ECFS: A decentralized, distributed and fault-tolerant FUSE filesystem for the LHCb online farm

Tomasz Rybczynski\textsuperscript{1,2} Enrico Bonaccorsi\textsuperscript{2} Niko Neufeld\textsuperscript{2}

\textsuperscript{1}AGH University of Science and Technology, Cracow, Poland
\textsuperscript{2}CERN

E-mail: tomasz.rybczynski@cern.ch

Abstract. The LHCb experiment records millions of proton collisions every second, but only a fraction of them are useful for LHCb physics. In order to filter out the “bad events” a large farm of x86-servers (~2000 nodes) has been put in place. These servers boot from and run from NFS, however they use their local disk to temporarily store data, which cannot be processed in real-time (“data-deferring”). These events are subsequently processed, when there are no live-data coming in. The effective CPU power is thus greatly increased. This gain in CPU power depends critically on the availability of the local disks. For cost and power-reasons, mirroring (RAID-1) is not used, leading to a lot of operational headache with failing disks and disk-errors or server failures induced by faulty disks.

To mitigate these problems and increase the reliability of the LHCb farm, while at same time keeping cost and power-consumption low, an extensive research and study of existing highly available and distributed file systems has been done. While many distributed file systems are providing reliability by “file replication”, none of the evaluated ones supports erasure algorithms.

A decentralised, distributed and fault-tolerant ”write once read many” file system has been designed and implemented as a proof of concept providing fault tolerance without using expensive - in terms of disk space - file replication techniques and providing a unique namespace as a main goals.

This paper describes the design and the implementation of the Erasure Codes File System (ECFS) and presents the specialised FUSE interface for Linux. Depending on the encoding algorithm ECFS will use a certain number of target directories as a backend to store the segments that compose the encoded data. When target directories are mounted via nfs/autofs - ECFS will act as a file-system over network/block-level raid over multiple servers.

1. Introduction

LHCb, one of the four large experiments in CERN is a dedicated heavy-flavour physics experiment designed to perform precise measurements of CP violation as well as rare decays of B hadrons in the Large Hadron Collider (LHC) [1].

During operation the detector records about 15 millions of events per second. Only a fraction of the data contains interesting physics. Furthermore, storage of such a great amount of data is not possible.
Interesting events are selected by a special triggering system composed of two levels. The first level is implemented in custom electronics. It is a real-time system and it selects one million events for further processing.

The High Level Trigger (HLT) is the second level of trigger in LHCb. It is a large collection of software algorithms, which process the data on a large farm of ~2000 nodes - the Event Filter Farm (EFF). The EFF is composed of 56 sub-farms. Each sub-farm consists of a boot server and several nodes, connected by a local area network. The nodes in the farm are x86 servers booting and running from NFS, as a diskless system, however they have a local hard disk, which can be used for temporary storage.

The farm was meant to process online data coming from the detector but the detector is sending the data only 30% of the time. Therefore to use CPUs more efficiently LHCb made a decision to introduce deferred triggering [2]. The data from the detector is now partially written to the local drives of the nodes, and processed later, when no live events are coming.

This approach leads to better use of CPU time, however has a serious inconvenience: with such a large number of nodes, disk failures are statistically more frequent. In case of failure, not only the data stored locally on the node is lost, but the node cannot be used for further processing since it has no access to the data.

The disk needs to be changed quickly in order to restore the server, which is inconvenient because the servers are installed underground (difficult access) and there is no 24/7 system administration support. One solution would be to mirror all disks locally, but this is rather costly and inefficient. We wanted to find a more efficient solution, which still protects is from individual drive failures.

2. Literature study

At the beginning we made a list of the requirements for the possible solution. The main goal is to provide fault-tolerance to the data. This has to come with distributed access to the files. The ideal situation would be having a single namespace for all files stored on the farm. Additionally the file system should use POSIX semantics and be adapted to our purpose - it has to be fast enough for LHCb’s purpose; required write speed of a single data stream from the trigger is about 5 MB/s per node). Conversely for the given use-case a “write once read many” implementation is sufficient because a single writer process writes one file sequentially per node and no modification of the file is ever done.

We have started with looking at some generally available file systems. GlusterFS [3] has been evaluated as first; it does only support fault-tolerance using file replication. This does not improve efficiency over local mirroring requirements and therefore has not been considered as a possible solution.

We tried to adopt Linux Redundant Array of Independent Disks (RAID) [4] to a highly available cluster [5] of nodes, exporting their local drives by iSCSI to all other nodes. However it seems that the RAID software, specifically the meta-data management during rebuilds, in the current release cannot be used in a distributed environment. This functionality is planned in future releases [6].

The last possible solution that we found was less well-known TahoeLAFS [7] - a distributed, peer-to-peer file system providing encoding of the data similar to used in RAID 6 - with use of Reed-Solomon codes. But the file system does not provide POSIX semantics and was not suitable for our purpose in other aspects: in our configuration only one write at time is possible, and so called Introducer node is a SPoF. Solving these issues, even if possible, would take a lot of time with no guarantee of success.

The biggest challenge was to achieve fault tolerance by using RAID or similar technique on top of distributed filesystem. Non of the tested filesystems provides client-server functionality on a single node together with erasure-coding of the data and allows to access this data by other client-server nodes in the network.

However, this study inspired us to use some of the functionalities and design and implement our own file system. A file system has been built on three main components: FUSE - to provide POSIX
semantics, and for easy implementation, NFS - a well-known and robust method of distributing file access over network, and Erasure Codes - to store the data in a similar way as in RAID 6.

3. Erasure codes
Erasure coding is a Forward Error Correction coding technique used for binary data [8]. A binary message, consisting of k symbols, is encoded into a k + n symbol message, so called code word, in such a way that the original message can be recovered from any k symbols. A symbol is a packet of binary data of given length.

Erasure codes are commonly used in data storage systems providing redundancy, like RAID 6, in data transmission, such as DSL, and in consumer electronics: CD players.

ECFS uses coding functions from the Jerasure [9] library. Several different coding techniques are present:
- Reed-Solomon-Vandermonde
- Reed-Solomon RAID 6
- Two Cauchy methods
- Two liberation methods
- Blaum Roth method

The algorithms differ in the level of redundancy which they provide, performance of encoding, decoding and other aspects. More details about the coding functions and techniques can be found in [10].

4. Filesystem structure
The file system has two functional segments: the implementation of the file system functions, together with the coding functions and all involved structures, and the threads performing I/O operations on the block files. These parts communicate using jobs and queues. These segments will be described separately.

A file written to ECFS mount directory is transformed to a set of redundant parts. These parts are then stored on a number of target directories, which can be any locally accessible directories. These parts are actually regular files, called the block-files. The number of block-files depends on the configuration.
A block-file consists of a set of blocks coming as a result of encoding. The encoding process is taking place during writing the file to the ECFS mount directory. The OS communicates with the FUSE module and it runs ECFS functions. The pointer to a buffer with the data is passed as a parameter to write function and the content of the buffer is copied to the coding buffer. The size of the coding buffer depends on the configuration. It is a multiplication of the number of blocks and the size of single block. The block size defines also a size of chunk of data written at one time to the block-file. With an example configuration the block size is 32 kB, the number of blocks is 18, which gives the size of a buffer equal 576 kB. The chosen block size is equal to configured NFS block size. Encoding starts when the buffer is filled with the data. A part of the buffer is empty and filled by encoding function, so encoding starts after getting 512 kB of original data.

The selected algorithm encodes the content of the coding buffer, which is afterwards divided to a set of blocks. Then the pointers to the blocks are passed to the output threads with use of the queues. The threads append these data to the block files.

The operating system block size is typically very small ~8kB, which can be extended to 128 kB, with the direct_io option used when mounting ECFS. The file is passed in chunks of data with this size. Therefore write function has to be run many times. The instances of write functions involved in the same operation communicate with each other by a structure called fuse_file_info. The structure stores information about the file. Additionally it has a field called file_handler, which is used to store the pointer to the data structure involved in coding.

Every operation creates an independent set of functions and structures, and many operations can be performed simultaneously, however the same resources are used for the threads.

Two types of threads are present: control threads and I/O threads. Each type exists in two groups separately for input (read) and output (write) operations so they are independent. On the other hand the resources are used in different way. Reading is always synced and it uses single buffer for the data. Writing is done asynchronously and it uses a buffer which size limits the total number of output jobs in the queues. Output buffering is useful especially when sending the data through the network to avoid unnecessary waiting on slowly corresponding NFS servers.

After all data is sent, ECFS puts a header at the beginning of every block-file. The header contains a set of coding relevant parameters and two hashes. One hash is calculated for the data inside a block-file, to find which file is causing corruption if this is the case. The other hash is calculated on the data given in system buffer therefore it is a hash of the original file.
When a user requests a file from the ECFS mount directory, ECFS looks inside the target directories for the presence of the proper block files. If it finds them, the files are opened and the header is read to find coding parameters and hashes. ECFS denies reading from a file which doesn’t have any of these values. Then the coding engine reads the file block by block, calculating hashes for the data to compare them with the values stored in the headers.

The coding buffer is filled with the data from the blocks. The data can be successfully decoded if enough blocks are present. Decoded data is passed to the system call.

5. Block file

Every block-file consists of a set of blocks and a header. The blocks are calculated by encoding function. The size of each block is constant and if the original file does not fit to the size, rest of the data is filled with zeros.

The header stores the meta-data of the file. It contains:
- Coding parameters
- Size of the original file
- Block number
- SHA1 hash of the block-file
- SHA1 hash of the original file

Files seen by the user inside ECFS mount directory are the virtual files. The function reading attributes opens the block-file inside first accessible directory, and reads the data from the header. The files accessed by ECFS mount directory can be opened only for reading. Editing the files is not implemented in current release. But still the block-files can be edited from outside ECFS. The user has to provide any necessary protection for the data.

6. NFS

As mentioned before ECFS can be configured to use any set of directories for storing the data parts. Especially it can use mounted NFS shares. This is the case in LHCb online farm.

The nodes on the farm are exporting theirs local drives to other nodes in the network, as shown on the picture below:

On each node all NFS shares are mounted by AutoFS. Every instance of ECFS running on single node is configured to use as the target directories a set of NFS shares and a local disk. ECFS instances can read the metadata directly from the block-files stored locally, therefore they have access to all files
stored at the time in the farm. By using this meta-data each instance of ECFS creates a namespace based on the files stored locally on a node. Because every node stores the same set of files (but different blocks), and newly created file (or deleted) appears in the namespace of every node, this namespace can be seen as unique, shared between all nodes.

ECFS provides locking mechanism using NFS v4 file locking. When a file is opened for writing by other node in the network, all block-files are also locked by writing process and other nodes don’t have access to this data until the lock is released.

Spare nodes are used only in case of a failure. Other nodes are using spares when they detect a failure of a backend storage for the block-files on one of the nodes. This allows to increase the level of redundancy without adding an additional “parity” device. It is convenient when, for example, encoding speed is very important: lower number of redundancy means less calculations and lower network traffic. However one should take into consideration that the spare is able to store only new data and the lost blocks cannot be recovered, therefore the redundancy of such data cannot be back increased. Reconstruction mechanism for the lost block-files is under development.

7. Test results

With the configuration stated above we performed a stress-test and noted performance of the whole system. We used 20 nodes in total; 16 data nodes, 2 parity nodes and 2 spare nodes. Such configuration guarantees required redundancy (for the tests it is even increased) and can be scaled to higher number of nodes by creating independent sub-systems – we don’t need to have all storage directories on nodes in the same namespace, just enough to secure the data.

The test case was based on real working scenario. Each node in network is supposed to write one stream of data at time and simultaneously read eight streams from the file system. The file system was tested in terms of stability also during situation when one of the nodes was switched off, to test the fault-tolerance.

Aggregated performance results taken from hard drive of each node are presented on the figure below:

![Aggregated HDD performance](image)

The test has shown that during normal operation average write speed of whole farm is about 188 MB/s, which means that single node is writing with speed of 10 MB/s. The average read speed is 119 MB/s, that is about 7 MB/s per node (total reading speed of eight files in parallel). ECFS does not provide any load balance mechanisms, only synchronization of written data and forced flushing on a backend storage device. Properly configured NFS server handles well many simultaneous operations, however a huge network traffic and multiple parallel writes to a hard drive slows whole system, which is the biggest issue.
8. Conclusions
The initial study of what available in the open source world in terms of distributed, decentralized and fault-tolerant filesystem has been an inspiration for the design of ECFS. Its main goal to provide storage of data across a number of nodes of the LHCb computing farm in a distributed, decentralized and high available way has been verified by the successful run of the LHCb High Level Trigger on a test sub-farm. In case of failure of a node a spare node kick in online.

The usage of erasures code instead of simple block/file replication it does allow to save a huge percentage of available space depending on the algorithm chosen. Comparing to replication proposed configuration of ECFS saves 80% (with k=16, m=2 and two spares) instead of 50% with use of single replica (and much less with more copies).

We performed a test on a sub-farm, in which every node has an instance of ECFS configured to use as the storage-directories a set of NFS shares. The tests have shown that the file system is working stable even with single failure of a node. The performance of the filesystem suits our needs at the moment, but we have to put an effort in improvement of the file system and investigate bottlenecks. In the future we expect increase in the number of files generated by the trigger and therefore higher writing speed would be needed for a proper operation.

References
[1] The LHCb Collaboration et al 2008 The LHCb Detector at the LHC JINST 3 S08005
[2] Frank M et al 2007 The LHCb High Level Trigger Infrastructure Int. Conf. on Computing in High Energy and Nuclear Physics (CHEP, Victoria, Canada)
[3] Website: http://www.gluster.org/
[4] Website: https://raid.wiki.kernel.org/index.php/Linux_Raid
[5] Website: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux /5/html/Cluster_Suite_Overview/sl-rhcs-intro-CSO.html
[6] Website: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux /6/html-single/6.2_Release_Notes/
[7] Wilcox-O’Hearn Z Warner B 2008 Tahoe: the least-authority filesystem (ISBN: 978-1-60558-299-3) StorageSS Proceedings of the 4th ACM int. workshop on Storage security and survivability (ACM New York, USA)
[8] Luo J Shrestha M Xu L Plank J S 2013 Efficient Encoding Schedules for XOR-based Erasure Codes IEEE Transactions on Computers
[9] Website: http://web.eecs.utk.edu/~plank/plank/papers/CS-08-627.html
[10] Plank J S Jerasure: A Library in C/C++ Facilitating ErasureCoding for Storage Applications 2008 TechnicalReport CS-08-627 (Department of Electrical Engineering and Computer Science, University of Tennessee, Knoxville, USA)