1 Introduction

In its basic form, **xrootd** is a component based system, implemented as daemons, that delivers data across a network. It is similar to many distributed network-based file systems (e.g. afs, gpfs, hdfs, lustre, nfs, etc). However, it has some significant differences that make it stand apart:

- high performance in a small memory footprint with low CPU overhead,
- uses a plug-in architecture to support a wide variety of environments,
- runs in user-mode (i.e. not root),
- uses multiple authentication mechanisms at the same time,
- clusters local and remote data servers using a federation model,
- can automatically copy data between arbitrary locations,
- decouples logical and physical name spaces,
- integrates a proxy service,
- built-in checksum support,
- offers detailed monitoring, and
- allows multi-source transfers.

Figure 1 shows **xrootd** components and how they related to various functions provided by the system. This guide describes these components, how they interact, and offers guidance along with examples on how to configure the system and build effective data delivery environments.

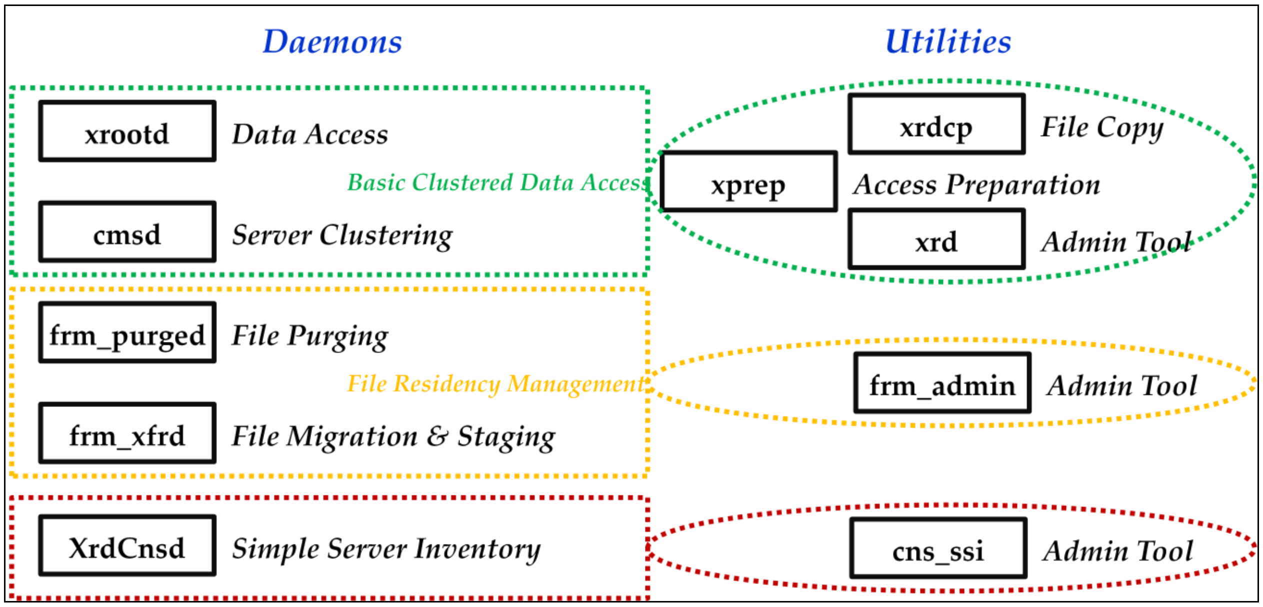


Figure 1 the xrootd System

In order to fully understand **xrootd**, the following sections discuss the unusual features that **xrootd** provides, how they impact its use, and why they are important.

2 The Background

In 2001 the BaBar experiment, a collaboration of 400 physicists from over 9 countries studying the relationship between matter and anti-matter, decided to switch their data analysis framework from Objectivity/DB database system to the Root framework. The new analysis framework relied largely on structured flat files either locally accessible to a compute node or served through a network-based file server. Flat files were seen as a great and absolutely essential simplification to the experiment's massive data handling and distribution requirements.

While data handling and distribution would be simplified, the problem area shifted into finding a file server solution that could scale to the peta-bytes of data the experiment would generate and handle peak loads from a thousand or more simultaneous analysis jobs. The nature of the load was driven by the peculiarities of the framework which would perform several meta-data operations on dozens of files per job prior to commencing analysis. This meant that any new data access system needed to sustain thousands of transactions per second, cluster hundreds of physical data servers just to handle the amount of data, and recover gracefully from failures expected when a massive amount of hardware is deployed. The three main system requirements: low latency, scaling, and recoverability, all needed to be met simultaneously; otherwise, it was clear that the BaBar collaboration would not be able to perform data analysis in a timely manner. Such an event would doom the experiment and the investment of hundreds of millions of dollars. A search of systems available in 2001 reveals now, as it did then, that no affordable commercial solution existed that could meet all three requirements. Hence, the stage was set for the development of **xrootd**.

2.1 Fast Forward

Today, **xrootd** is being deployed in environments and for uses that were never conceived in 2001. This speaks well for the system's adaptability but the underlying reason is that the system can simultaneously meet its three fundamental objectives (low latency, scaling, and recoverability) together with a feature set that that is virtually unrivaled.

Some of the current deployments include

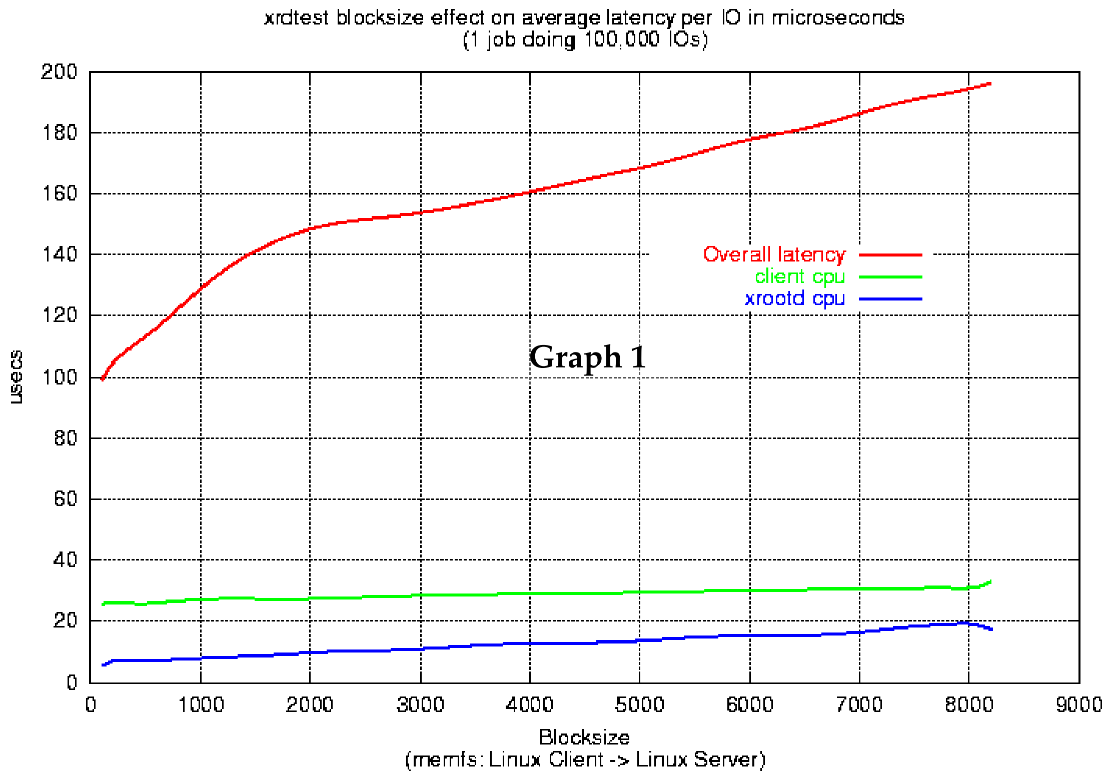
- The ALICE LHC experiment uses **xrootd** to provide world-wide data access by clustering storage over 60 autonomous sites in 20 countries.
- The US Atlas and CMS LHC experiments are using **xrootd** to create regional clustered data repositories consisting of dozens of sites to make peta-bytes of data available for on-demand copying as well as real-time data-access across the WAN.
- The Fermi-GLAST astrophysics experiment is using **xrootd** as a key component to perform timely data analysis and data reconstruction at SLAC and simulation at IN2P3 of data down-linked from its gamma-ray satellite observatory.
- The Star experiment at Brookhaven National Laboratory is using **xrootd** to augment data storage by clustering over 600 batch server nodes and making their storage uniformly available to all batch jobs.
- CERN/IT is using **xrootd** as the basis for their massive EOS data delivery system to serve analysis jobs across all experiments, initially starting with ATLAS and CMS.
- The widely used Parallel Root Facility (PROOF), a Hadoop-like system for the Root framework, uses **xrootd** as a fundamental part of its data access infrastructure and leverages the **xrootd** framework for its job scheduling system.
- The Large Synoptic Sky Telescope (LSST) experiment is using **xrootd** to cluster hundreds of mySQL servers to perform complex SQL queries on peta-bytes of data. The prototype system has recently passed its data challenge tests.

3 The Feature Set

3.1 High Performance with Low Resource Requirements

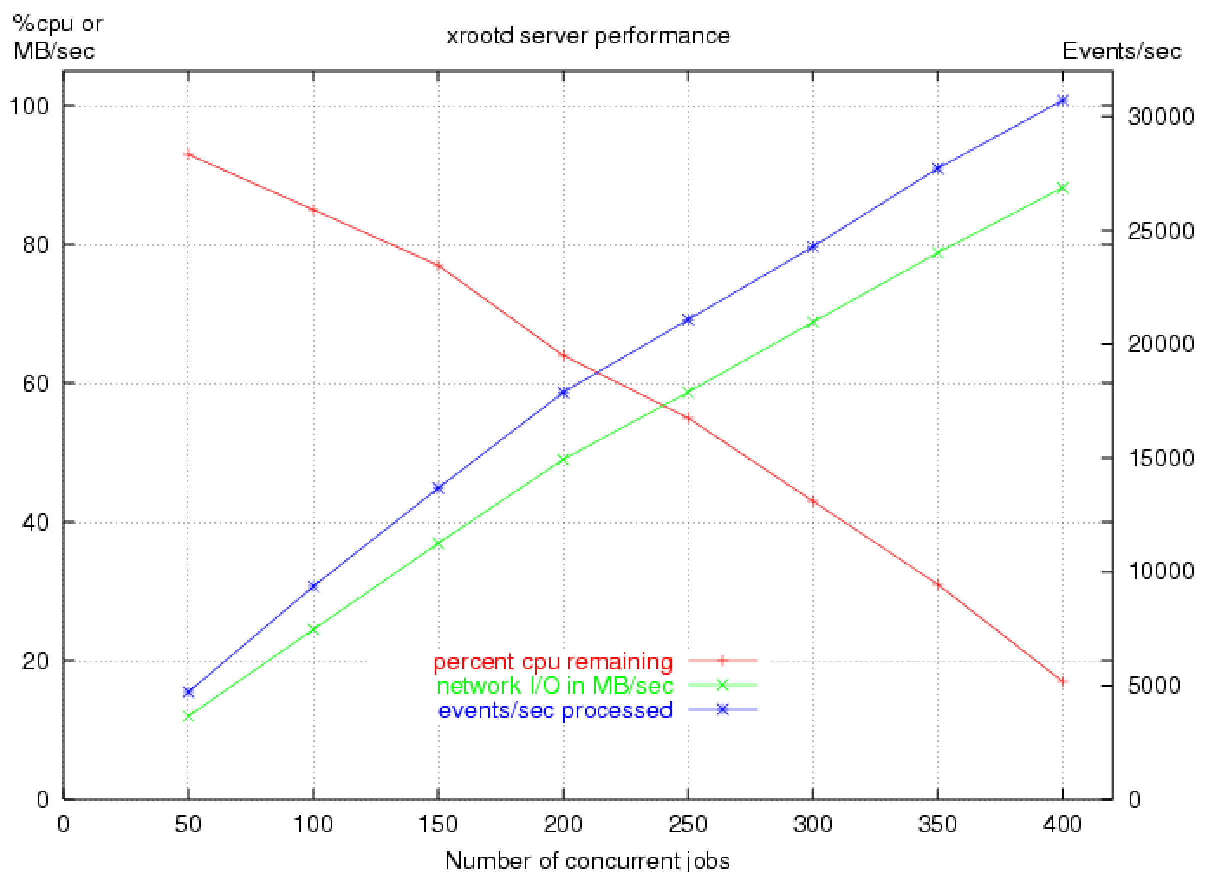
High performance with resource efficiency is very strong point for **xrootd**. Special attention has been paid to minimizing the use of memory as well as CPU. Typically, an **xrootd** server providing read access to data will need less that 100MB of main memory and usually less than 300MB when a write load is added. CPU utilization is linearly proportional to usage and typically adds a few percentage points over the same native operations.

Tests using machines[1] considered under-powered today show impressively low latency with minimal CPU overhead. Referring the Graph 1, you can see that added latency is less than 10µs meaning that network or disk latency dominates (i.e. **xrootd** provides the full performance of the underlying hardware).



Graph 2 shows the overall through-put as the number of simultaneous jobs increase. The most significant feature that the performance graph shows is that **xrootd** scales *linearly* with the number of clients (as indicated by the "CPU remaining" line). Linear scalability means that the number of clients that a single **xrootd** server can support is not limited by the server software but by factors such as memory, CPU, disk speed, and network interface. Linear scaling also explains why network bandwidth utilization and the number of events per seconds uniformly increase as more clients use the server.

[1] Sunfire V20Z with AMD Opteron 244 CPU's, each at 1.8GHz, 2 GB DDR SDRAM with a 333MHz clock rate, 2 MB L2 cache (1MB per processor), On-board gigabit Ethernet network interface, running Solaris 10.



3.1.1 Why low resource usage is important

As machines become more powerful while decreasing in price, it becomes easy to forget that resources are still limited. Witness the explosion of memory hungry desktop software as desktops increased in performance while the price dropped. Unfortunately, that also meant that fewer programs could run at the same time. Perhaps not so important for home computers but critical in batch worker environments where multiple jobs are competing for the same resources. Here, better performance at a lower price point is always taken as an opportunity to increase the useful workload.

In these environments the power of a small memory footprint along with low CPU utilization for support services becomes critical. With **xrootd**'s small resource requirements the trend is to use **xrootd** to leverage batch node disk resources to offer better data sharing opportunities while leveling disk utilization across those nodes; something that is difficult if not impossible with other clustered file systems.

3.2 Plug-In Architecture

The major pieces of **xrootd** pictured in Figure 1 are composed of discreet components. A default set of components is packaged with **xrootd** that usually suites most environments. However, **xrootd** is architected so that one or more of these components can be replaced with custom implementations. The plug-in architecture makes **xrootd** suitable for interfacing with a wide variety of data storage systems and is especially useful for network access to storage systems that do not provide adequate or secure network connectivity. Once the appropriate plug-in is created to handle a particular storage solution, all of the **xrootd** mechanisms not replaced are enabled for that storage solution.

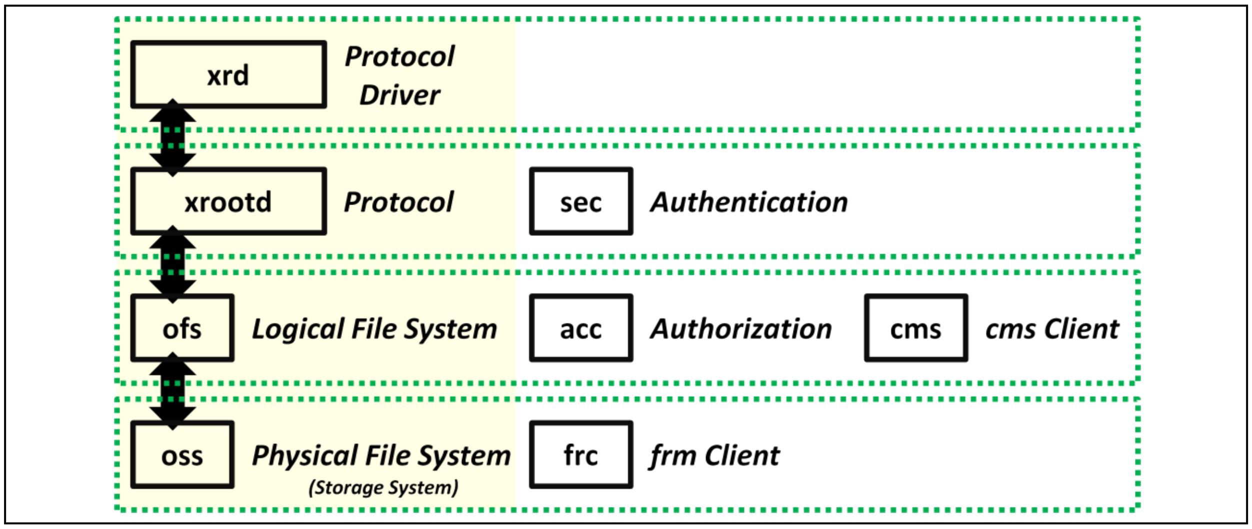


Figure 2: xrootd Components (aka plug-ins)

For instance, assume you want to provide WAN access with x.509 security for the Hadoop File System (HDFS). HDFS provides neither while **xrootd** provides both. It is a relatively simple matter to write a storage system plug-in that interfaces with HDFS to automatically provide secure WAN access to HDFS. In fact, such a plug-in already exists.

The plug-in architecture allows you to customize the areas of security (authentication as well as authorization), access protocols, clustering, name space handling, and file residency using self-contained modules constructed as shared library plug-ins. This makes **xrootd** applicable to a virtually limitless range of environments.

3.2.1 Why a plug-in architecture is important

The key feature of a plug-in architecture is that it allows you to leverage a substantial software investment. This is akin to bringing a house up to new building codes. There is no need to rebuild the house; you merely replace the outdated components. This has been shown again and again to be a powerful strategy as shown by the many plug-ins written for **xrootd** to accommodate evolving requirements over the years. Plug-ins enable a speedy software evolution with minimal additional investment.

3.3 User-Mode Execution

The **xrootd** system requires that you run the various components as a non-root user. This is done largely to make sure that should a compromise of any component occur, damage is limited to privileges afforded to the user running **xrootd**. Typically, most installations limit the user running **xrootd** to virtually no privileges to avoid any significant security exposure. However, this does mean that all files served by **xrootd** must be accessible by that user. Any files created by **xrootd** must be owned by the user running **xrootd**. Even though files are typically owned by the **xrootd** user you can still provide access control using the built-in (or custom) authorization mechanism to arbitrarily establish virtual access control lists for files and paths.

User mode execution does not significantly impact **xrootd** performance because it uses operating system interfaces that minimize movement of data to and from user-space.

3.3.1 Why is user-mode important

User-mode systems open the software to the masses. Anyone can setup an **xrootd** cluster without the involvement of system administrators. This encourages speedy deployment while providing the same level of security that is afforded to the user deploying the system. A user-mode data access system encourages data sharing at very little cost.

3.4 Multiple Authentication Mechanisms

Most data access systems provide a single authentication mechanism, restricting you to particular security architectures. In **xrootd**, authentication is handled at the protocol layer (i.e. it is fully integrated) using authentication plug-ins. A default set of plug-ins are provided that handle Kerberos, password, shared secret, Unix, and x.509 (a.k.a Grid Security Infrastructure) authentication. One or more of these may be enabled allowing a client to negotiate which authentication mechanisms will be used for access. This allows you to tailor security to whatever requirements are necessary. Since authentication is done via plug-ins you can easily add additional authentication mechanisms, as needed.

3.4.1 Why multiple authentication is important

In today's world many sites distinguish between inside and outside access. Users inside the site (i.e. those who have been vetted) are allowed access to data using low-cost authentication mechanisms, many times tied into the site's single-sign on system (e,g, Kerberos). Users outside of the site are typically required to provide additional credentials before access is granted; a relatively heavy-weight operation (e.g. x509). A multiple authentication mechanism allows a single data access system to distinguish between the two cases and use the appropriate level of authentication. The inside users can enjoy seamless data access while not endangering the system from outside users who, when properly authenticated, can also access the site's data.

3.5 LAN & WAN Federated Clustering

The **xrootd** system is architected for clustering data servers across a local area network as well as the wide area network. A novel and powerful B-Tree mechanism is used to cluster servers. This mechanism allows you to rapidly build extremely large clusters, thousands of nodes, without any sacrifice in I/O performance. Performance is maintained by establishing point-to-point (i.e. cross-bar) I/O connections at the time data is requested.

The B-Tree mechanism employs advance algorithms that not only allow any node to be replicated for added reliability but also to allow clusters to self-organize in real time. Nodes can be added or removed at will without impacting the integrity of the cluster. This means that clusters can be arbitrarily defined in real time as well.

An **xrootd** cluster can consist of not only a set of local nodes but clusters of clustered nodes. The latter affords the capability to federate administratively distinct clusters across the wide area network creating what appears to be a single storage system of loosely coupled storage domains.

The same security used by data servers is employed by the clustering mechanisms to provide full control on who can be a member of the cluster.

3.5.1 Why federated clustering is important

Unlike many of today's clustered file systems, **xrootd** clusters all nodes in a federation model. A federation model allows you to maintain local administrative control of each component in the system. This means changes may occur at the lowest local level without requiring any cluster co-ordination. Most importantly, hardware and software changes can be rapidly done on a schedule conducive to the local domain and failures remain isolated, providing far better reliability than can be achieved in non-federated clusters.

Perhaps one of the most exciting features of federated clustering is that clustering can become a social endeavor where clusters are rapidly created, augmented, and disbanded in response to the particular needs of the moment. This is bottom-up clustering where a group of researchers can get together and create a cluster on-the-fly to share data with minimal effort.

3.6 Automatic Agnostic Data Movement

A special external data movement framework called the File Residency Manager (**FRM**) is fully integrated with **xrootd**. The framework provides an agnostic way of copying data into and out of any **xrootd** server. Copying can be triggered by defined events (e.g. a request for a missing file) or by request (e.g. prepare file for access). The framework does not limit the copy mechanism nor the source or destination of the data. This allows using the framework to implement automatic file backup and restore, multi-tiered storage, migration, and fault staging using a Mass Storage Systems, other data servers, or even federated clusters. Indeed, practically all of these possibilities have already been implemented at various sites.

The **FRM** also includes services that automatically remove stale data, perform live relocation of files to different partitions, and allow assignment of residency attributes to files such as on-line duration and in-memory access.

3.6.1 Why automatic agnostic data movement is important

Most clustered file systems allow automatic data movement but do so in constrained frameworks (e.g. disk to disk, disk to tape, etc). Agnostic data movement removes such constraints allowing a site to construct an arbitrary storage hierarchy that is specifically suited to the problem at hand. Since the data source opportunities are unlimited; one can truly achieve the notion of "any data, anywhere, at any time"; greatly expanding data sharing options.

3.7 Decoupled Logical and Physical Name Spaces

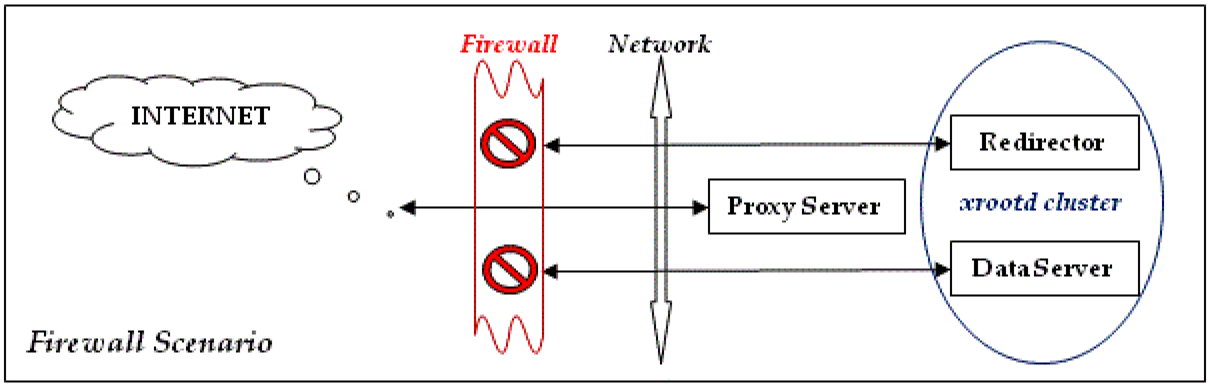
The **xrootd** file name space is designed as a pseudo-hierarchical logical name space. While it can be viewed as a hierarchy of directories, the design actually assumes it is flat. This simplifies uniform mapping of the logical name space to its physical implementation regardless of location and allows each node to implement the name space as efficiently as possible. A trivial logical-to-physical name mapping function is available while more complex mapping can be accomplished by using a name mapping plug-in. The mapping provides the bridge between logical names and their physical counterpart.

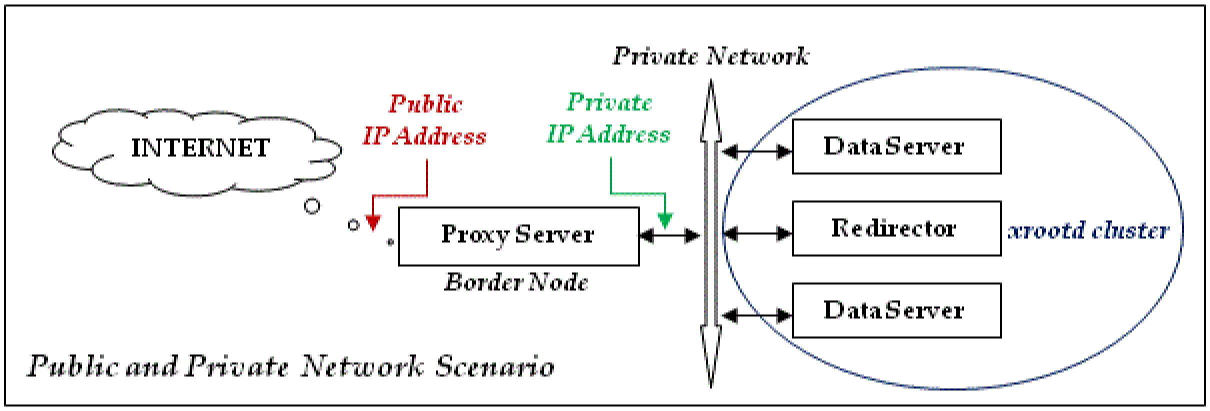
3.7.1 Why decoupled name spaces are important

Separating the logical from the physical names allows **xrootd** to present large clusters of file servers under a single name space regardless of how any file server implements that name space and independent of a file's location in the cluster. This is an extremely important property because it allows registry-free file movement (i.e. no external database). Not only does this significantly improve performance but makes the system far more reliable than registry-based systems since no permanent, possibly stale, state information is needed to locate files and file movement does not rely on a registry being accessible. This makes it possible to practically scale to extremely large server clusters and efficiently aggregate billions of files.

3.8 Integrated Proxy Service

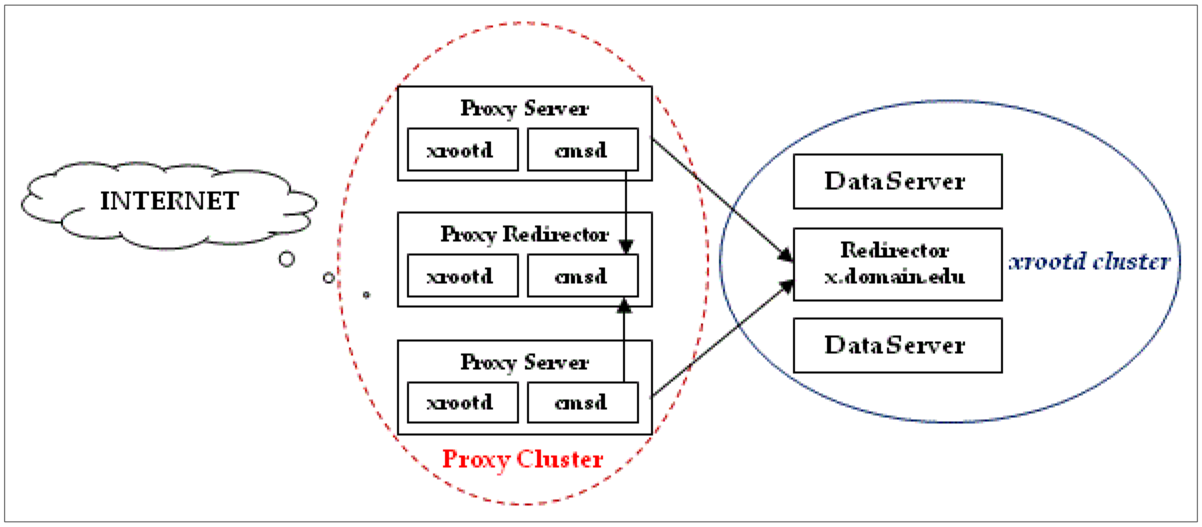
An **xrootd** proxy appears as a standard **xrootd** server but, in fact, fronts an **xrootd** cluster. Data flows between an **xrootd** cluster and a client through the proxy server. A proxy service is required when you wish to grant data access to external users when the data servers reside behind a firewall (i.e. Firewall Scenario, below) or when the data





servers reside on a private network (i.e. Public and Private Network Scenario. above).

Since all data flows through the proxy, this can represent a bottle neck. However, **xrootd** clustering technology also extends to proxy servers and these can be clustered in the same way as data servers, as shown in the following diagram.



A proxy cluster can be scaled to whatever leve is needed to provide the required level of performance. All of these scenarios are possible because an **xrootd** server can also act like an **xrootd** client. Thus, an **xrootd** proxy service is deployed in the same manner as a regular data server except that the data accessed by the proxy server resides elsewhere.

3.8.1 Why a proxy service is important

It has become common to deploy data services behind firewalls or on a private network for security reasons. Without a proxy service it would be impossible to share the data to users outside the site's boundaries, eliminating the possibility of forming WAN clusters. The **xrootd** proxy service provides a secure solution to this problem. Additionally, the proxy server can also act as a virtual firewall. A site can limit what data is accessible through the proxy server as well as who can access that data when the proxy is used. This further enhances security options without impacting internal users.

3.9 Integrated Checksum Support

As part of **xrootd** configuration you can enable file-level checksum support. This support allows you to compute, verify, and report checksums. Built-in support is provided for the three most common algorithms: adler32, CRC32, and MD5. However, any checksum algorithm can be implemented using an easy to write checksum plug-in. Once computed, the checksum is recorded as part of the file and automatically invalidated should the file's contents change.

While many clustered file systems use checksums they typically do so at the block level to provide internal consistency checks. In **xrootd** checksums are used specifically to provide verification of a file's contents. This is in addition to whatever checksum consistency checks are used by the underlying file system (e.g. HDFS, ZFS, etc).

3.9.1 Why checksums are important

When providing access to a large pool of distributed data spread across numerous autonomous sites it becomes crucial to verify that the data that comes from any site is in fact the data you expected to get. Without a common checksum framework sharing data at such a level level would be impractical.

3.10 Detailed Monitoring

Each **xrootd** server can be configured to report various statistics for monitoring purposes. The most common set of metrics that is usually requested is what is called summary statistics. Here the server reports aggregate values on various actions it has taken, the amount of data read, written, and transferred; among many other values. The data is selectable and can be sent to arbitrary monitoring collectors for rendering.

In addition to summary data, an **xrootd** server can also be configured to collect and report detailed information. The amount of detail is also configurable. At the low level **xrootd** tracks general actions taken by each client; at the highest level **xrootd** produces a complete history of every action taken by a client. To minimize the performance impact on the server at the highest monitoring levels, a densely encoded parallel protocol is used to shift the monitoring burden from the server to the data collector; where it arguably should reside. At the full level of monitoring, server performance is decreased by less than 5%; making detailed monitoring practical. The more commonly used summary monitoring has no performance impact.

3.10.1 Why detailed monitoring is important

Detail monitoring provides a unique look into the data access patterns employed by an analysis technique. Without such insight it becomes impossible to tune either the analysis or server data placement to optimize performance. Even finer grained data access pattern analysis is possible using **xrootd** by incrementing the application to insert transition markers into the monitoring stream when detailed monitoring is enabled. This allows discovering exactly which parts of a program require tuning. Optimization of a program's data access is quickly becoming the last frontier in tuning a system to efficiently utilize all available resources.

3.11 Multi-Source Transfers

When transferring a file from one location to another, you can request that all available copies of the file be used in parallel. This is essentially a bit-torrent like transfer. Unlike bit-torrent, partial file segments are neither created nor used. This makes multi-source transfers consistent with single source transfers. The validity of the copy can be transitively verified by simply matching the checksums of all the sources against the final product; an integral part of **xrootd**.

3.11.1 Why multi-source transfers are important

Much research has gone into how to pick the best location from which to fetch a file, even leading to the establishment of network weather services to be used as guides to making such a choice. Using such services is rather complex and, like weather, produces variable results. In the **xrootd** world, one can dispense with the concept of "best" by simply asking that everyone participate to the extent possible. The best will automatically participate most while the worst will add at least something. The approach is uncomplicated and always yields the best result possible even under varying conditions. In fact, multi-source copies easily decrease transfer time by an order of magnitude compared to single source transfers; further increasing productive time.

Of course, this all relies on having multiple copies. Given that the major point of **xrootd** is to easily establish real-time world-wide clusters, the probability of finding multiple copies dramatically increases; making multi-source copying practical.

4 Future Opportunities

While xrootd is a mature system, there are several areas that can benefit from additional research and development. These are, in alphabetic order:

- Authentication protocol transposition
- Automatic rebalancing
- Client-Side algorithms
- Federated site selection
- Monitoring
- Wide area access

The following sections discuss the issues and possible solutions.

4.1 Authentication Protocol Transposition

As described before, **xrootd** provides a complete spectrum of authentication mechanisms. Indeed, **xrootd** is a multi-protocol system. Each site chooses the appropriate authentication algorithms and, in general, the user is unaware of which algorithm is used when accessing a site.

The current choice for wide area access is x.509 based on the Grid Security Infrastructure (GSI) mechanism. While x.509 is a robust authentication protocol, it is also very CPU intensive. As more and more clients connect to a server, a significant amount of the server's time is devoted to authentication rather than data delivery.

A solution to this issue is to allow the server to substitute a lighter-weight protocol after doing the initial x.509 authentication for a client (i.e. protocol transposition). Then, the client would use the transposed protocol when connecting to any data server within the substituted protocol's domain (e.g. data servers within the local cluster). The existing **xrootd** protocol allows for transposing one authentication protocol for another; however, this feature has never been exploited because several research questions remain:

- The choice of a light-weight protocol,
- should multi-protocol support extend to protocol transposition,
- how protocol expiration times should be handled,
- what information needs to be maintained during the transposition, and
- the specification of security risks that may be introduced.

These are significant research questions and the proper implementation of the choices is no less significant in terms of development effort. However, the pay-off is rather large in the expectation that external access to data will become a large percentage of all data access requests.

4.2 Automatic Rebalancing

Currently, when a site adds additional disk to a server, the site administrator must manually rebalance files to maintain an equal amount of space utilization among all the disk resources. Inductively, this also means that the same manual intervention is required when adding a new disk server. To date, there was no overwhelming reason to automate the process since rebalancing usually required that files be moved in context. That is, random file movement for purposes of space equalization would usually produce hot-spots as groups of concurrently used files would wind up on the same disk subsystem or same server. Hence, only an administrator would know the appropriate set of file candidates that should be moved to avoid hot-spots.

In order to address this problem, several research questions need to be addressed:

- How to efficiently tag mutually exclusive files,
- should rebalancing be an ongoing activity or a triggered event,
- to what level should data access patterns enter into rebalancing decisions,
- algorithms that need to be employed to prevent data loss, and
- the role of the administrator in the rebalancing effort.

Automatic rebalancing is potentially a clear win in terms of space administration, if it can be properly done. Otherwise, the current manual scheme is a safer proposition.

4.3 Client-Side Algorithms

While the **xrootd** client performs well, more and more users are pointing out that it could perform far better, especially as the number of cores per worker node increases; necessitating running ever more jobs on a single node. The particular areas that need to be addressed are:

- Reduced CPU utilization (conservatively by 50%),
- a plug-in architecture for client-side pre-reading and caching of data blocks,
- using multiple data sources for random access, and
- a fully asynchronous threading model for enhanced parallelism.

No doubt, addressing these issues will likely require re-architecting the client code. However, the end result will be a client that can be automatically customized on-the-fly to perform as efficiently as possible for any particular job; allowing for the full utilization of large multi-core worker machines.

4.4 Federated Site Selection

For all practical purposes, **xrootd** employs the same server selection algorithm when selecting a server in a local cluster as it does when selecting a server in a globally federated cluster. This is not to say that the algorithms are identical. Indeed, xrootd is aware of the local and global differences. However, such awareness has not been fully exploited because server research questions remain:

- How to automatically determine round trip time between a client and the ultimate data source,

- metrics that apply to global selection that differ from local selection,
- the degree that server avoidance should play in site selection, and
- an appropriate plug-in model for server selection that has negligible latency.

While the current system does incredibly well in managing a local cluster, that same level of performance should be expected on the global level. Currently, that is not the case. Addressing these issues will go far in understanding how federated clusters should be managed to achieve high performance and make federated access a main-stream technology.

4.5 Monitoring

The **xrootd** system can uniquely provide massive amounts of information about its processing choices as well as performance metrics. Indeed, efforts to mine the available information as well as choices on how to render that information is a meaningful way have just started. Currently, only basic monitoring information is available to administrators via widely used monitoring agents (e.g. Ganglia, Mona Lisa, etc). This is in large part because no one really knows what information is truly meaningful, how the various pieces inter-play to produce a full image of a functioning system, and what information should reliably produce alerts to indicate possible problems.

The effort required to fully exploit available monitoring information is substantial simply because such exploitation is a nascent technology. Addressing, this issue will not only help in administering **xrootd** sites, federated or not, but also add to the understanding what monitoring information is needed in other similarly large-scale data access systems.

4.6 Wide Area Access

While **xrootd** was engineered for wide area access, and does so better than one would expect, there is still substantial room for improvement in terms of random access. The need for additional improvement is largely driven by the fact that wide area access is quickly becoming the more and more important as sites federate and site administrators do not want to locally replicate files that are generally accessed only a few times across the wide area network.

A promising solution is to interpose a local proxy cluster between clients at a site and the wide area network. The proxy cluster is then used to optimize random access across the WAN. This technology is already provided by xrootd but has not been exploited as several important questions remain to be addressed:

- How security should be handled through a proxy cluster,
- what algorithms should a proxy cluster employ to minimize latency,
- to what degree should data be semi-permanently be cached, and
- what are the scaling characteristics of such a setup.

A successful resolution of random WAN access using a proxy scheme would not only benefit current users but would immensely add to the understanding of how to efficiently provide random access across the wide area network to the field of computer science; ultimately making federated access the preferred solution.

5 Conclusion

Xrootd is as relevant today as it was a decade ago. It's dynamic with active development with collaborators at SLAC, CERN, BNL, UCSD, and the University of Nebraska. Timely support is provided by SLAC, CERN, Duke University, and the Open Science Grid (OSG). It is being adopted at an ever increasing rate by laboratories and universities across the world. It is no wonder why **xrootd** has become one of the cornerstones in data analysis for Astronomical and high energy physics in local clusters and world-wide grids.