/X509 [8])
- quota system for users and groups
- file level checksums (SHA1, MD5, ADLER32, CRC32)
- high availability of services
- tunable reliability of stored data
- high efficiency and low operational costs
- transparent life cycle management
  - dynamic pool hardware sizing
  - replacement without down-times

2.1. Architecture and basic concepts

EOS is based on three components: management server, message queue and file storage services. All components are currently implemented as plug-ins for the xrootd storage server [4]. Figure 2 explains the functionality implemented by each of the components.

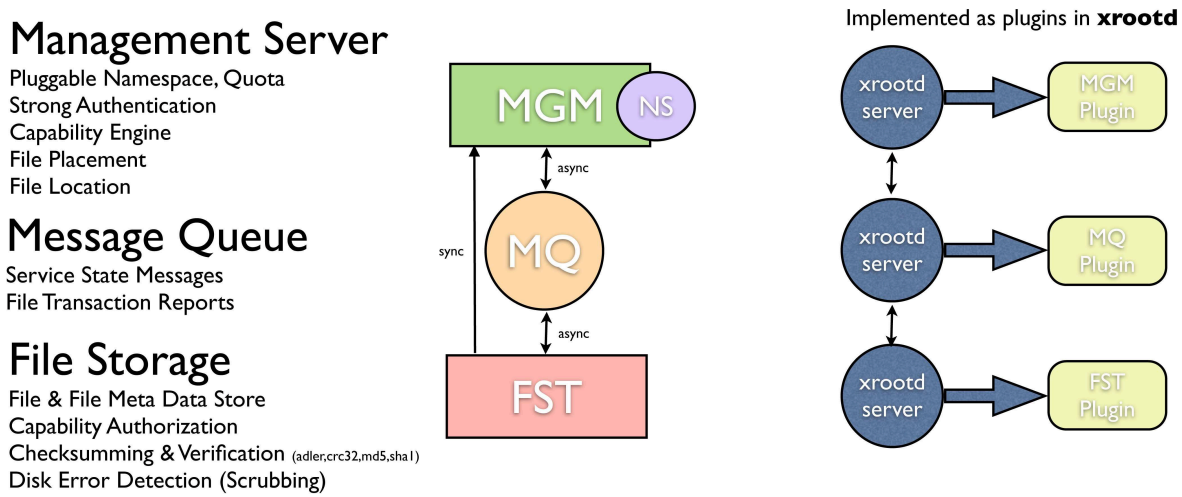

Figure 2. EOS Architecture

A fundamental concept of EOS is to use a set of single disks (JBOD) as storage media without the need to build local RAID arrays. All storage nodes are divided into groups and within one group files are placed using file-level network RAID algorithms. The currently implemented algorithm is RAID-1(N) where (N) is the number of replicas for each file. Dual and multiple parity RAID algorithms are also considered. The storage cluster is self-healing: missing replicas are recreated on the fly. For RAID-1(N) at least one replica has to be accessible to guarantee file availability. The used redundancy algorithm defines the quality of service in terms of file availability in case of single or multiple disk failures. File systems can be migrated online between nodes to simplify life-cycle management.
The namespace is a common bottleneck in storage systems. In EOS it is implemented as a pluggable component which can be exchanged easily. The current implementation is built on in-memory hash maps. The required centrality makes a trade-off between scalability and latency allowing to scale out read open requests. Write open scaling is possible by splitting the namespace (see figure 3). This limitation is sightly compensated by the fact that roughly only one quarter of all requests are indeed write requests (2010 data).

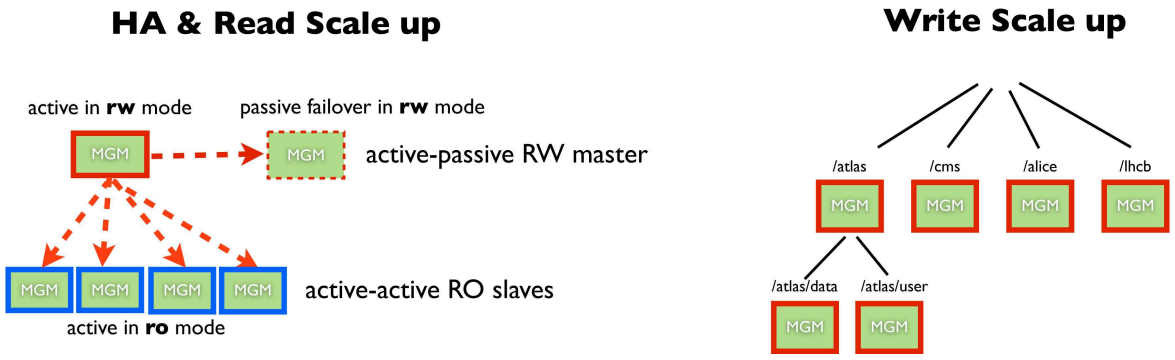

Figure 3. High Availability and Scale-Out

3. Namespace

The hierarchical in-memory namespace is built on top of the google sparse hash implementation [5], which has a minimal size overhead for each hash entry. Every creation, modification and deletion of meta data of files and directories is stored in a sequential change-log file. On startup the change-log file is read and the in-memory hash maps are rebuilt. After repetitive meta data updates on files obsolete meta data records accumulate in the change-log file. EOS provides a tool to compact the change-log files by removing obsolete records.
The namespace provides two views:

- directory view
- file system view

The directory view provides a familiar tree structure. The file system view allows to list all file ID-number on a particular file system. Keeping all meta data in memory is the fastest possible solution. The size is limited by the amount of available memory. The startup time is correlated to the in-memory size and the amount of change-log entries. A typical namespace uses 1 GB of memory for 1 million files. Size variations are dominated by the average length of file base names.

3.1. Performance Tests
Various tests have been done to evaluate the performance of this namespace. In these tests the message queue and manager service were running on a 128 GB 2.8 GHz Xeon machine. Files were stored on a single file server. ROOT [6] was used as a client avoiding consecutive authentication handshakes for each file open. Table 3.1 summarizes the results.

File Creation Test The average creation rate of 0-byte length files for a single client is 220 Hz. For 22 clients an average rate of 1 kHz is observed to create 10 million files in a namespace with 10.000 directories and 1.000 files per directory.

File Read Test The average file open rate for a 350 clients over 100 million read requests in a 10 million entry namespace results in 7 kHz with 20 % CPU usage on the namespace node.

Change-log Compacting and Namespace Boot The namespace of the 10 million files creation test (30 million change-log records) can be compacted with 135 kHz in 74 seconds. Booting this compacted namespace parses all meta data records at 200 kHz in 49 seconds.

Namespace Queries A 'find' on the full 10 million entry namespace returns full path names at 62.5 kHz. A 'find' by file system id returning file ID-numbers delivers 330 kHz (1.2 million entries returned). A 'find' by file system id returning full path names delivers them at 90 kHz.

	Create	Read	Compact.	Boot	find	find(ID) by fsid	find by fsid
1 client	0.22	-	135	200	62.5	330	90
22/350 clients	1	7	-	-	-	-	-

Table 1. Meausured namespace rates in kHz

It should be noted that a seperate deployment of message queue and management service allows further performance improvements.

3.2. Scalability
The described namespace implementation can scale to approx. 100 million files using 80-100 GB of memory. The boot time for such a namespace is around 10 minutes. Assuming an average file size of 200 MB/File and 2 replicas the total space of a storage pool would be 40 PB. Considering the current hardware used at CERN this would be 1.000 nodes providing 20.000 file systems. This is comparable to the current size of all CASTOR disk pools.

3.3. Optimized Namespace
Modern PCI-Express based solid state drives allow random access with more than 100.000 IOPS. These devices can be used to read meta data directly on demand from the device and keep only the two needed namespace views with offset pointers into the change-log file in memory. This will allow to increase the namespace size by another order of magnitude. If the active set is small even a normal disk array can already provide sufficient namespace performance due to the OS buffer cache.

3.4. Performance Evaluation
The EOS namespace implementation is two order of magnitudes faster than the current storage system at CERN and depending on the test setup comparable or slower than the performance of high-end storage systems like Lustre [9] or GPFS [10].

4. First operational experiences

A prototype instance has been setup for the ATLAS experiment and concurrent test jobs have been run successfully against a medium size storage system with 27 nodes. In these Hammercloud tests [11] analysis jobs were running with > 99.99 % efficiency and 63 % cpu/wall-time efficiency [12]. During the tests data stored on 27 nodes have been migrated temporarily to 8 new nodes to reconfigure the disk layout of the storage nodes. After migration all checksums have been verified to identify possible file corruptions. The migration has been completely transparent with no service unavailability.

5. Summary and Outlook

After six month of developement all needed functionality to run as a GRID storage element has been implemented and validated. Tools and procedures for operation are still evolving. We plan to further develop and evaluate the software in production in the context of large scale storage tests with LHC experiments.

6. Acknowledgements

We would like to thank Andy Hanushevsky for the excellent quality and stability of the xrootd server implementation used as a fundamental building block of the EOS prototype. Furthermore we thank all members of the CERN IT-DSS group, the ATLAS LST team, the ROOT team and Bernd Panzer for participation, help, contributions, operations and fruitful discussions.

References

[1] *CASTOR - CERN Advanced STORage manager*, http://castor.web.cern.ch/castor/
[2] *LHC - The Large Hadron Collider*, http://lhc.web.cern.ch/lhc/
[3] *LCG - Worldwide LHC Computing Grid*, http://lcg.web.cern.ch/LCG/
[4] *The Scalla Software Suite: xrootd/cmsd*, http://xrootd.slac.stanford.edu/
[5] *Google Sparse Hash*, http://goog-sparsehash.sourceforge.net/
[6] *ROOT*, http://root.cern.ch
[7] *MIT - Kerberos Consortium*, http://www.kerberos.org/
[8] *GT Security (GSI)*, http://globus.org/toolkit/security/
[9] *Lustre*, http://lustre.org,
[10] *GPFS*, http://www-03.ibm.com/systems/software/gpfs/
[11] *Hammercloud*, http://hammercloud.cern.ch/hc/
[12] *Commissioning of a CERN Production and Analysis Facility Based on xrootd*, Campana,S. et al., Preprint CERN-IT-2011-003/this proceedings