XrootdFS

A Posix File system for XrootD

Abstract

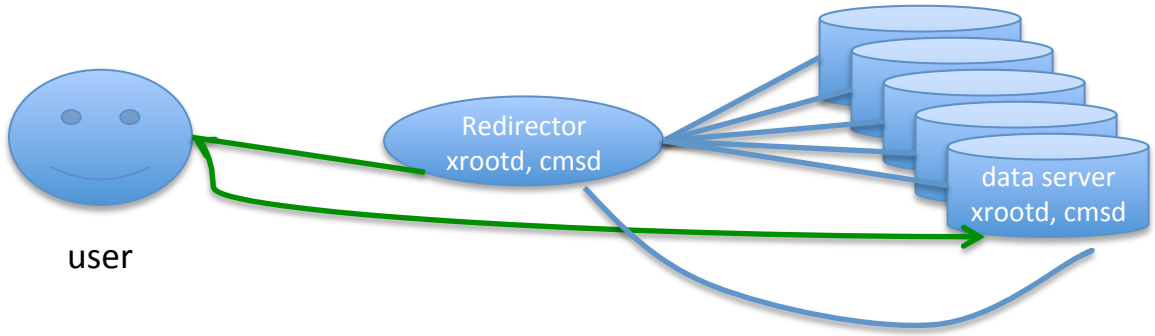
When we first introduced XRootD storage system to the LHC, we needed a filesystem interface so that XRootD system could function as a Grid Storage Element. The result was XRootDfs, a FUSE based mountable posix filesystem. It glues all the data servers in a XRootD storage system together and presents 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, including storage media latency; 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. Over the last several years XRootDfs have 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. Many of the technical methods mentioned above can also be used to glue together other types (i.e. non-XRootD) data servers to provide seamless data access.

Use Xrootd Storage as a Single Posix File System

- Glue a distributed Xrootd storage together
- Multi-user networked filesystem
- Posix compliant:
  - Run unix commands: cd, ls, cp, cat, mv, rm, mkdir, find …
  - Applications that work on posix filesystem: scp/sftp, bbcp, GridFTP, Bestman SRM …
  - Or posix IO functions in your application: stat(), open(), close(), read(), write(), unlink(), etc.

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:



The redirector is the user facing entrance point of the 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 corresponding data server's xrootd daemon for the actual data operation. The redirector caches the file location information to improve the efficiency. Each data server can itself be a cluster of xrootd with its own redirector and data servers. This allow us to build a hierarchy of xrootd clusters, which is especially useful for storage federation over difference domains and wide area network.

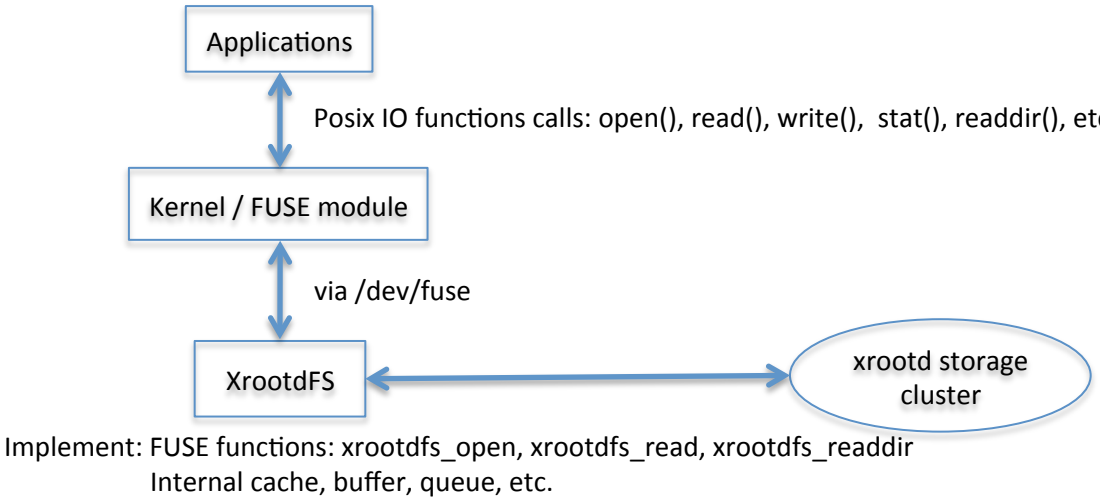
For users, after they contact to redirector, they are sent (in xrootd terminology, redirected) to a specific data server to access files (including creating new files). This happens without having to ask users to do anything extra. From the users point of view, this operation is similar an Object Store,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

While low overhead, this architecture (including the Object Stores) 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 get an overview of the storage.

XrootdFS as a Posix filesystem for Xrootd storage cluster

XrootdFS provides a posix filesystem to the xrootd cluster. It is mounted under the Unix filesystem tree and supports posix IO operations. Applications that are developed to access data via the posix IO function will be able to function on top of it, and regard the xrootd storage cluster as yet another filesystem.

XrootdFS is developed using the FUSE (Filesystem in User Space) framework. It is a non-privileged daemon running on a unix machine, and itself is a xrootd client. It understands the structure of a xrootd, including multiple layers in the tree structure. XrootdFS interaction with user requests goes through the FUSE kernel module. It turns users posix IO operations, including both data and metadata operation to xrootd Posix IO operations.



XrootdFS follows the FUSE framework but itself includes additional functions in order to work well with the Xrootd storage cluster. Those functions falls into three categories: data IO functions, metadata IO functions, query and control functions.

The XrootdFS implementation not only need to translate posix IO operations to the Xrootd IO operations, itself also needs to smooth out the difference between how a posix file systems behave (delay, error message, hardware failure) and how a storage cluster such as xrootd behave.

Implementation: three categories of functions

The three categories of functions include FUSE functions, and affiliated functions to improve performance/enhance users experience. FUSE functions are similar to POSIX IO functions in Linux kernel (which is 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. Applications developed using FUSE are responsible to tailor the implement of the FUSE functions according to the storage systems. Parameters passed to those FUSE functions are FUSE data structures, not those data structures in "stdio.h". Most FUSE functions we need to implement starts with prefix xrootdfs_, such as xrootdfs_open(). We will simply refer they as _open() here.

XrootdFS data IO functions includes _mknod() (equivalent of posix creat()), _open(), _read(), _write(), _release() (similar to posix close()), _truncate()/ _ftruncate(), _seek(), etc. File descriptor used in these functions are internal to the XrootdFS daemon. They don't corespond to the file descriptor returned by Linux kernel to application; 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; Until very recent combination of Linux kernel and FUSE releases, the linux kernel breaks large write reuqests 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 XrootdFS daemon.

Metadata IO functions includes _readdir(), _getxattr(), _unlink(), _rename(), etc. One of the important requirement in these functions is to be able to put together information scatted on the data servers. As the function that maps to those in 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 to the user. It also needs to merge the error 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 – 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 OS may first search the current directory for libxyz.so before it searchs /usr/lib64. Giving the a application may involves many shared libraries, a 5 second waiting will delay the execution of the application by minutes. To hide this delay, XrootdFS contact each and every data server and ask for a stat(). This brutal force approach is obviously costly – just imagine 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 brutal force mechanism.

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

Metadata operations, especially those that require brutal force mechanism, can impose high level of random IO to 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, we need to put a limit on how many concurrent metadata operation a XrootdFS can perform against all data servers.

XrootdFS implement a FIFO queue for internal usage. Tasks such as readdir(), stat(), unlink(), rename() (these are all xrootd IO functions resulted in the above brutal force mechanism) are appended to the tail of this queue by the corresponding caller function _readdir(), _getxattr(), _unlink(), _rename(), etc. A number of task workers will pickup tasks from the head of the queue. When a task is completed, the worker send a condition signal so that the caller will pickup the result and continue. Error condition is included in the returned results.

The brutal force mechanism provides us 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 tell XrootdFS users and administrators that the systems is in trouble and needs care. The alternative, which allows XrootdFS or other xrootd client to continue, often result in (in)consistency issues that can lead to long investigation and manual fixing.

Query and Control allows 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": retune 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. Though this can also be done by sending a unix signal "USR2" to the XrootdFS daemon.

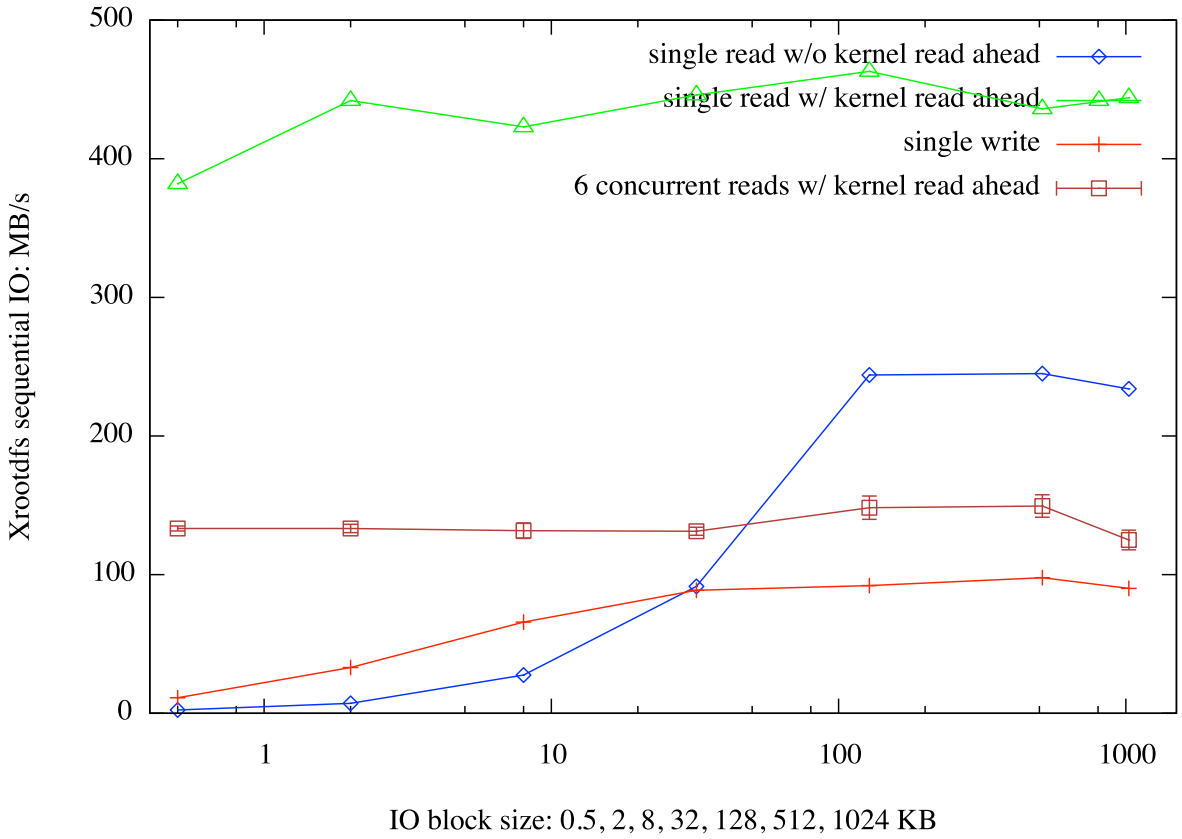
FUSE options are also available to XrootdFS when it starts. XrootdFS set a number of default values to FUSE options such as dirattr_time?, fileattr_time? Negative_attr_time? Other FUSE options such as "big_writes" and "direct_io" can also be set to prevent kernel from breaking large sequential write to 4KB writes, and bypass operating system page cache for IO. XrootdFS itself have a few options which can be found in the its man page or by just typing XrootdFS with no argument.

XrootdFS IO performance

Performance measurement environment:

1. SLAC ATLAS Tier 2 Xrootd storage: 13 data servers, each of them 1x 10Gbps NIC. 100-180 HDDs in groups of raid 6.
2. Client/XrootdFS, 8-core, 48GB, 10Gbps NIC

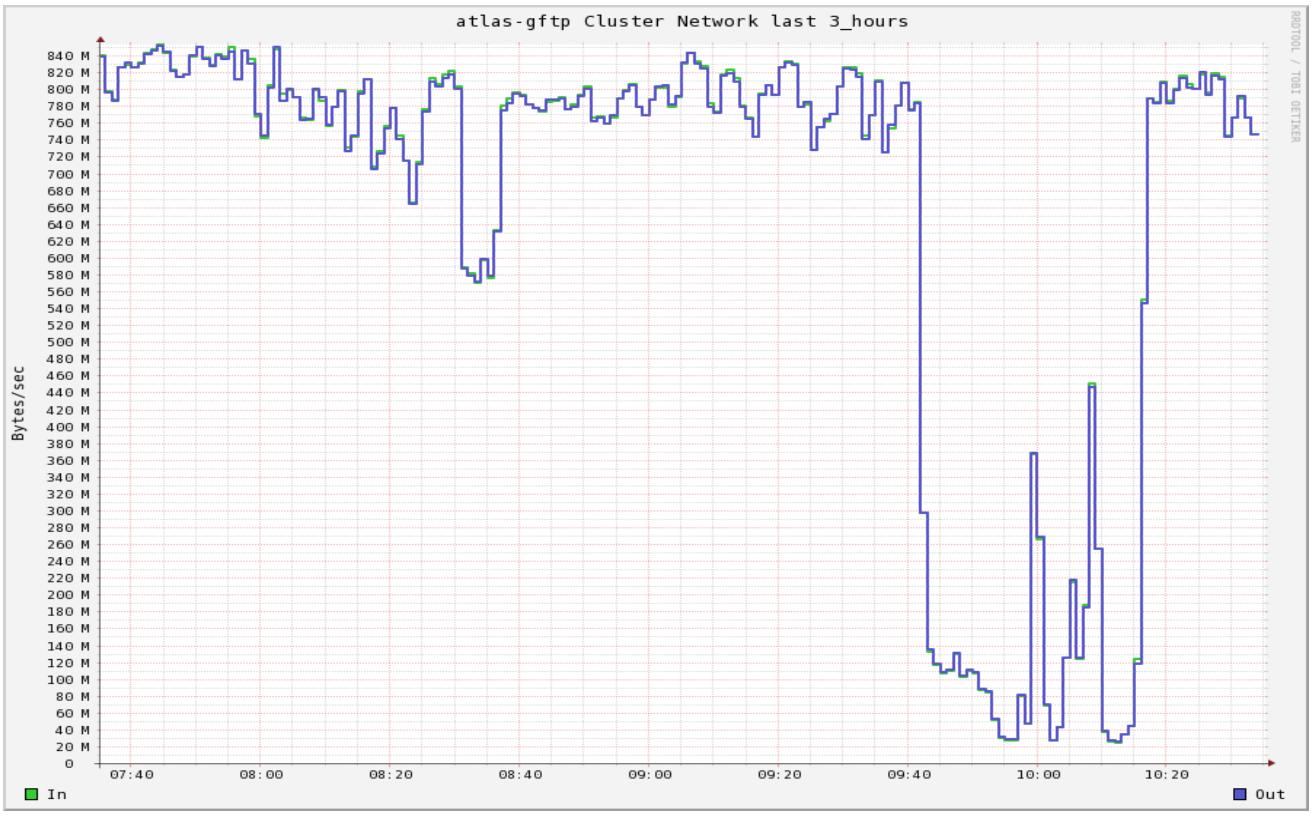
Sequential IO performance:



Metadata operations performance depend:

1. more on the number of data servers and their performance
2. Less on XrootdFS and FUSE overhead.
3. At SLAC Tier 2, deletion rate via XrootdFS is ~25Hz

SLAC Tier 2 GridFTP accesses Xrootd storage via XrootdFS



How to use

It is documented in “man xrootdfs”. Examples:

- run from command line with debugging output

  ```
  xrootdfs -d -o rdr=root://rdr:port//data,uid=daemon /mnt
  ```

- use with autofs

  add a line to /etc/auto.master

  ```
  /- /etc/auto.fuse
  ```

  create /etc/auto.fuse with the following one line

  ```
  /mnt -fstype=fuse,uid=2,rdr=root://rdr\:port//data :xrootdfs.sh
  ```

  create script /usr/bin/xrootdfs.sh (and set the +x bit)

  ```
  #!/bin/sh
  exec /usr/bin/xrootdfs $@ >/dev/null 2>&1
  ```

Security

XrootdFS can use any xrootd security plugins, for example, “unix”, “gsi”, “sss”, none. “unix” security plugin pass the username (not uid) of the user that runs the XrootdFS instance to the Xrootd cluster. This means everyone using that particular XrootdFS mount point as treated the same by the Xrootd cluster. This is OK for a single user environment or a small group of users. Similar issue exists in using “gsi” security plugin with XrootdFS.

“sss” allows XrootdFS to pass the actual usernames (not uid) to the xrootd server, making it a true mult-user environment. “sss’ setup requires data servers and XrootdFS instances to share a predefined “sss” key. This means the servers must trust the XrootdFS.

Other features and limitations

- Support Composite Name Space (CNS) – rarely used
  - CNS hosts the complete file system metadata info, it is an auxiliary xrootd server that is not part of the cluster.
  - Provide an overview of the Xrootd storage including tapes
- Do not support things unavailable in Xrootd
  - changing mtime, atime, etc.
  - file locking
  - File ownership – Xrootd supports ACL only