XROOTD - A highly scalable architecture for data access

ALVISE DORIGO[1], PETER ELMER[2], FABRIZIO FURANO[3], ANDREW HANUSHEVSKY[4]
[1]Physics Department Galileo Galilei (Padova, Italy) and INFN Padova
Via Marzolo, 8 35131 Padova ITALY
(alvise.dorigo@pd.infn.it)
[2]Princeton University, Princeton, NJ 08544 USA
(elmer@slac.stanford.edu)
[3]Università Ca' Foscari Venezia and INFN Padova, I-35131 Padova, ITALY
(fabrizio.furano@pd.infn.it)
[4]SLAC, Stanford University 94025, USA
(abh@slac.stanford.edu)

Abstract: - When dealing with the concurrent access from a multitude of clients to petabyte-scale data repositories, high performance, fault tolerance, robustness, and scalability are four very important issues. This work describes the architecture and the choices done in designing the xrootd file access system. The first goal of the system was to provide access to over 10^7 files representing several petabytes of experimental physics data. This work addresses the high demand data access needs of modern physics experiments, such as the BaBar experiment at SLAC, and covers method and tools useful to any other field in which reliability, performance and scalability in data access are a primary issue.

Key-Words: - Scalability, Fault tolerance, load balancing, peer to peer, XROOTD, TXNetFile, ROOT

1 - From a storage system to a data access architecture

The BaBar experiment [1] at the Stanford Linear Accelerator Center produces a huge amount of data to be accessed by a high number of analysis jobs. For this reason it requires a reliable and scalable data access system. In 2002, the BaBar Computing Model 2 committee decided to migrate the data storage system from Objectivity/DB to a flat file system built upon object streams (aka "Kanga", [2]). The new storage system is based on the persistency mechanism of the C++ ROOT framework, developed at CERN [3], that is able to stream an object on a binary file in a similar way as the Java framework does.

ROOT also provides a remote file access mechanism via a TCP/IP-based data server daemon known as `rootd` which has the only purpose to serve opaque data. A plugin manager (Fig.1) hides the user from the actual location of the files, by masking the path (local file system or remote server) which is going to use to access the data.

The purpose of getting access to remote file repositories could be reached, in principle, with other remote file access mechanisms. For instance, one could distribute the file repositories over many NFS servers [4], but a deeper requirements analysis shows that this kind of solution is not acceptable for many reasons:

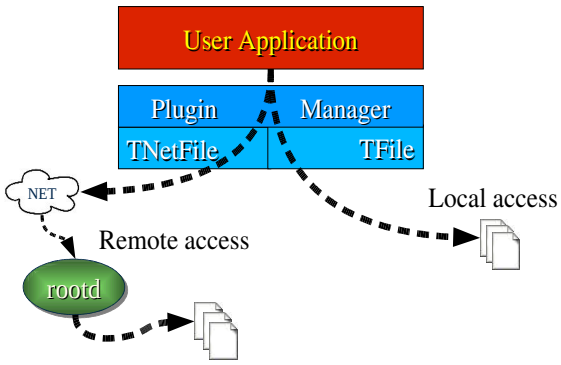

Fig.1 Transparent local and remote files access

- the size of the data repository: thousands of thousands of files must be scattered among multiple servers. Beside the complex organization needed, NFS does not provide any tool useful to automatically manage the choice of the right mounted partition disk to access for a request, i.e. what is often called "logical to physical name resolution";
- the number of concurrent accesses to the data: thousands of concurrent accesses from end users and batch jobs, that continuously analyze the data

1 This work was funded by the Department of Energy under contract DE-AC02-76-SFO0515 with Stanford University.

in a completely random way, greatly overcome the scalability of the basic NFS architecture [5];
- software engineering issues: NFS simulates a local file system and most jobs are not much tolerant to all possible troubles accessing local files;
- the use of NFS would implies mounting remote volumes on the machine where the user is running his jobs; this is not acceptable from the point of view of many system administrators, and especially in the case in which the user machine is a desktop computer or even a laptop;
- if for any reason some NFS servers are having troubles, typically the machine mounting the remote volumes will experience problems when any NFS access is tried, with no user (or application) control of timeouts, retries etc.

To overcome some of these limitations, the alternative of building a data server suggests a different paradigm which can be deployed or extended in order to satisfy the heavy requirements of the data analysis tasks. At the server side, `rootd` offers the solution to share this big load between many machines keeping the files on their local disks, while at the client side, a specialization of the ROOT's data access classes can provide a way to access the remote data which is transparent to the users of the framework.

2 - Performance, scalability and fault tolerance: the main goal

The deployment of big processing farms, as well as of data access systems able to handle millions of scattered files must be able to give data processing services to a wide community of users with high availability and performances. Some of the needed characteristics are:

- multiple servers have to cooperate with the purpose of handling huge amounts of distributed (and redundant if necessary) data without forcing the client to know which server to contact to access a particular file;
- the server has to hide the client applications from its underlying file system types, even if it manages one or more tape units;
- a load balancing mechanism is needed, in order to efficiently distribute the load between clusters of servers;
- the system resources (sockets, memory, cache, disk accesses, cpu cycles, etc.) have to be used at the best, at both client and server sides;
- a high degree of fault tolerance at the client side is mandatory, to minimize the number of jobs/applications which have to be restarted after a transient or partial server side problem or any kind of network glitch.

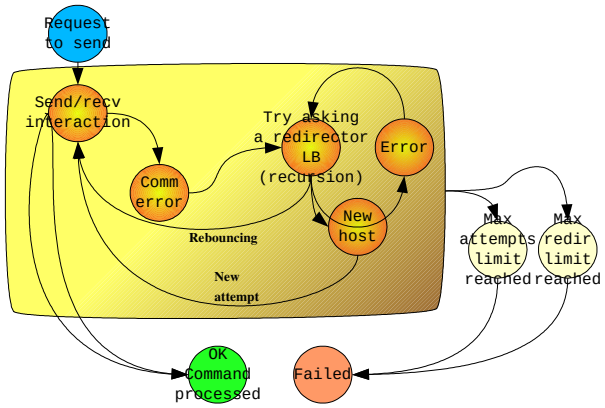

Fig.2 State transition diagram for the fault tolerant behavior of a client

The new architecture designed by the BaBar software specialists to achieve all the requirements listed above has been called **`xrootd`** ("eXtended" `rootd`). Its structure allows the construction of single server data access sites up to load balanced environments and structured peer-to-peer deployments, in which many servers cooperate to give an exported uniform namespace.
These kind of structures in any case present an interface defined as a communication protocol, which defines the possible interactions and the functionalities given to the clients.
The specific client of the `xrootd` system has been built both in ROOT compliant form (officially integrated in ROOT and multplatform) and also as a POSIX compliant one. Some of the design choices which give the needed functionalities are:

- the communication protocol, which defines an interface able to:
  - request access to an `xrootd` system through authentication handshakes
  - query a system for resource location
  - get access to the requested resource in the place where it can be accessed (i.e. servers giving access to local data or `xrootd` proxies allowing remote sites interoperability [6])
- sophisticated communication policies at the client side, able to handle any kind of communication errors (Fig.2). The failing requests are retried until:
  - another working server is found;
  - the same server becomes available again;
  - a specified maximum number of retries is reached;
- multiplexed persistent connections: this means that a single TCP connection from a client to a server can carry multiple independent data streams for other client instances. Also, TCP connections are persistent for a short period if connected to a data server, for a long time if connected to a redirector. This helps in lowering the system

resource consumption and the network overheads due to repeated multiple connections to the same host.

2.1 - Related work

Most of the work found in literature concerning fault tolerant and fast data access doesn't deal with communication robustness and highly available systems. Furthermore the distributed file system paradigm is usually tied to policies dealing with distributed caching and coherency, path and filename semantics. These are some reasons why a distributed file system usually causes a consistent network and cpu overhead when dealing with the operations on the file it manages [7][8].

One of the problems which may arise is given by the network overhead due to the synchronization of the internal caches, a serious issue when dealing with petabytes of data continuously accessed by thousand of clients, a scenario not so usual in the distributed file systems world.

However, such a perspective is common in many the organizations that rely on massive data sharing, not only depending from the ROOT package or the physics community. For instance, such a robust file server facility could be integrated in the many Grid [9][10][11] initiatives that will support the analysis on next generation physics experiments or other fields.

Scalability is another critical aspect considered in literature. Many existing systems are scalable in some way, [4]; but what can be noted is that very often the scaling measurements are done with numbers of tens clients per server, not hundreds of thousands that could be sources of critical server lockups. This is another reason for thinking about an architecture for data access, trying to reach a “nearly linear” scaling performance, limited only by the data throughput and latency of both disks and networks.

For this reason, the main focus in this work has not been the pure data throughput for a single client [12] [13]. In fact it can be noted that the pure data throughput of NFS [5], for a single server, may be higher than that of a user-mode application like a daemon implementing a TCP file server. But to handle thousands of clients one must have the possibility of putting together many servers in parallel, achieving scalability and total transparency from the clients' perspective and also minimizing the system resources needed by such communication mechanisms.

This work has many common points with the one described about the Google File System [14]. A difference with the Google File System is that it considers parts of files (chunks) as its data unit, while xrootd has the single file. Also, at this moment, the xrootd system does not have a mechanism to keep the coherence between multiple copies of a file which might be modified by an application, since it's not needed by its current deployment. It seems also that the xrootd/TXNetFile project put a bigger effort in refining the communication policies and the resource consumption.

Other interesting works can be found in the distributed file systems area. An interesting approach, for some verses similar to that of peer-to-peer file sharing networks [15][16] is depicted in [17] with the xFS implementation. Approaches for some verses more similar to that used by xrootd to locate and distribute files can be found in [18][19].

2.2 - XROOTD – The server side

2.2.1 - Overview

`xrootd` represents the culmination of work previously done in understanding the remote data server scalability issues with the Objectivity/DB’s Advanced Multithreaded Server (AMS) [20]. While the xrootd server is not innovative in the sense that it appears externally to be like any other remote data server, it does represent a significant advance in such data serving architectures. The server is composed of four layers: Network and thread management layer, Protocol layer, File system layer, Storage layer, see Fig.3.

A layered approach allows us to optimize a specific set of functionalities to minimize resource usage. Additionally, each layer is sufficiently isolated so that

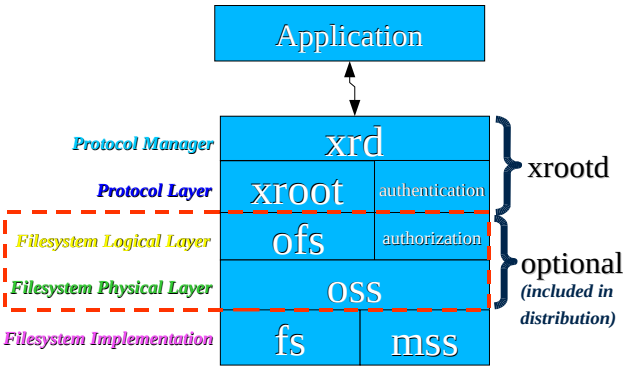

Fig.3 xrootd architecture

it can be dynamically loaded. This allows for a number of implementations to determine which functions best in any particular environment. By extension, multiple implementations of a particular layer can be loaded at one time. This proved to be essential in providing backward compatibility with rootd since the thread management layer could run the rootd as well as the xrootd protocol at the same time; the selection determined by the level of client software being run.

2.2.2 - Network and Thread Management

This layer is essentially the protocol dispatcher. It isolates all other layers from the details of socket and thread management. The particular optimization that were incorporated in this layer were:

- Socket stickiness: a socket is temporarily bound to a thread and associated objects to avoid rescheduling overhead between requests. The socket is unbound when it becomes inactive or when system resources become too constrained to allow such dedication of resources.
- Socket polling: the mechanism used to detect sockets ready for I/O is made architecture dependent. For instance, /dev/poll, a more efficient polling mechanism, is used on those platforms that support it. Otherwise, poll() is used but is optimized to significantly reduce the processing cost of the poll table.
- Object persistence: allocated objects remain allocated as long as there is a reasonable potential for reuse. This includes thread objects. Managing objects in this way optimizes memory usage while minimizing synchronization points where objects are allocated and deleted.
- Generalized object scheduling: the major object at this layer is "job" class. Most object derive from the job class and objects at other layers generally do so as well. Any job class derived object can be asynchronously scheduled to perform internal maintenance and tuning functions and is extensively used for global optimization purposes.

2.2.3 - Protocol Layer

The xrootd protocol is the default protocol run by network and thread management layer. This 64-bit TCP-based protocol provides generalized file access, that in many ways is similar to AFS. However, the protocol has been optimized in several ways to be more scalable:

- Multiple independent streams are supported on a single socket. This minimizes resource usage in the presence of multiple requests.
- Clients can be redirected to another server at any time. This allows dynamic server selection and load distribution while providing for direct point-to-point client-server connections.
- Clients may be asked to delay server contact. This allows the server to coordinate overloads and resource constraints by pushing the problem back into the network; avoiding typical server meltdowns when faced with a client onslaught.
- Clients may piggy-back read-ahead lists with any read request. This allows the server to optimize future disk access and provide better performance.
- Client may ask for files to be prepared for future access. This allows the server to make sure files are online and properly placed for future client requests.
- Servers may ask clients to perform certain actions at any time. This is known as unsolicited response requests that typically take the form of redirects and execution deferrals; providing the server maximum flexibility in coordinating the load.

Additionally, the protocol provides for a generalized security framework that allows any authentication protocol to be used; providing scalable security.

The scalable protocol elements also provide the foundation for the inclusion of peer-to-peer capabilities. Here, contacts would contact a distinguished server that would search for the best possible source of the requested data and then direct the client to the corresponding server. Should that server become unavailable, the client is always free to launch another search.

Because traditional client-server interactions and peer-to-peer model are incorporated in the same file access protocol, it is possible to cover the full range of file access architectures; maximizing the potential for scalability. Indeed, xrootd has been successfully deployed in a peer-to-peer environment; largely because of the protocol's ability to accommodate different access models.

2.2.4 - File System Layer

The file system layer provides an implementation independent view of a file system. It is at this layer where peer-to-peer file-access decisions, if enabled, are done. This allows uniform access in a variety of situations. Additionally,

- Multiple requests for the same file are merged.
- Redundant file operations across multiple clients are screened out.
- Idle files are closed.

This layer also provides file-based access control based on the authentication information provided by the protocol layer.

2.2.5 - Storage Layer

The storage layer provides the particular implementation of a logical file system. Here, logical file system operations are translated to specific actions optimized to underlying storage. Some of the enhancements provided by this layer are:

- Transparent access to multiple file systems. Here, any number of file systems can be aggregated to provide for a single view of storage. File system aggregation allows one to increase the number of actuators and provide greater flexibility in the physical placement of files for increased performance.
- Mass Storage System integration. Here, near-line

or off-line storage can be added to enhance the capacity of on-line space. Files are transparently staged and de-staged from on-line disk.

2.3 - The client side

The base of the fault tolerant and reliable behavior of the architecture is defined in the communication protocol and implemented in the client. The protocol defines the behavior of the client in the case of explicit redirection requested by the server (for example it can redirect the clients somewhere else because it is going off-line for maintenance) or communication errors (a particular data server crashed or unexpectedly closed a connection). In both cases the client has to apply some rules in order to launch a new search for the file and get redirected to a new available host.

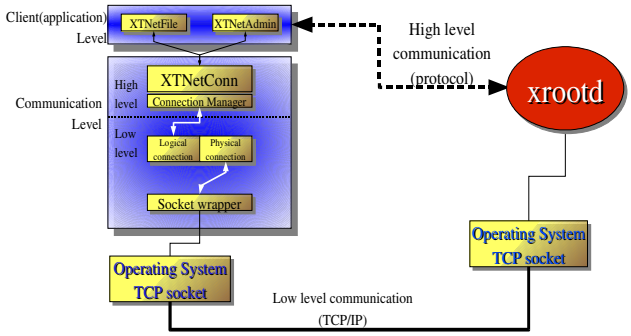

Fig.4 Architecture overview of the client side

The client is composed by three layers with different tasks:

- the interface layer: here the methods dealing with file access are defined;
- the high level communication layer: here the protocol directives are implemented, as well as the policies related to fault tolerance, read caching and read ahead;
- the low level communication layer: here the protocol packet structure is known, in order to give the functionalities of:
  - connection multiplexing;
  - raw data receiving and un-marshaling into an internal message stream, in a parameterizable asynchronous (using a queue and a reader thread per physical connection) or synchronous way;
  - raw data marshaling and writing to the connections through a socket wrapper layer. It is such a kind of physical layer that performs all socket-related operations like read/write, socket polling to handle read/write timeouts, connection and disconnection, connection timeout detection;
  - Socket errors handling.

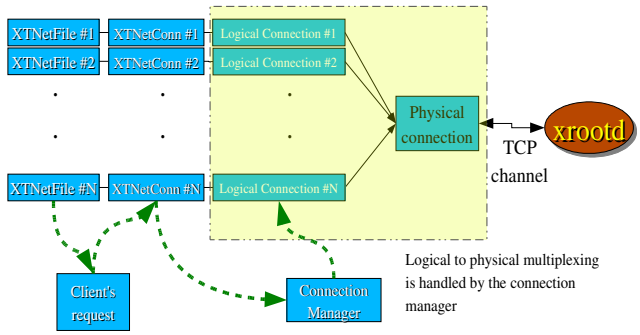

Fig.5 Connection multiplexing and path of a message

3 - Benchmarks

In order to test the real-world performance of the server, a series of BaBar analysis jobs were run against a single file, allowing data to be served from the file system memory cache and avoided disk-speed anomalies that make performance results hard to interpret. The CPU-intensive work in the analysis job was removed to force the maximum possible request rate from each client while preserving the original data access pattern. Thus an "event" in this context represents a bounded series of server transactions.

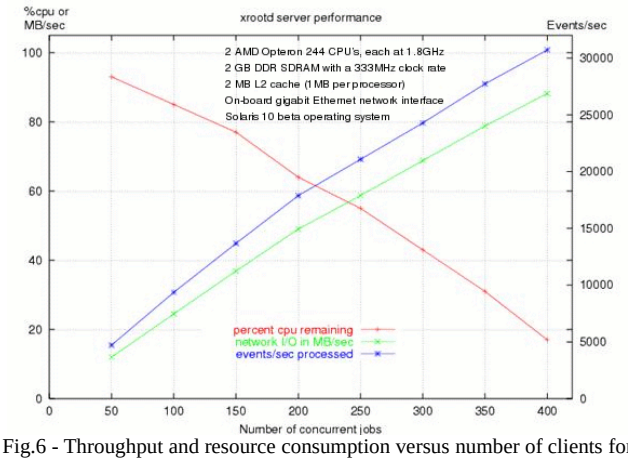

Fig.6 - Throughput and resource consumption versus number of clients for a single server

The red line in Fig.6 shows that the number of events per second scales linearly with the number of clients. The graph also shows that the rate of increase unexpectedly slows after about 200 clients. This effect is a benchmark induced aberration. The first two hundred clients were each run on a dedicated machine; after which up to two clients were run on each machine. Running more than one client on a machine adversely affected each client machine's performance by 9.7%. This loss of efficiency appears as a deviation of the expected event rate.

4 - Future directions

While the system described in this work is already being deployed for production use at many

computing centers used by the BaBar experiment, many enhancements are possible as future work.

We must remember that `xrootd/TXNetFile` was born as a sophisticated and scalable data access architecture, and its possible enhancements up to the level of a full scalable and fault tolerant file system must be compatible with the requisites of the huge data repositories which are its main aim.

Future work will involve studying how non-contiguous name spaces impact peer-to-peer message passing performances and what kind of restrictions, if any, need to be put into place to keep such a system from showing bottlenecks due to an excessive message exchange between peers.

Other tasks that will be started include gathering a more precise knowledge about the performances of the various possible architecture levels, from a single server site to a big site with many load balanced servers, up to a network of cooperating servers.

Since gathering real world performance measures is a task which can be done in simpler deployments, an interesting possibility is to calculate or simulate the behavior of a complex system. The main objective of such a task would be trying to acquire knowledge about the behavior of complex multisite deployments, interconnected by WANs and proxies.

References:

[1] - The BaBar Experiment, 2004, http://www.slac.stanford.edu/BFROOT

[2] - T.J. Adye et al., *Kanga(ROO): Handling the micro-DST of the BaBar experiment with ROOT*, 2003, pg. 174-214, Computer Physycs Communications, vol. 150

[3] - Official ROOT site, 2004, http://root.cern.ch

[4] - F.Garcia, A.Calderon, J.Carretero, J.M.Perez, J.Fernandez, *A Parallel and Fault Tolerant File System based on NFS Servers*, PDP 2003 - 11-th Euromicro Conference on Parallel Distributed and Network based Processing Genoa - Italy, IEEE Computer Society, 2003

[5] - James Hall, Roberto Sabatino, Simon Crosby, Ian Leslie, Richard Black, *Counting the Cycles: a Comparative Study of NFS Performance over High Speed Networks*, 22nd IEEE Conference on Local Computer Networks (LCN '97), IEEE Computer Society, 1997

[6] - Andrew Hanushevsky, Heinz Stockinger, *A Proxy Service for the xrootd data server*, Proc. of the First International Workshop on Scientific Applications on Grid Computing (SAG'04), Sept, 2004, http://xrootd.slac.stanford.edu/papers/xrootd-sag2004.pdf

[7] - John H. Howard et al., *Scale and performance in a distributed file system*, ACM Transactions on Computer Systems, Feb, 1988, 6

[8] - Cary Whitney, *Comparing Different File Systems' NFS Performance. A cluster File System and a couple of NAS Servers thrown in*, The sixth SCICOMP Meeting, SCICOMP 6 (Univ. of Berkeley), IBM System Scientific Computing User Group, 2002, http://pdsf.nersc.gov/talks/talks.html

[9] - Official European Grid web page, 2004, http://eu-datagrid.web.cern.ch/eu-datagrid/

[10] - Official Grid-Egee web page, 2004, http://egee-intranet.web.cern.ch/egee-intranet/gateway.html

[11] - Ian Foster, *Grid Technologies & Applications: Architecture & Achievements*, CHEP'01 – Computing in High Energy and Nuclear Physics, IHEP, Bejing (China), 2001

[12] - Andrew Hanushevsky, Artem Trunov, Les Cottrell, *Peer-to-Peer Computing for Secure High Performance Data Copying*, CHEP'01 – Computing in High Energy and Nuclear Physics, IHEP, Bejing (China), 2001

[13] - T. F. La Porta and M. Schwartz, *The MultiStream Protocol: A Highly Flexible High Speed Transport Protocol*, IEEE Journal on Selected Areas in Communications, 1992, IEEE Computer Society

[14] - Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, *The Google file system*, Proceedings of the nineteenth ACM symposium on Operating systems principles, ACM press, 2003

[15] - Steve R. Waterhouse, David M. Doolin, Gene Kan, Yaroslav Faybishenko, *Distributed Search in P2P Networks*, IEEE Internet Computing Journal, 2002, 6

[16] - Matei Ripeanu and Ian Foster and Adriana Iamnitchi, *Mapping the Gnutella Network: Properties of Large-Scale Peer-to-Peer Systems and Implications for System Design*, IEEE Internet Computing Journal, Jan-Feb, 2002, 1, Springer-Verlag

[17] - Thomas E.Anderson, Michael Dahlin, Jeanna M.Neefe, David A. Patterson, Drew S. Rosselli, Randolph Y.Wang, *Serverless Network File Systems*, ACM Transactions on Computer Systems, Feb, 1996, 1, ACM press

[18] - Dennis Heimbigner, *Adapting Publish/Subscribe Middleware to Achieve Gnutella-like Functionality*, Technical Report CU-CS-909-00 Department of Computer Science University of Colorado, University of Colorado at Boulder, Sept, 2000

[19] - Ian clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong, Freenet: *A Distributed Anonymous Information Storage and Retrieval System*, Proc. of the ICSI Workshop on Design Issues in Anonymity and Unobservability, International Computer Science Institute, 2000

[20] - Andrew Hanushevsky, The Advanced Multithreaded Server, http://www.slac.stanford.edu/~abh/objy.html