Code health in EOS: Improving test infrastructure and overall service quality

Elvin Alin Sindrilaru[1,*], *Georgios* Bitzes[1,**], *Fabio* Luchetti[1,***], and *Mihai* Patrascoiu[1,****]

[1]CERN, Esplanade des Particules 1, 1217 Meyrin, Geneva, Switzerland

Abstract. During the last few years, the EOS distributed storage system at CERN has seen a steady increase in use, both in terms of traffic volume as well as sheer amount of stored data. This has brought the unwelcome side effect of stretching the EOS software stack to its design constraints, resulting in frequent user-facing issues and occasional downtime of critical services. In this paper, we discuss the challenges of adapting the software to meet the increasing demands, while at the same time preserving functionality without breaking existing features or introducing new bugs. We document our efforts in modernizing and stabilizing the codebase, through the refactoring of legacy code, introduction of widespread unit testing, as well as leveraging Kubernetes to build a comprehensive test orchestration framework capable of stressing every aspect of an EOS installation, with the goal of discovering bottlenecks and instabilities before they reach production.

1 Introduction

First introduced in 2011, EOS [1] is the distributed storage system developed by the CERN IT Storage Group. Initially conceived as a simple plugin to the XRootD [2] framework with the goal of easing instance management by administrators, EOS has since expanded in scope to include support for multiple additional file access protocols (HTTP, gRPC), software RAID, automatic replication, a mountable FUSE POSIX filesystem [8], as well as being the backend to CERN's sync & share platform for physics and user data, CERNBox. [3]

The software was initially designed for modest scale, particularly its namespace subsystem. Optimized for performance and low latency access, instead of scale, the namespace was not intended to surpass 100 million files. However, since then several EOS instances have grown well past that point, with some storing 500 million files or more, thus pushing the design constraints of the legacy namespace implementation to a breaking point.

While EOS has been a widely used and successful product, as evidenced by the constantly increasing user traffic and sheer volume of stored data, the mounting code complexity required to support a wide array of features, coupled with a non-scalable namespace implementation have occasionally led to service instability in the past and an unacceptable rate of production incidents leading to downtime.

*e-mail: elvin.alin.sindrilaru@cern.ch
**e-mail: georgios.bitzes@cern.ch
***e-mail: fabio.luchetti@cern.ch
****e-mail: mihai.patrascoiu@cern.ch

https://doi.org/10.1051/epjconf/202024505027

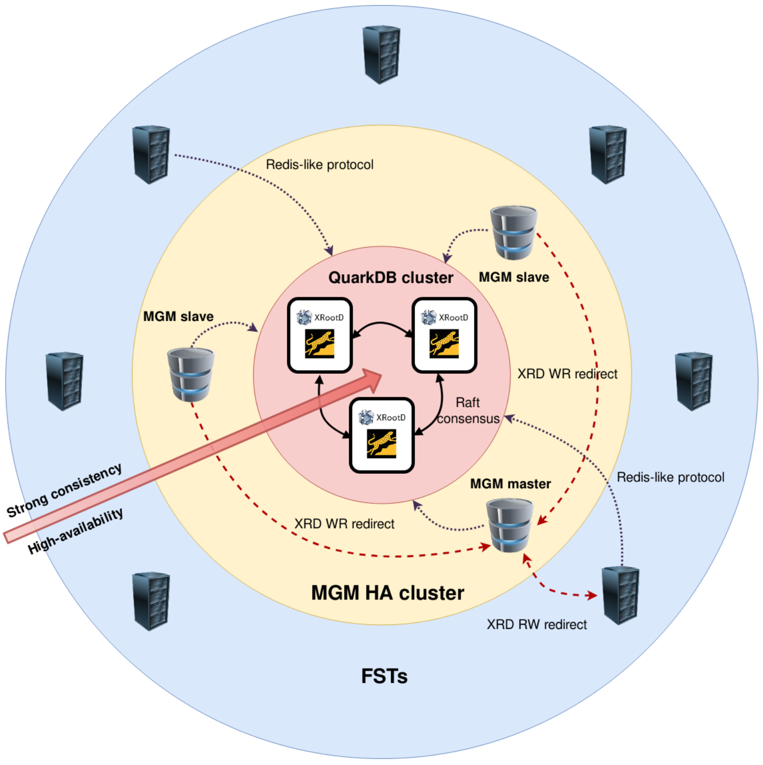

Figure 1. A schematic diagram of the EOS architecture

In this paper, we will describe and characterize the sources of instability, as well as document our efforts in improving the situation through extensive code cleanup and increased focus on automated testing. The rest of this text is structured as follows:

- Section 2 describes the high-level architecture of EOS, including all service components.
- Section 3 gives an overview of our new automated testing infrastructure based on Kubernetes, together with a description on the types of tests being run on it.
- Section 4 summarizes opportunities for refactoring and code modernization in EOS.
- Section 5 offers our conclusion and points to areas for improvement in the future.

2 The EOS system architecture

An EOS instance has the high-level goal of abstracting the files stored across a large number of hard drives (typically in the thousands) behind a single unified namespace and coherent access control mechanisms. To achieve this, we employ several distinct kinds of services:

- The **File Storage Nodes** (FSTs) run on machines physically connected to the hard drives, and are responsible for handling their contents. End-user clients connect to these nodes when reading or writing data.
- The **Metadata Manager Node** (MGM) is tasked with redirecting end-users to the appropriate FSTs, and keep track of where each file is stored within the cluster. In addition, it performs all kinds of administrative tasks such as balancing and draining.

- The **QuarkDB** service persists the contents of the namespace, most notably all file and directory metadata, consisting of three replicated nodes in the standard setup. It is an internal service, no end-clients interact with it. The replicated nodes are kept in sync through raft consensus. [4]
- The **Messaging Queue** (MQ) is responsible for routing messages and heartbeats between the MGM and FSTs. We are in the process of deprecating this service entirely, and handing off its responsibilities to QuarkDB.

Figure 1 shows a schematic of the evolved EOS architeture.

A recent, major improvement in system availability and reliability was brought by the introduction of QuarkDB as the persistency layer for the namespace. [5] [6] Prior to that, the entire contents of the namespace were kept in-memory by the MGM, and persisted through on-disk changelogs. However, keeping the entire namespace always in-memory required inordinate amounts of physical RAM (500+ GB was common), while imposing an excessive delay (30-60 minutes) to MGM service restart due to having to replay the on-disk changelogs and reconstruct the in-memory contents. The resolution of the above constraints, through the introduction of the QuarkDB namespace, has already brought on visible improvements to overall EOS system availability: An MGM crash in a large instance now typically brings it down for less than a minute, compared to half an hour or more before.

3 Automated Testing Infrastructure

Automated testing is a vital component in large-scale software projects, ensuring stability and relability of the codebase. The EOS project has dedicated a good amount of effort into improving the automated testing infrastructure. At the core of this effort stands the continuous integration pipeline, which runs a subset of the testing scripts on the EOS system upon every commit. This section gives an overview of the testing infrastructure employed, the kind of testing performed, and dwelves into a few "exotic" features.

3.1 Continuous Integration

Arguably the strongest aspect of the EOS automated testing infrastructure is the continuous integration pipeline. Upon each commit, the pipeline executes several stages, namely build, docker image creation, testing and RPM publishing. The testing stage involves unit testing, as well as system testing, with a particular emphasis being put on testing the FUSE clients.

In order to perform system testing, a running EOS instance is needed. Before the testing begins, EOS is deployed within Docker containers or Kubernetes pods in the following configuration: one head node, seven storage nodes, one client node, one Kerberos node, one messaging node and, according to the configuration, one namespace node. The system tests will verify most of the functionalities, such as various client commands or server-side mechanisms and will perform different scenarios, both common use-cases as well as others that used to fail in the past. The testing also targets various access interfaces, such as XRootD, HTTP(s) and GRPC. Certain modules of the testing infrastructure target a specific component, such as draining, filesystem checking or message queue communication.

A strong emphasis is placed on testing the FUSE client. A special testing repository is setup, which will perform common cenarios, as well as regression testing. The EOS FUSE client is mounted on the client node, using the Kerberos node for authentication. Once mounted, benchmark tests are performed, followed by scenario-based testing, including demanding tools such as `createrepo`.

If all the testing jobs perform well, the testing stage is greenlit to continue to the publishing stage, where the RPMs get uploaded to the EOS designated repo.

3.2 Kubernetes

By making use of Gitlab's Docker integration, we are able to build Docker images with all necessary components to run an EOS instance on containers in a straightforward manner. Since an EOS instance is composed of multiple services interacting with one another, we chose Kubernetes as the orchestration engine to simplify our continuous integration workflows.

A side project, *eos-on-k8s* has been put in place for the purpose of setting up test EOS instances on top of Kubernetes virtual clusters, straightforward to set up and with short deployment time. During the CI pipeline test stage, templated manifests drive the creation of several Kubernetes resources:

- Multiple Services exposing the appropriate ports.
- Each Service is composed of a Deployment for a single-replica Pod, each running a Container with a particular EOS component such as MGM, QDB, or FST.
- A ConfigMap to handle eos-specific configuration according to the type of test run.
- An optional PersistentVolumeClaim which allows preserving data after cluster deletion or dysfunction. The persistent storage space associated with it is claimed through our Manila/Openstack CERN provisioner.

This way, a fully functional, disposable and distributed instance is deployed in a matter of seconds. Furthermore, by exploiting the Kubernetes Namespace abstraction various instances can coexist at the same time within the same Kubernetes cluster, thus saving computing resources and reducing the pipeline completion time.

The orchestration scripts are generic, and and can be used outside of Gitlab CI as well. This has proved very convenient during targeted debugging sessions and, as a plus, served as a stepping stone in enriching the test suite with more advanced testing strategies, such as continuous fuzz testing.

3.3 Exotic Features

To ensure over-all quality of the software system, different measures are employed, such as constant building and testing on every commit. Among those measures we should also include the so called 'exotic' builds, which are ran only upon schedules, such as nightly or monthly. These 'exotic' builds include:

- Building for different distributions, such as Ubuntu and Fedora
- Building with different compiler toolsets, such as gcc and clang
- Building with different compiler features enabled, such as sanitizers or code coverage

3.3.1 Address Sanitizer

The address sanitizer [7] build is there to check for various kinds of improper memory usage, such as buffer overflows, especially off-by-one scenarios, heap user after free, leaking memory and so on. One of the main advantages of using address sanitizer lies in being a compile-time approach, as well as the low overhead when compared to other memory analysis tools.

The address sanitizer build will compile EOS with the `-fsanitize=address` flag enabled. The resulting RPMs are later installed within a docker container and unit tests are executed. If no errors are triggered by address sanitizer, the unit test job is considered successful.

3.3.2 Code Coverage

The EOS coverage build is an experimental build meant to provide code coverage metrics. At compile time, the following flags are added: `-fprofile-arcs -ftest-coverage --coverage`. The resulting RPMs are used to install and deploy EOS on docker containers, execute the testing infrastructure and collect the coverage data.

The coverege-metric algorithm follows these steps:

– Build EOS with code coverage flags enabled

– Deploy on docker containers

– Execute extensive tests: unit tests, instance tests, stress tests, client tests

– Collect coverage data using the **gcov** tool

– Filter out certain source files (test code and external libraries) using the **lcov** tool

– Create HTML report via **lcov**

When it comes to collecting code coverage metrics on EOS, the most challenging aspects arise from its distributed nature.

First of all, the code coverage *gcov* tool requires certain coverage counters, which are produced during compile time. For this reason, the coverage build produces an additional package called *eos-coverage*, which allows shipping the needed coverage counters. All subsequent calls to *gcov* must be made aware of where this coverage counters have been installed.

The second challenge is related to the mechanism in which coverage data is collected. By default, coverage data is flushed at the end of binary execution. However, for EOS, this would mean needing to stop the service. To overcome this, a signal handler has been implemented, which will capture the *SIGPROF* signal. When that signal is encountered, coverage data is flushed and execution continues.

The last challenge is presented by the loading of dynamic libraries within EOS itself. In order to have proper coverage data, not only the main binary must flush its coverage data but also all the libraries that it loaded dynamically. Therefor, when the *SIGPROF* signal is encountered, coverage data is flushed for this binary, as well as for each dynamic library loaded by it.

Upon addressing these challenges, the resulting EOS system can provide coverage data without having to stop the service. This technique is used in the coverage pipeline job. After executing all the tests, the *SIGPROF* is sent to the EOS processes. Each one will flush its own coverage data, as well as that of dynamically loaded lbiraries, such as the namespace library in the case of the head node. That data is aggregated and filtered and the final coverage report is generated.

4 Further opportunities for refactoring and code modernization

The introduction of the new namespace implementation also meant some of the internal components needed to be re-designed and this proved to be an excellent opportunity to refactor parts of the code base. For example, the draining subsystem was putting too much pressure on the QuarkDB backend since the initial assumption was that all namespace entries were in memory therefore access to them was in the nanosecond range. Once we moved to QuarkDB, access to a namespace metadata object could be in the order of milliseconds and this introduced additional constraints on all other subsystems using the namespace.

The draining subsystem was completely re-desinged and the basic transfer job that represents a drain operation was implemented in a generic way so that other components could reuse it. Furhtermore, we employed the latest features from the XRootD client when it comes to verifying data integrity through checksum verification, fast cleanup and cancellation of third-party-copy jobs with the help of the progress monitor handle.

Following the same line of thought, the fsck subsystem which is responsible for early detection of errors on the diskservers and also correcting them uses the same code for recovery as the drain subsystem. The refactoring of the fsck subsystem is an important milestone since it helps us improve data availability and subsequently user perceived performance. Both the drain and fsck subsystems have dedicated unit and integration tests that ensure future regressions are caught early on.

Last but not least, since EOS now depends on quite a number of external projects integrated as submodules in our git repository, we also started modernising our cmake build infrastructure. First of all, the entire EOS codebase is now C++17 compliant using many of the latest features provided by the standard. Secondly, there are ongoing efforts to move our cmake infrastructure from a pretty ad-hoc design to a modern approach taking full advantage of cmake's targets and properties. The idea behind this philosophy is that targets model the components of an application and they can be anything from executables to libraries. Targets have properties that represent the source files used for the build, the compiler flags required or the library dependencies. Building a project boils down to creating a list of targets and defining the necessary properties on them, eliminating the need of specifying handcrafted include paths or enumerating explicitly all the library dependencies for a target. Going down this path will improve the maintainability of the project as a whole, while also simplifying future additions to the codebase which is paramount in a collaborative environment.

These are just a few areas that have received special attention during the last development cycle, but we're constantly trying to asses and decide on implementing various measures to improve the quality of the code and also to prepare for future challenges.

5 Conclusions and future work

In this paper, we summarized our efforts in raising the code quality of the EOS codebase with the overarching goal of improving service stability and reliability. We believe that the evolution of the namespace subsystem, the adoption of cleaner coding practices with enhanced focus on unit testing, together with a powerful test orchestration framework based on Kubernetes for system-level tests, have all had a positive impact on overall service quality of EOS. In the future, we plan to continue in the same direction, especially with regards to increasing the percentage of code covered with unit tests, and ensuring all EOS code is well-structured, testable, and robust.

References

[1] Peters, Andreas J., Janyst, L.: Exabyte scale storage at CERN. Journal of Physics: Conference Series. Vol. 331. No. 5. IOP Publishing (2011)

[2] Dorigo, Alvise, et al.: XROOTD-A Highly scalable architecture for data access. WSEAS Transactions on Computers 1.4.3 (2005)

[3] Mascetti, L., et al.: CERNBox+ EOS: end-user storage for science. Journal of Physics: Conference Series. Vol. 664. No. 6. IOP Publishing (2015)

[4] Ongaro, Diego, and John K. Ousterhout.: In Search of an Understandable Consensus Algorithm. USENIX Annual Technical Conference (2014)

[5] Bitzes, G., Sindrilaru, E., Peters, A.: Scaling the EOS namespace–new developments, and performance optimizations. EPJ Web of Conferences: Vol. 214. EDP Sciences (2019)

[6] Peters, Andreas J and Sindrilaru, Elvin A and Bitzes, Georgios: Scaling the EOS namespace. International Conference on High Performance Computing, Springer (2017)

[7] Serebryany, K., et al.: AddressSanitizer: A fast address sanity checker. USENIX Annual Technical Conference (2012)

[8] Adde, G., et al.: Latest evolution of EOS filesystem. Journal of Physics: Conference Series, IOP Publishing (2015)