XrootdFS: A Posix Filesystem for Xrootd

Wei Yang, Andrew Bohdan Hanushevsky

SLAC National Accelerator Laboratory

yangw@slac.stanford.edu, abh@slac.stanford.edu

Abstract. XrootdFS is a FUSE based mountable Posix filesystem. It glues all the data servers in a Xrootd storage cluster together and presents it as a single, Posix compliant, multi-user networked filesystem. The ways XrootdFS handles IO operations and metadata operations are specifically designed for the Xrootd's unique redirection mechanism. With XrootdFS we can use the Xrootd storage system as a Grid Storage Element. XrootdFS has been adopted by many Xrootd sites for data management, as well as for data access by applications that do not use the native Xrootd interface.

1. Motivation and history

When we first introduced Xrootd storage system[1] to the LHC, we needed a filesystem interface so that Xrootd system could function as a Grid Storage Element. The result was XrootdFS, a FUSE[2] based mountable Posix filesystem. It glues all the data servers in a Xrootd storage system together and presents it as a single, Posix compliant, multi-user networked filesystem.

Xrootd's unique redirection mechanism requires special handling of IO operations and metadata operations in the XrootdFS. This includes a throttling mechanism to gracefully handle extreme metadata operations; handling of returned results from all data servers in a consistent way; hiding delays of metadata operations in a distributed storage environment; enhancing the performance of concurrent IO by multiple applications; and using an advanced security plugin to ensure secure data access in a multi-user environment.

XrootdFS was briefly mentioned in a CHEP 2009 presentation [3]. Over the last several years XrootdFS has been adopted by many Xrootd sites for data management, as well as data access by applications that were not specifically designed to use the native Xrootd interface. Additional features such as the Xrootd "sss" security module was developed to make it more useful in multi-user environment (see section 6). The original dependence on the Xrootd Composed Name Space [3] was removed, making it easier to use XrootdFS with the Xrootd storage, reliability is also improved. Many of the technical methods mentioned in this paper can also be used to glue together other types (i.e. non-Xrootd) data servers to provide seamless data access.

2. Overview of Xrootd as a cluster of storage

As a distributed storage system, machines in a cluster of Xrootd are divided into two roles: a redirector and a number of data servers in a tree structure (Figure 1)

The redirector is the user facing entrance point of a Xrootd storage cluster. The data servers hold data. Each machines run two daemons: xrootd and cmsd. The cmsd daemons are internal to the Xrootd cluster. They are responsible for forming the cluster, and help the redirector to locate files on data servers. The xrootd daemon on the redirector responds to users request by first locating the file using the cmsd network, and then send the user to the relevant data server's Xrootd daemon for the actual data and metadata operation. The redirector caches the file location information to improve efficiency. Each data server can itself be a cluster of Xrootd storage with its own redirector and data servers. This allows us to build a hierarchy of Xrootd clusters, which is especially useful for storage federation over different domains and wide area network.

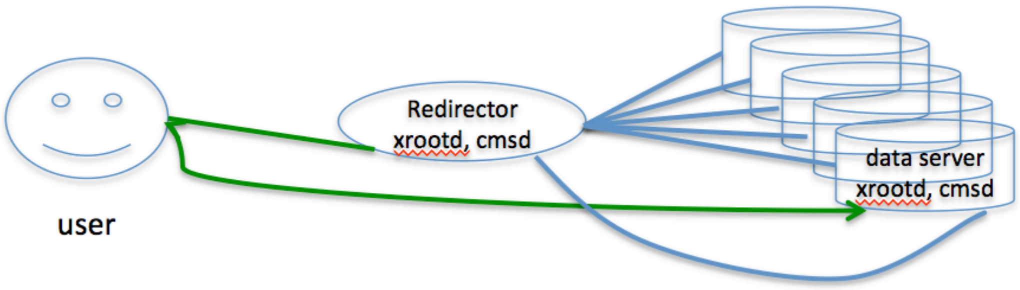

Figure 1. A Xrootd sorage cluster topoloty

For users, after they contact to redirector, they are sent (in Xrootd terminology, redirected) to a specific data server to access files (including creating new files). This happens without having to ask users to do anything extra. From the users point of view, this operation is similar to an Object Store, except that in addition to just provide a simple GET/PUT interface, Xrootd also supports file access from a given offset – those seen in the Posix IO functions.

While low overhead, Xrootd doesn't provide a traditional Posix filesystem to the users. A special set of tools is needed for users to check their file status and obtain an overview of the storage.

3. XrootdFS as a Posix filesystem

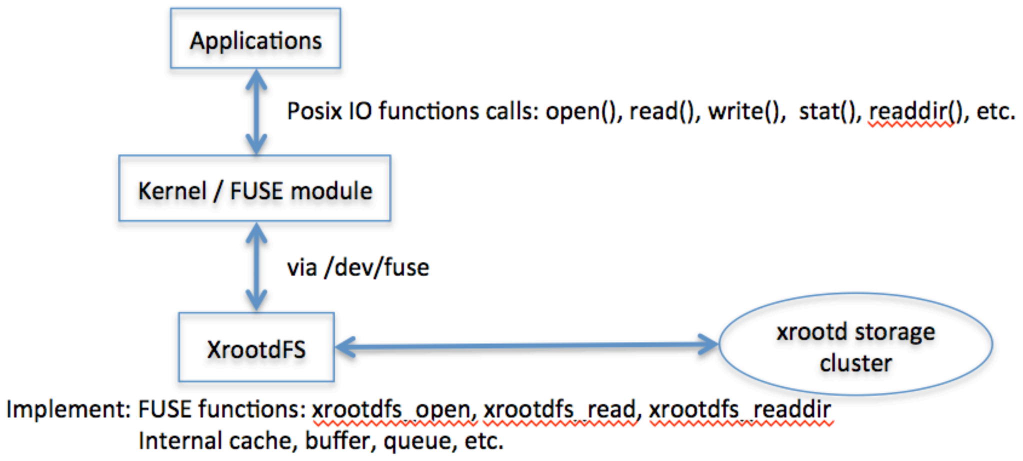
Applications
Posix IO functions calls: open(), read(), write(), stat(), readdir(), etc.
Kernel / FUSE module
via /dev/fuse
XrootdFS
xrootd storage cluster
Implement: FUSE functions: xrootdfs_open, xrootdfs_read, xrootdfs_readdir
Internal cache, buffer, queue, etc.

Figure 2. Relation between Posix compliant Applications and the XrootdFS

XrootdFS mounts a Xrootd cluster under the Unix filesystem tree. It was developed using the FUSE (Filesystem in User Space) framework (see Figure 2). It runs as a non-privileged daemon, and itself is a Xrootd client. Upon starting up, XroodFS automatically discovers the topology of a Xrootd cluster, including multiple layer tree structure. XrootdFS communicates with user through the FUSE kernel module. It turns applications Posix IO operations, including both data and metadata operations to Xrootd Posix IO operations. It also smooths out the difference between how a Posix file systems behave (delay, error message, hardware failure) and how a storage cluster such as Xrootd behave.

There are other software packages/tools that provided somewhat similar functions. Xrootd's Posix Preload Library allows users to first define LD_PRELOAD=/usr/lib64/libXrdPosixPreload.so, and then use most Unix command such as "cat", "cp", "ls" to access the Xrootd storage via the ROOT URL. gfalFS [7] is a tool that allows one to mount a Xrootd cluster to a filesystem directory. However, these two lack XrootdFS' ability to provide a full filesystem view of the whole Xrootd storage cluster. For example, both gfalFS and XrootdFS allow a user to "cd" to a directory in the Xrootd storage via the mount point. A "ls" on gfalFS will not provide the contents of a directory. For this reason, gfalFS also can't perform some common functions such as recursive deletion.

4. Implementation: three categories of functions

XrootdFS follows the high level FUSE framework but itself includes additional functions in order to work well with the Xrootd storage cluster. Those functions fall into three categories: data IO functions, metadata IO functions, query and control functions. The first two categories are implemented via the FUSE functions. The last one is a set of affiliated functions to improve performance and enhance users experience. Most of those FUSE functions provide interfaces to the Posix IO functions in the Linux kernel, which are similar but not identical to the Posix functions available to the users. In most cases there is a one-to-one mapping between the two sets but not always. Parameters passed to those FUSE functions are FUSE data structures, not those data structures in "stdio.h". FUSE based applications such as the XrootdFS are responsible to tailor the implement of those FUSE functions according to the storage systems.

Most FUSE functions we implemented start with prefix xrootdfs_, such as xrootdfs_open(). We will simply refer they as _open() later in this paper

4.1. Data IO functions

This includes _mknod() (equivalent of Posix creat()), _open(), _read(), _write(), _release() (similar to Posix close()), _truncate()/_ftruncate(), _seek(), etc. While similar to the standard Posix IO function, the file descriptors used in these functions are internal to the XrootdFS daemon. They don't corespond to the file descriptors returned by the Linux kernel to user applications. Issues that we need to address includes:

- The _release() function is supposed to close the file, but it immediately returns and the actual Xrootd file closing happens asynchrony, ofter with very short delay. In theory, bad things could happen during this short delay. In practice, we have never observed any issues related to this delay.
- Unless using the very recent combination of Linux kernel and FUSE releases, the linux kernel breaks large write requests into 4KB blocks. This has a noticeable negative impact to the performance of large sequential writes. To compensate this behavior, a 128KB cache is used to capture sequential writes.
- A user may open a file and leave it idle for a long time. The Xrootd client library used by the XrootdFS will close the corresponding Xrootd file and network connection. When the user become active on this file again, the Xrootd file will be opened again, transparent to both the user and the XrootdFS daemon.

4.2. Metadata IO functions

These include _readdir(), _getxattr(), _unlink(), _rename(), etc. One of the important requirements of these functions is to be able to put together information scatted on the data servers. As the function that maps to Linux kernel readdir(), _readdir() includes steps to find the corresponding directory on the data servers, get dir_entry from all of them, merge and remove redundant ones and return a complete list of unique dir_entry to the users. It also needs to merge the error (err_no) from all data servers and return a reasonable error_no to the user. The merged dir_entry's are saved in a cache with an expiration time for later use.

The _getxattr() function is essentially a stat() function. While not need to piece together file/directory info from all data servers, it needs to deal with the Xrootd file locate delay – the Xrootd redirector uses real time query/respond mechanism to locate a file. A data server will immediately respond to Xrootd redirector's file locate query if it has the file. If the file doesn't exist in the Xrootd cluster, no data server will respond. After a period (default 5 seconds), the Xrootd redirector will timeout and respond to the user with a file not exist message. This mechanism has been proved to be very scalable. But in many cases, XrootdFS needs to hide this delay. For example, if user "cd" to a directory in XrootdFS and run an application, the application may first search in the current directory for libxyz.so before it searches /usr/lib64. Giving that an application may involve many shared libraries, a 5 second waiting could end up delaying the execution of the application by minutes. To hide this delay, XrootdFS contacts each and every data server and ask for a stat(). This approach, while solving the delay problem (and other problems that will be discussed later in this paper), is obviously costly – just imagine a user does a "ls –l directory" or a "find /directory". Luckily, a user "ls –l directory" involves a _readdir() first, followed by _getxattr() on each and every dir_entry retuned by _readdir(). Since _readdir() already saved the dir_entry to a cache, we just need _getxattr() to check the cache. If the cache has the dir_entry, then we can use the default Xrootd file locate mechanism instead of the brute force mechanism.

_rename() and _unlink() (correspond to the Unix "mv" and "rm"/"rmdir") are two other examples that we use brute force. This helps to guarantee consistent results across all data servers. We also use brute force mechanism for _statvfs() (unix "df"), given that its usage is not as often as other commands.

Metadata operations, especially those that require brute force mechanism, can impose high level of random disk IO on the data servers, which negatively impact their performance in serving data. Also in a busy client machine with multiple users using XrootdFS, FUSE will create a thread for each and every metadata operation, significantly increase the number of threads and memory usage of the XrootdFS. For these reasons, It is desired to put a limit on how many concurrent metadata operations a XrootdFS can perform against all data servers.

XrootdFS implement a FIFO queue for internal usage. Brute force mechanism used by functions such as _readdir() creates corresponding tasks against all data servers. These tasks are appended to the end of the queue, and the calling functions will wait for them to complete. A number of task workers (threads) will pickup tasks from the head of the queue. When a task is completed, the worker sends a condition signal so that the caller can pickup the result and continue. Error condition is included in the returned results to the caller. The caller then merges the error code in an appropriate way and return to the user.

The brute force mechanism provides to the XrootdFS a way to detect abnormal behavior of the Xrootd cluster. For example, if a data server is offline, all metadata operations against the data server will hang. This is often the desired behavior because this tells XrootdFS users and administrators that the systems is in trouble and needs to be fixed. The alternative, which allows XrootdFS or other Xrootd client to continue, often result in inconsistency issues that can lead to long investigation and manual fixing.

4.3. Query and control functions

These functions allow us to access a running XrootdFS and control some of its behavior. This is done via FUSE's extended attributes interface. A user can use Unix command "getfattr" against a mounted XrootdFS filesystem. The following extended attributes are available for query:

1. "xrootdfs.fs.workers": retune the number of task workers. The default is 4.
2. "xrootdfs.fs.dataserverlist": returns a list of data servers currently known to the XrootdFS.
3. "xrootdfs.file.permission": returns the access permission of the querying user.

Privileged users (such as root) can also use Unix "setfattr" command to add or remove task workers - if the number of task workers is set to zero, metadata operations will freeze until a worker is added. Privileged users can also use “setfattr” to ask XrootdFS to refresh the internal list of data servers (this can also be done by sending a Unix signal “USR2” to the XrootdFS daemon).

FUSE options are also available to XrootdFS when it starts. XrootdFS pre-sets a number of FUSE options values such as attr_timeout, entry_timeout, negative_timeout to optimize the performance. Other FUSE options such as “big_writes” and “direct_io” can also be set at start up time to prevent kernel from breaking large sequential write to 4KB writes, and bypass operating system page cache for IO. XrootdFS itself have a few options. They can be found in its man page or by just type XrootdFS with no argument.

5. XrootdFS performance

Although IO performance is not the solo goal for XrootdFS, it is still useful to understand and improve the IO performance. Several factors impact the performance of the XrootdFS. This includes the overhead of using the FUSE framework, the efficiency of the Xroot protocol via the Xrootd Posix API and the storage system hardware (e.g. the hard disk). In addition, the metadata operation’s performance is also related to the topology of the storage system.

The following measurements were performed using the setup of ATLAS Tier 2 at the SLAC National Accelerator Laboratory. The Xrootd storage system has one redirector and 13 data servers. All of them have 10 Gb/s NIC. All have XFS filesystem. A data server can have 96 to 180 HDDs in groups of raid 6 configuration. The client machine (the machine the runs XrootdFS) has 8 CPU cores, 48GB memory and a 10Gb/s NIC,

5.1. Sequential IO performance

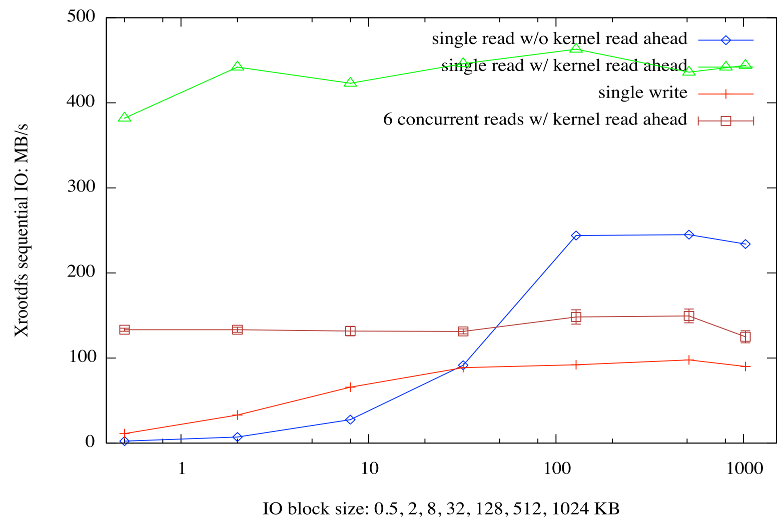

Figure 3 Sequential IO performance. In the 6 concurrent reads test, the result is the average of all 6 reads.

Figure 3 shows the single sequential read and single sequential write performance under various setup, as well as the performance of 6 concurrent sequential reads. The performance is acceptable in most use cases. It, however, shows the gap between XrootdFS performance and the maximum performance allowed by the 10Gb/s network.

5.2. Overhead of XrootdFS

The overhead of XrootdFS mainly come from two areas: the FUSE framework, and choice of using the Xrootd’s Posix API instead of native Xroot IO API. Using Xrootd’s Posix API make it easy for XrootdFS to combine return codes and error codes from all data servers. But we then can’t use features such as asynchronous IO and multiple TCP streams, which are only available in the native Xroot IO API. We made this trade off out of the consideration that XrootdFS will mainly be used in LAN environment and mostly by applications that do random/sparse reading and whole file sequential writing. Compare to the WAN data transfer use case, the benefit of those extra features is less important. Also a single XrootdFS instance doesn't need maximum performance due to other limitations of a single machine.

To understand these overheads, we measured network IO performance in several setups: use Unix "cp" to move files between local file system and XrootdFS (this includes overhead of both FUSE and Xrootd Posix API); use "cp" with Xrootd's Posix preload library to move files between local file system and the Xrootd storage directly (only Xrootd Posix IO involve); and use "xrdcp" (which use native Xroot IO functions, including asynchronous IO) to move files between local file systems and the Xrootd storage. In all setups, we use medium-large sized file (2.5GB) and we eliminate the impact of filesystem page cache where needed. Also note that "cp" uses 32KB block size, while xrdcp use 16MB block size.

The following measurements provide comparison regarding the overhead on reads:

- "cp" on XrootdFS with kernel read ahead disabled (FUSE option "direct_io"): ~117MB/s
- "cp" on XrootdFS with kernel read ahead enabled: 457MB/s
- "cp" via Xrootd Posix Preload Library (doesn't go through kernel, no read ahead): 246MB/s
- "xrdcp": 1.1GB/s

The following measurements provide comparison regarding the overhead on writes:

- "cp" on XrootdFS: 143MB/s
- "cp" via Xrootd Posix Preload Library: 128MB/s
- "xrdcp": 763MB/s

Note that writing via XrootdFS outperforms writing via Xrootd Posix Preload Library because XrootdFS has an internal cache that, when possible, assembles small sequential writes (such as those 32KB blocks by "cp") and make them a single 128KB write.

5.3. Real use case performance

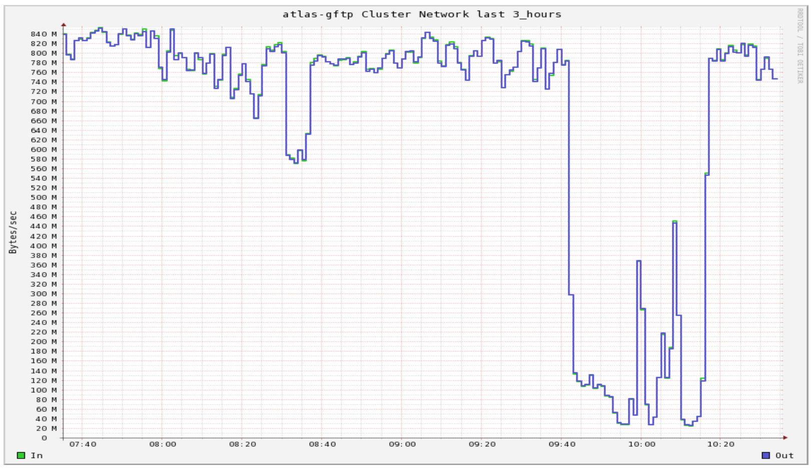

Figure 4. Performance of a GridFTP server running on XrootdFS.

Figure 4 shows the network traffic when using the above client machine as a GridFTP server at the SLAC's ATLAS Tier 2. It is a Ganglia monitoring plot of the network traffic in and out of the NIC. Since all data go into the NIC will also go out of the NIC, the network in and network out overlap most of the time. The GridFTP access the Xrootd storage via the XrootdFS. The GridFTP reads and writes multiple files simultaneously. Each GridFTP process may use several TCP streams to access data. The ratio of read and write variant from time to time.

5.4. Performance of the metadata operations

The metadata operation's performance is related to the number of data servers in the Xrootd storage cluster, how many concurrent XrootdFS worker threads are used to performance the metadata operations, and the overhead of user applications themselves. The most important parameter ATLAS cares is the file deletion rate. At the SLAC ATLAS Tier 2, there are 13 data servers. On a virtual machine running XrootdFS and BeStMan [4] [5] SRM [6], we use 24 task workers threads, and we can deleted file as ~25 Hz.

6. Security

In principle, XrootdFS can use any Xrootd security protocols, for example, "unix", "gsi", "sss", or nothing. The "unix" security protocol passes the username (not uid) of the user that runs the XrootdFS instance to the Xrootd cluster. This means all users using that particular XrootdFS mount point are treated the same by the Xrootd cluster. This is OK for a single user environment or a small group of users but may not be OK in larger multi-user environment. Similar issue exists in using "gsi" security protocol with XrootdFS.

The "sss" (simple shared secret) protocol allows XrootdFS to pass the usernames (not uid) of the actual users to the Xrootd server, making it a true multi-user environment. Setting up "sss' requires Xrootd data servers and XrootdFS instances to share a predefined "sss" key. Doing so obviously means the Xrootd servers must trust the XrootdFS instances. The Xrootd document and XrootdFS man page provide detail instructions on how to setup the "sss" security with the Xrootd data servers and with the XrootdFS instances.

7. Limitations, rarely used features, and How-To

As an application that is built using the Xrootd client library, XrootdFS inherits the limitation of Xrootd itself. Xrootd doesn't support mtime and atime of file, as well as file locking. Xrootd supports Access Control List (ACL) but not Posix file ownership and permission. XrootdFS naturally doesn't support any of them. For access control list, XrootdFS supports querying ACL via querying of an extended attribute. A user will only see the ACL's related to himself/herself.

A rarely used configuration feature in Xrootd is the Composite Name Space (CNS). CNS is an auxiliary Xrootd server that is not part of the Xrootd storage cluster, but it hosts the complete file system namespace metadata info. This is particularly useful when part of the storage in an Xrootd storage cluster doesn't provide an easy to access name space info, for example, a tape system. In a conventional configuration, XrootdFS gets the name space info from the Xrootd storage cluster itself. XrootdFS can also be configured to get the name space info from the CNS.

Although XrootdFS can be started from the command line, or being put in /etc/fstab, the most useful way is the put it under the control of AUTOFS. This is documented in the XrootdFS man page.

8. Conclusion

With XrootdFS, we can use the Xrootd storage cluster as a Posix compliant networked filesystem. XrootdFS has been used in many US universities and labs that support the ATLAS experiment [4]. Over the last several years, it has been proved to be reliable. Other than a few limitations inherited from the Xrootd itself, most application designed for a Posix filesystem should just work on XrootdFS/Xrootd. Most of the XrootdFS performance overheads come from the FUSE framework and the Xrootd Posix API. But since the XrootdFS runs on a client machine, and there can be many of them, XrootdFS' IO and metadata performance can satisfy most use cases except a few extreme ones.

References

[1] Document and software of the Xrootd: http://www.xrootd.org
[2] Document and software of the FUSE at the GitHUB: http://github.com/libfuse/libfuse
[3] W. Yang, A. Hanushvesky, A, Sim, F. Furano, Scala as a Full-Fledged LHC Grid SE, http://indico.cern.ch/event/35523/contributions/840117/attachments/705427/968450/CHEP09ABH112.pdf

[4] Open Science Grid Twiki on installing BeStMan Xrootd storage as a Grid Storage Element: https://twiki.grid.iu.edu/bin/view/Documentation/Release3/InstallBestmanXrootdSE
[5] A. Sim, A. Shoshani, J. Gu, V. Natarajan CHEP 2009 poster, https://sdm.lbl.gov/bestman/docs/BeStMan_Poster_090816.pdf
[6] A. Shoshani, A. Sim, J. Gu, "Storage Resource Managers: Middleware components for Grid Storage", Proceedings of Nineteenth IEEE Symposium on Mass Storage Systems, 2002.
[7] gfalFS: https://dmc.web.cern.ch/projects/gfalfs