

Operational experience running Hadoop XRootD Fallback

J M Dost, A Tadel, M Tadel and F Würthwein

University of California, San Diego, 9500 Gilman Dr, La Jolla, CA 92093, USA

E-mail: jdost@ucsd.edu

Abstract. In April of 2014, the UCSD T2 Center deployed `hdfs-xrootd-fallback`, a UCSD-developed software system that interfaces Hadoop with XRootD to increase reliability of the Hadoop file system. The `hdfs-xrootd-fallback` system allows a site to depend less on local file replication and more on global replication provided by the XRootD federation to ensure data redundancy. Deploying the software has allowed us to reduce Hadoop replication on a significant subset of files in our cluster, freeing hundreds of terabytes in our local storage, and to recover HDFS blocks lost due to storage degradation. An overview of the architecture of the `hdfs-xrootd-fallback` system will be presented, as well as details of our experience operating the service over the past year.

1. Introduction

The UCSD Tier-2 is one of the six US CMS Tier-2 centers that run Hadoop [?] as its storage backend. In addition, UCSD, as well as all CMS Tier-1's and US CMS Tier-2's, and over 30 EU CMS Tier-2's, belong to the AAA XRootD Federation [?]. In general, assuming a Hadoop cluster is self contained, it is vital to keep redundant copies of files to ensure stability and availability. This comes with a price—it significantly reduces the physical capacity of the storage cluster. However, the AAA Federation provides an alternative to local file replication. There exists a subset of files, or namespace, that is guaranteed to be found at another site in AAA. At UCSD we realized this untapped potential, and developed the Hadoop XRootD Fallback system, a collection of services that allows a Hadoop site to depend on the global replication already provided by an XRootD federation. This in turn reduces the local file replication requirements in the storage cluster, and increases the usable physical disk space available at the site level. The next two sections give some preliminary information on Hadoop and XRootD. Section 4 describes the details of the Hadoop XRootD Fallback architecture. In Section 5, we discuss our operational experience running the system at the UCSD Tier-2.

2. Some details on Hadoop internals

Hadoop consists of two main components, the Hadoop Distributed File System (HDFS), and Hadoop MapReduce. In the context of CMS, we are interested in using Hadoop as a distributed storage system, so this document focuses on HDFS. As the name suggests, the Hadoop Distributed File System spreads a site's storage over a cluster of distributed computers. In order to make this happen, there are two types of roles that a machine might play. It can either be a NameNode or a DataNode. The NameNode is responsible for maintaining the database list



of files present, including their metadata information, and the handling of any client requests to read, modify, or browse the file system. All other machines in the cluster are DataNodes. These nodes are responsible for actually storing the file on disk, and handling the underlying I/O streams responsible for passing data to and from the client.

2.1. How files are stored in HDFS

A given file is first divided into smaller files of a fixed length called blocks. A typical default block size is 128MB. The last block can be less than the block length since files can be of arbitrary length. Assuming site-level file redundancy, each block is then copied N times where N is the replication factor. 1x replication means there is only one copy of the file. Hadoop recommends a minimum replication factor of 3x to ensure availability. However in practice, we use 2x replication at the UCSDT2 for most files. HDFS then distributes the blocks throughout the cluster in a way that no two replicas of the same block should be located on the same node. This ensures that if a node goes down for any reason, that portion of the file can still be accessed from another node. This is effectively how file redundancy works in an HDFS cluster.

3. A high level picture of XRootD

In most general terms, XRootD [?] is a software framework for object discovery and access. Most commonly, it is used for accessing files identified by their name in a directory structure. XRootD supports the construction of a composite namespace by federating sites into a tree-like hierarchy, thus making their data inter-accessible between each other. This hierarchy is known as a storage federation. The advantage of forming a storage federation is that end users wishing to access sets of data across multiple sites do not need to be concerned with where the data is actually stored. Instead a user can ask the top level namespace manager, which will then perform the lookup and redirect the client to talk directly to the server that holds the data. The details of the file lookup and where it actually exists on disk do not need to be known in advance. Subsets of data may or may not be replicated over multiple sites in the federation, the end user only needs to know the global namespace and the logical name of a given file.

3.1. XRootD disk caching proxy

An optional service that can greatly reduce network overhead in an XRootD federation is the XRootD disk caching proxy [?]. Generally this service is meant to run on a node that is geographically close to where the end users are located. The caching proxy can be configured to either cache full files on access, or partial file blocks. When a user reads a file from XRootD, the request transparently goes through the proxy. The data is then written to disk on the cache, as well as served back to the client. Subsequent reads then can access the data from the cache, rather than going back out the Wide Area Network to re-fetch the data from the original end point in the federation.

4. HDFS XRootD Fallback system architecture

The HDFS XRootD Fallback system is composed of three main components—the `hdfs-xrootd-fallback` mechanism, the `hdfs-xrootd-healer` daemon, and an XRootD disk caching proxy; the former is installed on each DataNode in the HDFS cluster, while the later two are co-located in a separate node from the cluster, but also on the site LAN. The following subsections describe the `hdfs-xrootd-fallback` and `hdfs-xrootd-healer` components in more detail.

4.1. hdfs-xrootd-fallback

When a file is accessed from Hadoop, a region of bad blocks is attempted to be read, and there are no good replicas available, the client will normally give up and fail with an I/O exception.

However with `hdfs-xrootd-fallback`, we overloaded the underlying HDFS read call to handle this exception, and rather than bailing out, the handler code will trigger an XRootD client call to locate the region of requested file from the storage federation. This way, a user read request will continue to be served, independent of the health of the file in the local storage cluster. In effect, this functionality provides the local storage an additional level of redundancy that is external to the site, at the XRootD federation level. As a corollary, block level replication factor becomes less important in the local Hadoop cluster.

In order to reduce the network I/O load on the data federation, and also improve read performance at the site, all `hdfs-xrootd-fallback` client accesses are configured to go through the XRootD disk caching proxy. This also provides the added benefit of obtaining pristine new replicas of blocks from files which were previously corrupt in the HDFS cluster, and storing them locally. The `hdfs-xrootd-healer` component discussed in the next subsection will take advantage of the cached blocks, where they will be used to repair the originally corrupt files.

4.2. hdfs-xrootd-healer

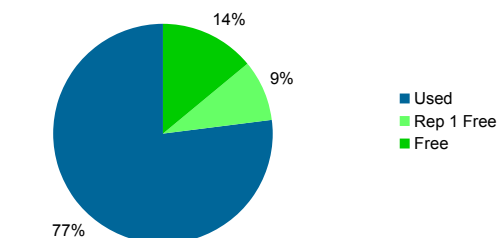
The `hdfs-xrootd-healer` runs periodically on the disk cache node. On each run, the healer queries the Hadoop NameNode to build a list of current broken files. It also examines the cache to discover the list of files that have already been cached. These are all files that had a block corruption when a user tried to read them, and subsequently the cache contains the XRootD-fetched block on disk. The healer then generates a list of files to repair by only considering those that are currently broken, and have a portion cached. The idea is that we don't care to heal broken files that users have never accessed. The cache thus also serves as a way to determine popular files. It is more likely that subsequent user accesses will target a file that was already accessed previously, so these are the files we are interested in repairing.

The healer proceeds to iterate over the list of files to repair, and reads each file from beginning to end into a temporary location in HDFS. The healer simultaneously calculates the full MD5 checksum of the file during the read. Any bad blocks encountered will simply trigger `hdfs-xrootd-fallback` calls, and any data already cached on disk will be reused. Some blocks may be encountered on healer read that were never cached, but this is acceptable, because the fallback will still fetch them remotely as needed, just as it would if it were the first user-triggered fallback. The MD5 checksum is calculated because assuming files were originally transferred into HDFS using GridFTP, the original MD5 checksums were stored along with them in the cluster. The `hdfs-xrootd-healer` script can then compare the newly calculated MD5 checksums with the original, to be absolutely sure the new file is the same, even with the newly included blocks that were fetched remotely via XRootD. Finally, assuming the MD5 checksums match, the healer will then move the newly repaired file from its temporary location into the original location, overwriting the broken file.

5. Deployment at the UCSD Tier-2

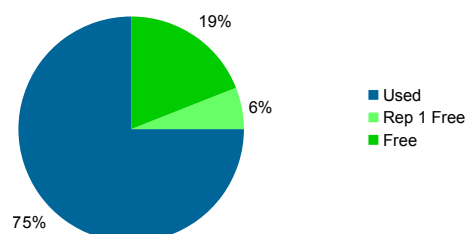
We put the `hdfs-xrootd-fallback` mechanism, along with the XRootD caching proxy into production in April, 2014. Since then the fallback code base has remained the same, modulo some minor bug fixes. We let the system run for a few months without changing the replication factor for any files—we kept replication at the default factor of 2x. Then in August of 2014, we reduced replication to 1x on a subset of CMS data that we know can be retrieved from elsewhere in the AAA Data Federation. The first working version of the `hdfs-xrootd-healer` was completed and put in place in September, 2014. In early 2015, the healer was completely redesigned in order to become production ready for software distribution, based on design improvements as recommended by the Open Science Grid Technology team. We updated the UCSDT2 to use this version in March, 2015. In the subsections to follow, we present a picture of how much disk

Figure 1. Storage status in Oct 2014



Configured Capacity	2.55 PB
DFS Used	1.96 PB
DFS Used %	77 %
Live Nodes	101
Rep 1 Namespace	236 TB

Figure 2. Storage status in Apr 2015



Configured Capacity	3.1 PB
DFS Used	2.32 PB
DFS Used %	75 %
Live Nodes	112
Rep 1 Namespace	190 TB

space the fallback system saves, a snapshot of the fallback and healer working over time, and discuss how we overcame some difficulties we encountered along the way.

5.1. UCSD Tier-2 storage statistics

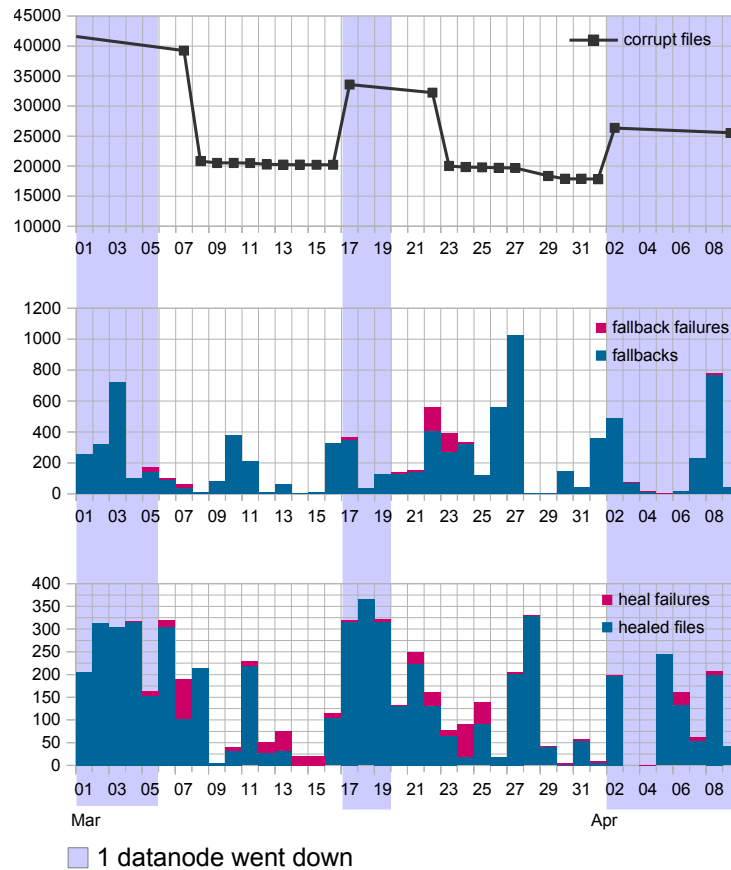
Figure ?? and figure ?? compare storage statistics in our HDFS storage between October, 2014 and April, 2015. "Rep 1 namespace" is the logical file size of all files we have reduced the replication factor to 1x. Focusing on Figure ??, given that the original replication factor was 2x, without hdfs-xrootd-fallback in place, we would have required 472 terabytes of physical disk space to store these files. Therefore, the fallback system is effectively freeing up 236 TB of local disk space. The total percentage of free physical capacity increased from 14% to 23%. Figure ?? reveals that in April, 2015, the impact of hdfs-xrootd-fallback contributes less to our net free storage. This can be explained by circumstances that are unrelated to the hdfs-xrootd-fallback system. First, 11 dead DataNodes were recovered. This increased our total configured storage capacity from 2.55 PB to 3.1 PB. Second, note that the replica 1 namespace has decreased by 46 TB. This is because a subset of files in the namespace became less popular, and are therefore no longer hosted at UCSD. However, the fallback system continues to save us 190 TB of disk space, which is still a significant amount of storage.

5.2. Running HDFS XRootD Fallback

The three plots in figure ?? show the hdfs-xrootd-fallback system running at the UCSD Tier-2 over the period between March 1st and April 9th, 2015. The top plot shows the number of corrupt files in our cluster over time. The middle plot shows how many fallback failovers were triggered each day, or more specifically, how many times a user accessed a corrupt portion of a file, and the hdfs-xrootd-fallback kicked in to fetch the data. Note that sometimes a fallback is not successful. In this case, the software falls back to the default Hadoop behavior of throwing an I/O exception, and the end user application experiences a read failure. These errors are shown in the middle plot as "fallback failures." The bottom plot shows how many files the hdfs-xrootd-healer healed a day. The bottom plot also shows how many corrupt files the healer was unable to repair. The purple areas signify when a single DataNode went down in the cluster, and our node count went from 112 to 111.

There are a number of interesting things that can be observed from figure ?. First, as

Figure 3. Fallback system activity Mar 1 - Apr 9, 2015



expected, the total number of corrupt files in our cluster is decreasing over time. In addition, file healing tends to increase after an increased burst of fallback triggers. In particular this can be seen on March 17th and 27th. This is due to a greater frequency of end users accessing files that have not already been cached, thus these are files that were not previously considered for healing.

One observation that comes as a surprise is what happens when a DataNode goes down. As expected, the number of corrupt files dramatically increases. However, one would expect to see a huge burst of file healing as well, corresponding to users accessing a larger set of corrupt files. Instead, the hdfs-xrootd-healer script tends to saturate at healing a maximum of 350 files a day. We know the healer is hitting some kind of bottleneck, because during periods of a downed node, the script takes multiple days to complete, rather than a few hours on a normal day. This is why there is a lack of data points on the top plot around the time when a node is down. Given a maximum of 350 files, and that our average file size in Hadoop is 4 GB, the healer I/O is:

$$\frac{350 \text{ files}}{1 \text{ day}} \times \frac{4 \text{ GB}}{1 \text{ file}} \times \frac{8192 \text{ Mb}}{1 \text{ GB}} \times \frac{1 \text{ day}}{86400 \text{ sec}} = 133 \text{ Mbps}$$

We can therefore rule out network bottlenecks because our nodes are interconnected with 1 Gbps links. More investigation is needed to determine what is capping the healer speed.

5.3. Details on HDFS XRootD Fallback failures

We are most concerned about `hdfs-xrootd-fallback` failures. When this happens, it causes user jobs to fail. We are also interested in why the healer might fail to repair a file, however this is not as critical, since it does not directly affect the end users. The middle plot in figure ?? shows a large increase in fallback errors around March 22nd. On investigation, we discovered an unexpected behavior in XRootD. The caching proxy passes an opaque URL parameter `/tried=xrootd.t2.ucsd.edu/` when opening a file through the federation to prevent redirections back into UCSD. However, it turns out that some XRootD redirectors ignore this flag. This means that when a fallback gets triggered, the redirector tells the client to look for the data at UCSD. Yet, the reason why fallback was triggered in the first place was because this portion of the file is already corrupt at UCSD, so UCSD will never be able to serve the file. This in turn causes the fallback to bail out, and thus the user job encounters a read error. We were able to put a software patch into the caching proxy side to get around this limitation, and indeed from the middle plot the errors clear up on March 25th. However, a proper fix needs to be applied at the meta-managers in the federation. This fix exists in XRootD 4.2.

6. Conclusions

Overall, the HDFS XRootD Fallback system at the UCSD Tier-2 has been a success. It has allowed us to save about 200 TB of physical disk space. However these are only preliminary results. Given the change in data popularity, we need to revisit the replica 1x namespace. The original 200 TB namespace served as a useful first step in the testing phase, so next we plan to increase the namespace to 1 PB. When choosing a larger subset of files, care must be taken to ensure they are guaranteed to be found elsewhere in the AAA Federation. Another area requiring further investigation is to better understand why the `hdfs-xrootd-healer` is bottlenecking and healing a maximum of 350 files a day. This will become more important when we serve a larger replica 1x namespace, because the impact of a DataNode going down will be greater, so we need to ensure that the healer can keep up and repair files at an optimal rate. One option in particular that we will investigate is the possibility of expanding the healer to run concurrent processes simultaneously. Now that we have shown the HDFS XRootD Fallback system works, the next steps will be to learn how to eliminate the bottlenecks, and better fine tune the system to take full advantage of the global replication made available by the AAA federation.

7. 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] Hadoop [Internet].; [cited 2015 May 16]. Available from: <https://hadoop.apache.org>
- [2] Bauerdick L *et al* 2012 Using XRootd to Federate Regional Storage *J. Phys.: Conf. Series* **396** 042009
- [3] XRootD [Internet].; [cited 2015 May 16]. Available from: <http://xrootd.org>
- [4] Bauerdick L *et al* 2014 XRootd, disk-based, caching proxy for optimization of data access, data placement and data replication *J. Phys.: Conf. Series* **513** 042044