THE NEXT GENERATION ROOT FILE SERVER

Andrew Hanushevsky, SLAC, Stanford, USA
Alvise Dorigo, INFN, Padova, Italy
Fabrizio Furano, INFN, Padova, Italy

Abstract

As the BaBar experiment shifted its computing model to a ROOT-based framework, we undertook the development of a high-performance file server as the basis for a fault-tolerant storage environment whose ultimate goal was to minimize job failures due to server failures. Capitalizing on our five years of experience with extending Objectivity's Advanced Multithreaded Server (AMS), elements were added to remove as many obstacles to server performance and fault-tolerance as possible. The final outcome was xrootd, upwardly and downwardly compatible with the current file server, rootd. This paper describes the essential protocol elements that make high performance and fault-tolerance possible; including asynchronous parallel requests, stream multiplexing, data pre-fetch, automatic data segmenting, and the framework for a structured peer-to-peer storage model that allows massive server scaling and client recovery from multiple failures. The internal architecture of the server is also described to explain how high performance was maintained and full compatibility was achieved[1]. Now in production at Stanford Linear Accelerator Center, Rutherford Appleton Laboratory (RAL), INFN, and IN2P3; xrootd has shown that our design provides what we set out to achieve. The xrootd server is now part of the standard ROOT distribution so that other experiments can benefit from this data serving model within a standard HEP event analysis framework.

THE XROOT SERVER

The xroot server architecture is shown in figure 1. It is composed of multiple components. Each component serves a discreet task and is easily replaceable. The collection of components is called xrootd.

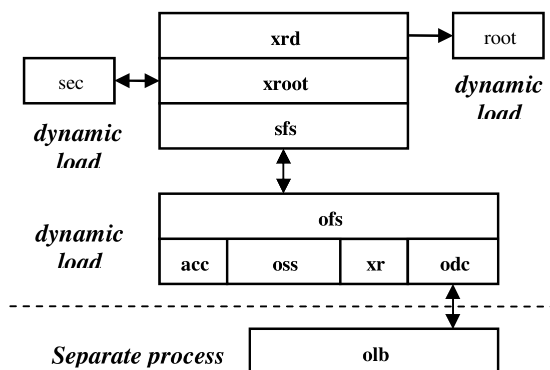

Figure 1: The xrootd Architecture

The xrd Component

The xrd component provides networking support, thread management, and protocol scheduling. This component has the potential to severely impact sever scalability. Careful attention was given to algorithms used to insure minimum overhead per client-server interaction. The performance oriented features include:

Use of the best socket handling features that the underlying OS provides.

Threads are managed by a lightweight scheduler, The scheduler attempts to keep enough threads ready to handle the recently experienced load. Generally, threads are created for each incoming request, up to the maximum computed based on system resources. The exact number can also be configured. Once the maximum is reached, threads are shared by all clients. When threads become idle, they are automatically eliminated using an exponential decay function.

The xrd component allows multiple protocols to be used at the same time. Each configured protocol is asked whether it can handle an incoming connection. A protocol object instance is created once a match is found. This object is then scheduled, when necessary, to handle client interactions. This feature is used to provide simultaneous xroot and root protocol support

The xrd component provides for the highest level of parallelism by avoiding functions that tend to serialize execution, maintaining suitably grained locks, and using threads whenever possible to perform internal housekeeping. As such, it implements a very low overhead protocol engine capable of serving thousands of clients.

The xroot Component

The xroot component implements the xroot protocol. This protocol provides generalized POSIX-like file access enhanced by High Performance Computing (HPC) extensions, and fault recoverability (FR) features. The protocol is architected as a platform-neutral simple binary stream to eliminate most of the encoding-decoding overhead associated with other similar protocols. While this reduces the chances of general inter-operability, the HPC and FR extensions make the protocol unlikely to inter-operate with other network based file protocols without sacrificing significant usability. The significant HPC extensions include:

- asynchronous responses so that a client can launch multiple requests at the same time,

- asynchronous I/O when the operating system supports it and resources are available,
- pre-reading data so that it is available in memory on the next client interaction,
- asynchronous file access preparation to minimize file open overhead,
- automatic I/O segmenting to allow data stream multiplexing, and
- client-directed request monitoring to allow application tuning.

As can be seen, most of the HPC features involve enabling a rich set of asynchronous facilities to provide clients the maximum number of opportunities for parallelism. The significant FR features include:

- request redirection so clients can be dynamically steered to least loaded operating servers,
- request deferral so that the server can even out highly variable loads without increasing its own state overhead,
- client-directed error state notification, and
- unsolicited responses to asynchronously reconfigure client-server connections.

The FR features are used to implement dynamic load balancing, server failure recovery. Most of these features are used in combination with the Open Load Balancing (olb) structured peer-to-peer (SP2) system that can be run in conjunction with xrootd. The olb system is described later in this paper.

In addition to providing xroot protocol file access, the xroot component is also responsible for invoking the authentication component. The authentication component, not described in this paper, is architected as a general authentication protocol plug-in mechanism capable of simultaneously supporting multiple protocols.

The sfs Component

Since file serving is the focus for xrootd, it interacts with another component to provide file access. This service is based on the Standard File System class, XrdSfs. The actual implementation of this class is loaded at run-time; allowing for numerous implementations, as needed by any particular installation. A default implementation is provided that provides the minimum set of features to support the xroot protocol. Another configurable implementation is also provided. This implementation supports all the xroot protocol features and is called the Open File System (ofs) component.

The ofs Component

The ofs component provides enhanced first level access to file data. Since this component is expected to support the full set of xroot protocol features, it is architected as a multi-component service. Each component is responsible for implementing a particular set of features that can be easily re-implemented to correspond to the actual underlying architecture. These components are:

- Access Control (acc based on the XrdAcc class),
- Open Distributed Cache (odc based on the XrdOdc class),
- Open Storage Service (oss based on the XrdOss class), and
- Peer proxy service (xr based on the XrdXr and XrdOss classes).
- The ofs component is responsible for coordinating the activities of these components to provide an effective file system view.

The acc Component

The acc component implements the authorization service. This service is responsible for granting clients access to files. It uses the authentication information, if any, passed through by the xroot component. The authorization component is implemented as a reverse file capability list. A capability oriented implementation was chosen to optimize operations when the number of files substantially exceeds the number of users capable of accessing files. In this scheme, each user and user association can be granted or denied access to files that start with a particular prefix. The set of privileges correspond to those implemented by Windows XP and is a super-set of POSIX privileges. Generally, this provides the ability to associate capabilities (or lack of capabilities) to users. It is a reverse file capability list in that specifying a file prefix completely effectively implements an access control privilege scheme where a file (or set of files) is associated with a number of users and their capabilities. Thus, allowing for ACLs in those cases where fine-grained access control is necessary.

The odc Component

The ods is responsible for locating the right server to use for a particular file open request. It is invoked by the ofs when dynamic load balancing or proxy support is configured.

The odc provides numerous services under the guise of finding the right server for the requested file. The four main functions are:

- communicating with the olb to discover the location of a file and appropriate server to provide access to that file,
- passing xroot protocol requests to the olb that may need to be handled on a remote host (e.g., file preparation, file removal, etc),
- coordinating the activities of other xrootd servers running on the same host, and
- initiating the use of a proxy service should remote file access be needed.

When invoked, the odc may respond with a server-port pair indicating that the client should be redirected to that host for subsequent file access. This occurs when the server is configured in "redirect remote" mode. The odc may respond with a simple port number, indicating that the client should be redirected to another xrootd server running on the same host. This occurs when the server is configured in "redirect local" mode. The odc may respond

with an instance of a oss object that should be used for actual file access. This occurs when the server is configured in "redirect proxy" mode. When the server is configured in "redirect target" mode, the odc passes execution state information to the local olb. That information is used in redirection decisions by other olbd's serving xrootd configured in "redirect remote" mode. Finally, the odc may respond with a null response indicating that the incoming request should be processed by the oss component as if the odc was not configured.

For redundancy, the odc can communicate with multiple olbd's in order to provide a fault tolerant environment as well as to load balance requests among all of the olbd's. The mechanisms used to distribute requests so that a consistent file system image is maintained is outside the scope of this paper.

The oss Component

The oss component is responsible for providing access to the underlying file system. It is invoked by the ofs component to perform actual I/O as well as execute file system mete-data operations (e.g., rename, remove, etc). As such, it implements a physical storage system.

We differentiate the phrase file system and storage system in that a storage system provides access to stored data that may or may not be reside in an actual file system. For instance, the file may reside in a Mass Storage System (MSS) and will need to be retrieved prior to access. Alternatively, the file may reside on another xroot server and a proxy relationship will need to be established in order to provide access. In all cases, the actual act of providing access to data is handled by the oss. The mechanism used to provide that access is encapsulated by the oss to provide a uniform view of storage regardless of how it must be accessed.

The ofs dynamically loaded library contains a generic implementation of an oss that provides the following essential services:

- access to an MSS using configurable agnostic call-outs so that any kind of MSS can be used,
- a generic file system cache facility so that multiple file systems can be aggregated into a single uniform view,
- a proxy service to provide real-time access to files that reside on other xroot servers,
- I/O and meta-data access to a UFS-type file system, and
- asynchronous I/O capabilities, should the operating system support it.

ADDITIONAL PERFORMANCE AND FAULT TOLERANCE

While the xrootd server was written to provide single point data access performance with an eye to robustness; it is not sufficient for large scale installations. Single point data access inevitably suffers from overloads and failures due to conditions outside the control of the server. Our approach to solving this problem involved aggregating multiple xroot servers to provide a single storage image with the ability to dynamically reconfigure client connections to route data requests around server failures. Such an approach can work as long as servers are not interdependent. That is, while servers can be aggregated, a failure of any server should not affect the functioning of other servers that participate in the scheme.

The approach we took was modelled after many existing peer-to-peer systems which have shown to be extremely tolerant of failures and scale well to thousands of participating nodes. The structure consists of one or more servers, called redirectors, rooting a B64 tree structure of data servers. Information flows up the tree to the redirectors that redirect client requests to servers lower in the hierarchy.

From the server's perspective, data servers only know that they are participating in a cooperative structure but no single data server is aware of the structure. Servers at the root of each B64 node only know the existence of their immediate neighbours and one or more servers higher in the tree. This effectively isolates failures to small areas within the configuration. Even the most significant failure in the structure only causes a small part of the overall structure to reconfigure in order to maintain a cooperative data access view.

We chose a hierarchical model because this minimizes the number of messages that needed to flow through the system and creates predictable access paths. The choice of a B64 tree was done out of practical necessity to keep the decision making overhead to reasonably low levels; avoiding latency pile-ups that could cause the system to become unstable. We call this a structured peer-to-peer model because while servers work in a peer-to-peer fashion within the system, a particular structure is imposed.

The system provides high levels of scalable performance because clients can be dispersed throughout a large set of servers. Adding additional servers naturally allows more clients to participate. This happens because clients will either be directed to servers that have the requested data or should those servers near saturation levels, clients are directed to less active servers that will replicate the requested data. Hence, the load will be balanced across all of the servers. Unanticipated hot spots are naturally alleviated because the protocol allows any server that finds itself in a hot-spot to redirect clients away from itself. This forces clients to settle upon other servers that are less loaded. We call this mode of operation "dynamic load balancing" and it is one of the major reasons that the system scales.

Since the protocol allows connection configuration changes to occur at any time, the system also provides unprecedented fault tolerance. Should a server failure occur, a client needs only to contact a redirector to find an alternative source of the data.

The olbd Process

As we mentioned in the previous section, the system was designed using an independent set of servers to

provide the control information to effect xrootd server selection. This set of independent servers forms what we call the control network. It is logically independent of the data access services provided by the xroot servers; which form the data network. Either network can be replaced in total. Indeed, the we have shown that the structure can work as well with xroot servers as it can with Objectivity AMS servers in a production environment. The only requirement is that the protocol allows clients to be redirected to other servers; something the Objectivity protocol allows.

The control network is made of servers called Open Load Balancing Daemons (olbd), Each node that provides data must have an xroot server and an olbd running on it. The olbd runs in what we call server mode since it is responsible for relaying information about the node providing a data service.

The xroot server running on a data node connects to the local olbd. This allows the olbd to know the status of the server and the port number that it is using. This information is relayed to other olbd's so that clients can be properly directed to the data node, as needed.

Each local olbd subscribes to one or more olbd's running in manager mode. A subscription effectively tells the target olbd that the node is capable of providing a data service on a particular port. Identification of manager olbd's is done by administrative configuration. This is not an odious task because there are only a handful (usually two) of olbd's running in manager mode.

A manager olbd is special in that it resides at the top-most level of the connection hierarchy. It differs from server mode olbd's in that it accepts connections from multiple xroot servers. These xroot servers form the redirectors. That is, clients making requests of these servers are always redirected to appropriate servers lower in the hierarchy. Redirectors do not provide data only request steering information.

The xroot servers that connect to the manager olbd's are administratively configured to ask the manager olbd where to direct the incoming client request. Again, this configuration is simple since there is no need to have more redirecting xroot servers than manager olbd's.

When a manager olbd is asked for request guidance, it first checks its cache of recent requests to see if it already knows where the request should be sent. If the information exists in its cache, the response is immediate. Otherwise, the redirecting xroot server is told to delay the client for a fixed period of time while it asks the olbd's that are subscribed to it whether or not they have the requested file. All olbd's that have the file report its existence. Those olbd's that cannot find the file on their node stay silent. File existence information is collected by the manager olbd and cached. Eventually, the client comes back and asks for the file which prompts xroot server to ask again. This time the information is in the cache and the response is immediate.

CONCLUSIONS

In building the xroot system, we have shown that it is possible to construct a large loosely coupled highly distributed data access service that exhibits an unprecedented degree of scaling. In the process we discovered that a fundamental paradigm shift needs to occur on what constitutes a scalable system and the algorithms that need to be employed to achieve that goal.

Based on our experience, we can state seven fundamental rules of scalable systems:

1. Basic building blocks must exhibit high performance and low latency,
2. Client requests must be dispersed throughout the system as quickly as possible. This argues against top-heavy systems where significant decisions are made when a request first enters the system. Instead, such decisions should be distributed and performed as late as possible.
3. The amount of information flow in the system must be minimized. This argues that information about any component within the system is only as accurate as it is close to the relevant component that the information describes; further bolstering the argument for distributed decision making as well as systems that attempt to micro-manage the request flow.
4. Latencies within the system must be kept as even as possible, even with it means that latency has to be introduced to achieve that goal. In some sense, when all points exhibit the same latency, the system is capable of "pipelining" requests and can thus reach maximum throughput.
5. A system is scalable in proportion to its fault-tolerance. The larger the system, the more fault-tolerant it needs to be. This seems counter-intuitive until one realizes that scaling not only involves servers but clients as well. The larger number of servers the larger number of clients. Systems that are not fault tolerant tend to exhibit large fluctuations in client request load as components fail. Fault tolerant systems tend to even out those fluctuations avoiding request avalanches.
6. Scaling is a two-way street. Servers and clients must be full participants in the information flow.
7. Scalable systems are limited by administrative overhead. If the administrative overhead grows in direct proportion to the size of the system, scaling becomes unsustainable simply because of the human cost in maintaining the system and the consequent human errors inherent in maintaining large systems. This argues for self-configuring systems.

REFERENCES

[1] http://xrootd.slac.stanford.edu.