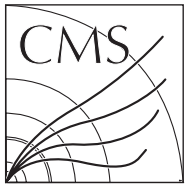
Available on CMS information server

CMS CR -2013/366



The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

28 October 2013

XRootd, disk-based, caching-proxy for optimization of data-access, data-placement and data-replication

Matevz Tadel for the CMS Collaboration

Abstract

Following the smashing success of XRootd-based USCMS data-federation, AAA project investigated extensions of the federation architecture by developing two sample implementations of an XRootd, disk-based, caching-proxy. The first one simply starts fetching a whole file as soon as a file-open request is received and is suitable when completely random file access is expected or it is already known that a whole file be read. The second implementation supports on-demand downloading of partial files. Extensions to the Hadoop file-system have been developed to allow foran immediate fallback to network access when local HDFS storage fails to provide the requested block. Tools needed to analyze and to tweak block replication factors and to inject downloaded blocks into a running HDFS installation have also been developed. Both cache implementations are in operation at UCSD and several tests were also performed at UNL and UW-M. Operational experience and applications to automatic storage healing and opportunistic computing, especially on T3 sites and campus resources, will be discussed.

Presented at *CHEP2013 Computing in High Energy Physics 2013*

XRootd, disk-based, caching-proxy for optimization of data-access, data-placement and data-replication

L A T Bauerdick[1], K Bloom[3], B Bockelman[3], D C Bradley[4], S Dasu[4], J M Dost[2], I Sfiligoi[2], A Tadel[2], M Tadel[2,5], F Wuerthwein[2] and A Yagil[2] for the CMS collaboration

[1] Fermilab, Batavia, IL 60510-5011, USA
[2] UC San Diego, La Jolla, CA 92093, USA
[3] University of Nebraska – Lincoln, Lincoln, NE 68588, USA
[4] University of Wisconsin – Madison, Madison, WI 53706, USA

E-mail: mtadel@ucsd.edu

Abstract. Following the success of XRootd-based US CMS data-federation, AAA project investigated extensions of the federation architecture by developing two sample implementations of an XRootd, disk-based, caching-proxy. The first one simply starts fetching a whole file as soon as a file-open request is received and is suitable when completely random file access is expected or it is already known that a whole file be read. The second implementation supports on-demand downloading of partial files. Extensions to the Hadoop file-system have been developed to allow for an immediate fallback to network access when local HDFS storage fails to provide the requested block. Both cache implementations are in pre-production testing at UCSD.

1. Introduction

In February 2013 the CMS experiment [1] at the CERN LHC finished its first data-taking period, called "Run 1", and entered into the "Long Shutdown 1" period expected to last until spring 2015. However, the physics analyses of the harvested data are still ongoing, as are the detector simulations and related computing activities required for an efficient commencement of upcoming "Run 2". The 20 PB of experiment data in various formats is distributed among participating Tier 0, Tier 1 and Tier 2 computing sites with the goal of optimizing the usage of available computing resources as well as to provide sufficient processing power to all physicists that require access to the data. The "Anydata, Anytime, Anywhere project (AAA) [1] was started with the goal of opening up computing model of CMS to various degrees of remote data-access among all the involved sites. The first stage happened in the US in 2011 by exposing all Tier 1 and Tier 2 storage to the collaboration via the XRootd system [3] and by implementing a comprehensive monitoring framework [4]. The main initial use-case was interactive access for data-analysis. Soon after, standard computing jobs were allowed to utilize remote access both as a fallback in case of a local access error as well as intentionally to better utilize the available CPU resources. Within US, during the first half of 2013, average data-rate among all sites was 250 MB/s, corresponding to about 1,000 concurrent running jobs or about 4% of total US

[5] To whom any correspondence should be addressed.

CMS capacity. There is an ongoing campaign to export data from all remaining, non US, CMS computing centers before summer 2014.

Success of the AAA project, expected increase in data rates for "Run 2", and the promise of 100 Gbps networks becoming available in 2014 are all arguing in favor of loosening up of the CMS computing model. In particular, usage of Tier 2 CPU and disk resources should become more flexible: with all data available at Tier 1 sites there is little incentive for pre-placement of most data on Tier 2 sites — it can always be downloaded when it is actually needed and then kept for as long as it seems reasonable. A significant part of Tier 2 storage, up to 50% and more, could thus be operated as a fluid cache space. Furthermore, as it is known the data exists elsewhere there is no need to store the files in a redundant manner as long as the fallback to remote access can be provided at any point of file-access. Efficient reuse of data cached at Tier 2 centers requires further attention as job scheduling programs need to be both aware of and interact with the file caching infrastructure.

This paper presents two implementations of a XRootd, disk-based caching-proxy developed in the context of the AAA project. We believe that these two services can be used to demonstrate operation of a Tier 2 center on non-subscribed data-sets. Section 2 describes the two caching proxy implementations in detail and section 3 shows results of a scaling test of a proxy running on standard server hardware.

2. Two implementations of disk-based caching-proxy

Since CMS data federation already relies completely on XRootd to provide remote file access, the decision to base caching proxies on XRootd was an obvious one. The XRootd system provides a basic proxy service [5] with a limited in-memory cache. Its main purpose is to provide access into and out-of private networks. However, the implementation allows for an user-provided implementation to be loaded at start-up as a plugin — our two implementations are such plugins, specializations of *XrdOucCache* interface. Both of them are currently undergoing pre-production testing at UCSD.

2.1. Complete-file auto-prefetching proxy

The first implementation simply prefetches complete files and stores them on local disk, serving client request as they come along. This is suitable for optimization of access latency, especially when reading is not strictly sequential or when it is known in advance that a significant fraction of a file will be read. Of course, once parts of a file are downloaded, access speed is the same as it would be for local XRootd access.

The prefetching is initiated by the file open request, unless the file is already available in full. It proceeds sequentially, using a configurable block size (1 MB is the default). Requests from clients are put to the beginning of the download queue and are served as soon as all requested data becomes available. Vector reads are fully supported, too. If a file is closed before prefetching is complete, further downloading is also stopped. When downloading of the file is complete it could in principle be moved to local storage. Currently, however, there are no provisions in the proxy itself to coordinate this procedure. Scheme of proxy operation is shown in figure 1.

A state information file is maintained in parallel with each cached file to store the block size used for the file and a bit-field of blocks that have been committed to disk; this allows for complete cache recovery in case of a forced restart. Information about all file-accesses through the proxy (open & close time, # of bytes read and # of requests) is also put into the state file to provide cache reclamation algorithms with ample details about file usage.

It would be straightforward to modify the proxy code to only fetch required blocks but a preliminary analysis showed that granularity of file access is much higher than a reasonable network request size. Besides, this would lead to increased latency which presents, at this time, a larger problem than the newtork bandwidth.

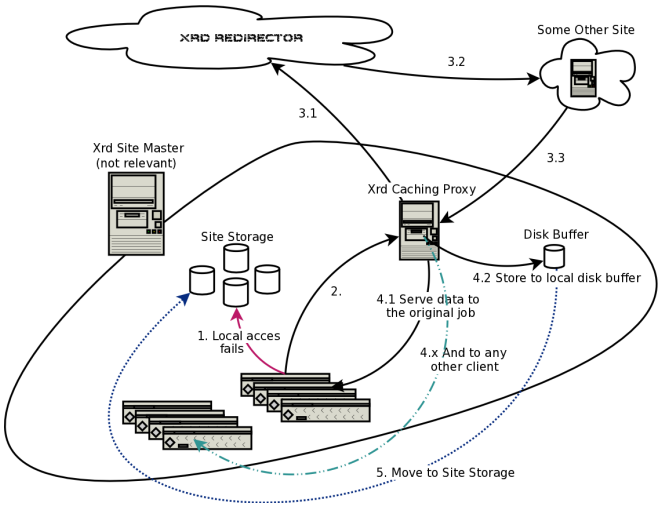


Figure 1. High-level diagram of caching-proxy operation. The same steps happen for both proxy implementations.

1. Reading from local storage fails.
2. Client contacts local proxy.
3. Proxy contacts a redirector to find a replica of the file on some other site.
4. Proxy downloads data, serves it to local clients and stores a copy to disk.
5. File or file fragments can be injected into local storage.

2.2. Partial-file block-based on-demand proxy

The distinguishing feature of the second implementation is that it only downloads the requested fixed-size blocks of a file. The main motivation was to provide prefetching of HDFS blocks (typical size 64 or 128 MB) when they become unavailable on local site, either permanently or temporarily due to server overload or other transient failures. When additional file replicas exist in a data-federation, the remote data can be used to supplement local storage, to improve its robustness, and to provide a means for healing of local files. In particular, our intention is to avoid any local file replication of rarely-used, non-custodial data at Tier 2 sites. As HDFS block size is a per-file property, it has to be passed to the proxy on per-file basis as an opaque URL parameter.

Unlike the full-file prefetching version, the partial-file proxy does not begin prefetching any data until a read request is actually received. At that point a check is made if the blocks required to fulfill the request exist on disk and, if they don't, they get queued for prefetching in whole. The client request is served as soon as the data becomes available. Each block is stored as a separate file, post-fixed by block size and its offset in the full file; this facilitates potential reinjection back into HDFS to heal or increase replication of a file-block.

2.2.1. Extension of HDFS client for using XRootd fallback. After detailed inspection of HDFS client code it was decided to develop a new specialization of *DFSInputStream* class, *XFBFSInputStream* (standing for XRootd fall-back file-system input stream), and to bind it to a custom protocol, *xfbfs*, in HDFS site configuration. This allows users to specify if they want XRootd fallback or not by simple selection of access protocol name. However, in typical HDFS deployment at US Tier 2 centers, HDFS is mounted via FUSE centrally for all users, thus breaking the flexibility of the scheme. To compensate for that a special configuration entry allows system administrators to enumerate HDFS namespaces for which *XFBFSInputStream* should be instantiated. Note that one instance of input-stream gets instantiated for every file that gets opened.

Internally, the work of maintaining an XRootd client instance and communicating to a block-based proxy gets delegated to Java class *XrdBlockFetcher* that has the majority of its functions implemented in C++ using JNI. This class only gets instantiated when lower-level classes of HDFS client can not locate a data source and throw an exception that gets intercepted in *read()* functions of *XFBFSInputStream*. A list of bad blocks is kept so any further attempts at accessing

the same block get redirected to XRootd without retrying to locate it in HDFS.[6] Both classes, *XFBFSInputStream* and *XrdBlockFetcher*, report their operations via UDP to allow monitoring of failures in real time, to estimate performance and load on the proxy, and to, eventually, provide information for storage healing algorithms. This is important because the XRootd fallback can get invoked on any node, for any HDFS client, and the common reporting scheme provides a way of aggregating reports from all computing nodes into a single log file.

Both XRootd proxy implementations, the minimal changes that had to be made to HDFS, and the additional *hdfs-xrootd-fallback* package are expected to become available via OSG *yum* repository before the end of 2013.

3. Scaling test of a complete-file prefetching caching-proxy

It is interesting to observe the limits of caching-proxy in action as this affects its deployment setup at Tier 2 centers under various work-load conditions. A standard, current server machine was chosen for the test: 2×6-core Xeon E5-2620 processors, 64 GB RAM, with ZFS file-system over a RAID–5 disk array. One 1 Gbps NIC was used in the test and kernel network parameters were tuned so as to saturate incoming traffic with a single multi-request stream, e.g., by running *xrdcp* of a file from FNAL.

The test itself was, of course, designed to cause trouble on the machine so that the weakest link in the setup could be determined. The test ran at UCSD Tier 2. In the beginning the disk-cache was empty and all data-files used in the test were known to be available at other US Tier 1 and Tier 2 sites. After that, a dummy CMSSW job was run every 5 seconds on a set of worker machines, each opening a new, unique file. Each job was asking for 2.4 MB every 10 s (240 kB/s). Size of each file was about 1 GB. These numbers are based on average values observed over a large sample of CMS computing jobs, obtained from the AAA XRootd detailed monitoring.

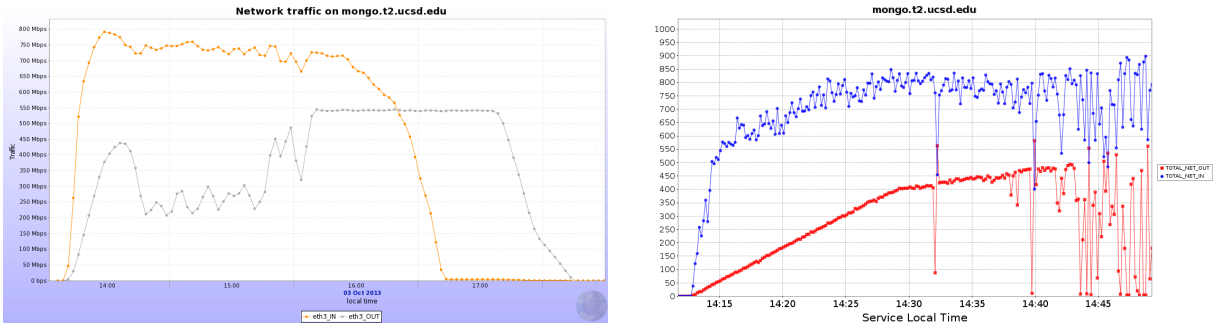


Figure 2. Network traffic on the caching-proxy node during the scaling test. The left picture shows the complete test, data-points are averages over 2 minute periods. The right figure is a detail of the ramp-up period, sampled every 15 s.

Network traffic on the proxy node as a function of time is shown in figure 2. The prefetching traffic rose steeply from the start and began to saturate towards 800 Mbps when 25 connections were open (5 min). Outgoing traffic rose linearly, fully satisfying the requested data-rate up to 400 Mbps or 200 client connections (16 min). After that, a lower output rate increase continued up to 240 connections. At that point, the network stack of the machine become overloaded, the outgoing traffic become chaotic and soon dropped for about 50%. Incoming traffic also dropped

[6] HDFS peforms the block search three times with increasing, randomly staggered delays, all together taking between 20 and 30 s.

for 6%. The machine did not appear to be otherwise stressed, both process load and disk I/O remained low. When prefetching of files began to ramp-down, the output rate soon climbed up to reach the total request rate of all 250 jobs. As another example, a 5-year old server machine had trouble with both disk I/O and network interrupts much earlier, causing lower input traffic (600Mbps) and earlier saturation of output rate (about 280 Mbps or 140 standard jobs).

Therefore, one proxy machine with a single NIC can, with very little tuning, deliver up to 400 Mbps of output traffic and provision about 200 average CMS jobs when no files are available in the cache. We believe this performance can be improved further and following that we will also test setups with multiple, higher bandwidth NICs.

4. Conclusion

Two disk-based prototype implementations of an XRootd caching-proxy have been presented. The main motivation for this development was to provide an optimized access to remote-data, both in terms of latency and data reuse, as well as to facilitate more flexible data-placement strategies among Tier 2 and Tier 3 centers. The prefetching caching-proxy implementation is also suitable for just-in-time data placement. The partial-block caching-proxy implementation, on the other hand, allows computing center operators to reduce replication factor of non-custodial files residing on HDFS-based storage, freeing up disk space for other uses.

Further work will focus first on final optimizations of the proxy implementations and on full-scale, production-grade testing at UCSD. After that, in anticipation of proxy deployment across the whole data federation, integration with job scheduling will be investigated to provide early preloading of data and a better reuse of existing, cached replicas. Interaction of cache with local storage will be studied and tools needed to manage block replication factors and block movement from cache into a running HDFS will be developed.

Acknowledgments

This work is partially sponsored by the US National Science Foundation under Grants No. PHY-0612805 (CMS Maintenance & Operations), PHY-1104549, PHY-1104447, and PHY-1104664 (AAA), and the US Department of Energy under Grant No. DE-FC02-06ER41436 subcontract No. 647F290 and NSF grant PHY-1148698 (OSG).

References

[1] CMS Collaboration 2008 *JINST* **3** S08004.
[2] Bauerdick L, *et al* 2012 "Using XRootd to Federate Regional Storage", *J.Phys.Conf.Ser.* **396** 042009.
[3] XRootd project page: http://www.xrootd.org/.
[4] Bauerdick L, *et al* 2012 "XRootd monitoring for the CMS experiment", *J.Phys.Conf.Ser.* **396** 042058.
[5] XRootd proxy service documentation, in http://xrootd.slac.stanford.edu/doc/prod/ofs_config.htm.