

Scalla: Scalable Cluster Architecture for Low Latency Access Using xrootd and olbd Servers

Chuck Boeheim, Andy Hanushevsky, David Leith, Randy Melen,
Richard Mount, Teela Pulliam, Bill Weeks
Stanford Linear Accelerator Center
(boeheim, abh, leith, randym, rmount, teela, wcw@slac.stanford.edu)

Abstract

The Scalla software suite provides two fundamental building blocks: an xrootd server for low latency high bandwidth data access and an olbd server for building scalable xrootd clusters. This paper describes the architecture, how low latency is achieved, and the scaling opportunities the software allows. Actual performance measurements are presented and discussed. Scalla offers a readily deployable framework in which to construct large fault-tolerant high performance data access configurations using commodity hardware with a minimum amount of administrative overhead.

1: Introduction

At its core, the Scalla software suite consists of a file server called xrootd[4]¹ and a clustering server called olbd². The server names are historical. The xrootd server was developed for the Root analysis framework to ostensibly serve root files. However, the server is agnostic as to file type and provides byte level access to any type of file. The olbd server was originally developed to cluster and load balance Objectivity/DB AMS³ database servers. Because the olbd was designed to work independently of the data server, it was easily usable with xrootd.

Nothing in Scalla requires that xrootd be run with the olbd. Indeed, in simple environments xrootd can be used in a stand-alone fashion. However, for effective scaling in the presence of multiple file servers, the olbd is used to federate all of the xrootd servers into a common name space.

¹ eXtended Root Daemon which replaced the original rootd in the analysis framework.

² Originally, named Objectivity Open Load Balancing Daemon (oolbd). Since renamed to be simply Open Load Balancing Daemon.

³ Advanced Multithreaded Server

The following sections describe the xrootd and olbd architectures and how the two servers work together to provide multi-dimensional scaling. In the final section, Scalla performance characteristics are presented.

2: xrootd Architecture

The xrootd server is designed to provide POSIX-like access to files and their enclosing directory namespace. The architecture is extensible in that it relies heavily on a run-time plug-in mechanism so that new features can be added with a minimum of disruption. The plug-in components are shown in Figure 1.

Seven plug-in components are shown. The components mate (i.e., plug in) at different architectural junctions.

xrd Layer (Threading, Memory, Protocol Driver)	
Protocol Plug-in (static xrootd protocol)	Authentication Plug-in
Logical Filesystem Plug-in (libXrdOfs.so)	Authorization Plug-in
Physical Filesystem Plug-in (libXrdOfs.so)	Name-2-Name Plug-in

Figure 1: xrootd Server Architecture

2.1: The xrd Component

The core component is the “xrd”. This component is responsible for network, thread, data buffer, and protocol management. Because the “xrd” is responsible for a compact set of functions, it was easily optimized to do them exceedingly well. For instance, network management was engineered to use the most efficient mechanism available for each type of host operating

system⁴. Data buffer management is optimized to provide fast allocation and de-allocation of I/O buffers on page boundaries. Protocol management is designed to allow any number of protocols to be used at the same time. The protocol is selected at the time an initial connection is made to the server.

2.2: The xroot Protocol Component

By default, the component that provides the xroot protocol is statically linked with the “xrd”. As mentioned before, additional protocols may be specified, and the “xrd” loads these at run-time from appropriate shared libraries. For instance, the PROOF[1]⁵ system runs both the xroot protocol as well a special protocol that provides parallel access to multiple data analysis servers within the Root Framework.

The xroot protocol is optimized to provide the lowest possible latency for network data access. This is accomplished by using a compact network byte order binary request and response headers (i.e., 24 bytes for the request and 8 bytes for the response). Additionally, the protocol allows request multiplexing on a single connection. In asynchronous request mode, multiple requests may be issued concurrently. This allows for parallel as well as pipelined data access to one or more files. The server is free to choose the optimum order in which the requests are satisfied without placing an undue burden on the client. For low latency WAN access as well as small request LAN access, multiple requests may be combined into a single transaction, minimizing the number of server interactions per byte of data. The protocol also allows client-directed parallel access using multiple connections, read-ahead hints, as well as the full set of operations required for POSIX file access.

Clustering elements are naturally integrated into the protocol. This allows an xroot server to run seamlessly with or without clustering (i.e., olbd). The clustering elements are described in the olbd section.

From an engineering stand-point, the protocol is implemented using an optimistic run-to-completion transaction model. In this model, as long as the client issues requests within a reasonable time window, a thread is dedicated to the client. This dramatically reduces latency for active clients because much of the OS overhead involved with thread switching and network polling is eliminated. Another optimization used is to run the read/write code path without mutex locks, moving

potential synchronization points into the OS kernel where lock handling is most efficient.

To further reduce overhead in a multi-CPU environment, the implementation avoids sharing data buffers between threads. This effectively eliminates much of the costly overhead in cross-CPU memory cache synchronization. A special buffer sizing algorithm is used to minimize the memory foot-print as memory usage would tend to grow when using unshared buffers. Finally, buffer management is completely eliminated when the client requests access to a memory mapped file.

2.3: The Authentication Component

The authentication component, XrdSec, plugs into the xroot protocol component. Multiple authentication protocols can be used as the xroot protocol is merely used to encapsulate the client/server interactions required by the protocol. Currently, GSI, Kerberos IV and V, as well as simple password authentication are supported. Additional authentication protocols may be implemented and placed in shared libraries. These protocols are dynamically loaded and used whenever the client supports the particular protocol. Authentication models may also be restricted on a host name and domain basis.

By default, the authentication component is not enabled and only host based authentication is available. This is done for those installations that require the minimum amount of file access overhead when the value of the data does not warrant full authentication control.

2.4: The Logical File System Component

The file system component, XrdSfs, also plugs into the xroot protocol component. The interface to the file system uses an enhanced but otherwise typical object oriented file model providing for a full set of POSIX operations. The major enhancements include opaque hints, clustering support, and worm-hole call back objects. The latter is an important element in minimizing the overhead associated with object oriented interfaces by reducing the number of call layers to accomplish a task without exposing the underlying implementation. The call back mechanism also supports cross-server call backs, necessary to reduce the latency inherent in clustering protocols.

The default implementation that is statically linked with the xroot component provides basic access to the underlying file system. A run-time selectable implementation, XrdOfs, that fully supports asynchronous I/O, memory mapped files, an authorization plug-in, and clustering is also provided as a shared plug-in library (i.e., libXrdOfs.so).

⁴ Xrootd runs in 32- and 64-bit Linux, Solaris (SPARC and x86), and MacOS.

⁵ Parallel Root Facility

2.5: The Authorization Component

The authorization component, XrdAcc, is supported by the special file system component, XrdOfs. The default implementation is statically linked with XrdOfs and provides ACL-like access control via capability lists on the name space. The authorization component uses the authentication information provided by the XrdSec component to make POSIX compatible file access decisions.

By default, the authorization component is not enabled and clients have full access to all exported files. This is done for those installations that require the minimum amount of file access overhead when the value of the data does not warrant full authorization control (e.g., public read-only files).

2.6: The Physical File System Component

The physical file system component, XrdOss, is supported by the special file system component, XrdOfs. The default implementation is statically linked with XrdOfs and provides POSIX-like access to physical files. The reason for the architectural split into logical and physical layers is to compartmentalize functionality, allowing for a much cleaner implementation. The logical layer deals with file system implementation independent functions such as clustering and authorization. The physical layer must deal with the actual file system implementation being used.

The default XrdOss implementation supports aggregation of multiple file systems under a common name space and an interface to a Mass Storage System. The latter allows for files to be automatically migrated as well as retrieved from other, possibly taped-based, storage systems. Additionally, the XrdOss component is responsible for implementing the asynchronous I/O model and controlling memory mapped files. Both of these are highly dependent on the underlying OS implementation. The asynchronous I/O model relies on the worm hole call back mechanisms to substantially reduce the read/write code path length.

2.7: The Name2Name Component

The Name2Name component, XrdOucN2N, is a run-time selectable plug-in for the XrdOss component. This component is responsible for translating a logical file name to a physical file name. The default implementation is statically linked with XrdOss and, depending on configuration file specifications, merely adds a prefix to the logical file name. Since files can exist either in the

underlying local file system or in a remote file system accessible via the MSS interface, the Name2Name component can be configured for different translations for local and remote file access.

3: olbd Architecture

The olbd server is designed to provide file location functionality and cluster health and performance monitoring. The architecture is extensible in that it also relies on a run-time plug-in mechanism so that new features can be added with a minimum of disruption. The plug-in components are shown in Figure 2.

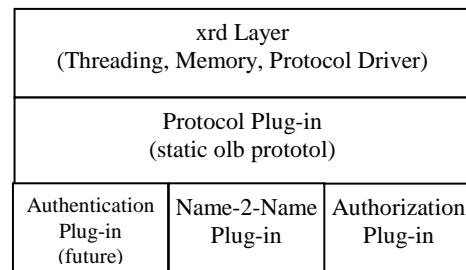


Figure 2: olbd Server Architecture

Two plug-in components are shown. These components are loaded and used by the olb protocol component.

The olbd uses a structured hierarchical subscription model. That is, olbd's connect to other olbd's in order to form a compact B-64 tree, as shown in Figure 3.

A special olbd, called the redirector, sits at the root of the tree. This is typically known as the *head node* in traditional clusters. In an olbd controlled cluster this server is given the role of a *manager*. The manager is responsible for issuing file queries and collecting the responses from nodes lower in the tree.

Since this is a B-64 tree, each node can only accommodate 64 sub-nodes. When more than 64 nodes exist, additional olbd's must be given *supervisor* roles. The function of a supervisor is identical to that of a manager in that it accepts connection from other olbd's, issues file queries and collects responses from nodes lower in the tree. However, a supervisor olbd also connects to another olbd higher in the tree. From the manger's perspective a supervisor is simply another node that can respond to a file query.

The leaves of the tree are given *server* roles. A server olbd is in 1-to-1 correspondence with a data server (i.e., a machine that serves data files).

This kind of architecture scales very quickly (i.e., $O(h^{64})$) with a minimum amount of message traffic. A tree of height 1 accommodates 64 nodes while a tree of height 3 accommodates 65,536 nodes. The limit of 64 nodes is deliberate. Sixty-four allows efficient bit-slice

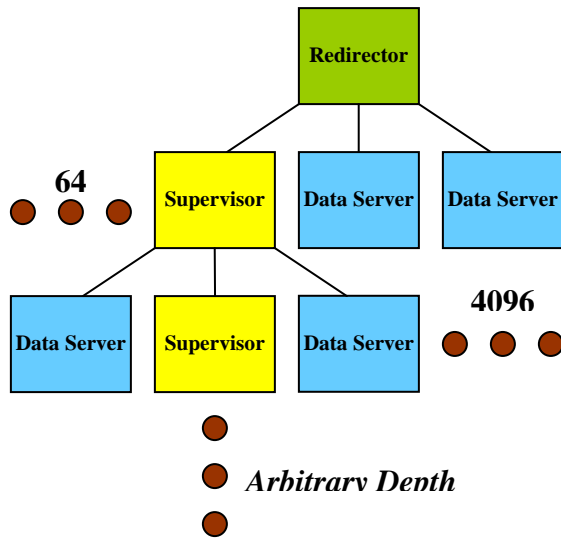


Figure 3: Olbd B64-Tree Architecture

vector operations using 64-bit integers and deterministically limits the amount of work any particular server needs to do. The latter is critical in providing a deterministic time bound for file queries.

A subscription model is ideal in that a single configuration file for all nodes can be used to describe the nature of the cluster. Minimally, all nodes must know the manager's location. This allows each node to connect to the manager and if the manager's sub-node limit has been reached, the manager tells the incoming node to connect to a supervisor lower in the tree. In turn, if the supervisor's limit has been reached, the supervisor informs the incoming node where it must connect lower in the tree. As supervisors always have precedence over servers, supervisor nodes naturally migrate to the top of the tree while server nodes sink to the bottom. A breadth first connection strategy is used to keep the tree at a minimum height. Hence, the actual connection structure need not be pre-defined. The olbd's automatically structure their connections to form a B-64 tree. This greatly simplifies the configuration of large clusters and allows nodes to be added on-the-fly.

To provide for fault-tolerance and scalability, the manager node (i.e., root of the tree) can be replicated. When this is done, the tree is automatically configured so that that all nodes are reachable from all managers. Replicated managers always function in fail-over mode. Additionally, the file query load can be distributed across all of the managers using a hash function based on the name of the subject file.

Supervisor fault-tolerance is accomplished by merely designating more olbd's to have supervisor roles than is actually needed. Thus, if a supervisor fails, its

nodes are automatically redistributed across the remaining supervisors.

3.1: The xrd Component

The core component is the "xrd". This component is identical to the one used by xrootd. Thus, the olbd benefits from the performance features engineered into the "xrd".

3.2: The olb Protocol Component

By default, the component that provides olb protocol is statically linked with the "xrd". As mentioned before, additional protocols may be specified and the "xrd" loads these at run-time from appropriate shared libraries. For instance, the PROOF[1]⁶ system runs both the olb protocol as well a special protocol that provides clustering for multiple data analysis servers within the Root Framework.

When the manager is asked for the location of a file, it first checks its look-aside memory cache to see if it already knows the location of the file. If the file is found in its cache, the manager responds with the name of the host and the port number of the file server. If more than one node has the file, the manager either chooses the least chosen node (i.e., round-robin) or chooses the least loaded node. Load is defined by a configuration file formulae and can include CPU usage, network bandwidth available, among other factors.

If the file is not found in the cache, the manager issues a query to all of its nodes that could potentially have the file. Not all nodes may be eligible since a topological name space can be overlaid on the B-64 tree⁷. As supervisors are just specialized managers, they in turn rebroadcast the query to their nodes and respond that they have the file should at least one sub-node respond affirmatively.

The query is implemented using a request-rarely-respond protocol. That is, if a node has the requested file, the node must respond. If the node does not have the file, it does not respond. The R³ protocol is provably the most efficient protocol if less than half the nodes have the file[3]. It is also provides the least latency if at least one node has the file. This is typical in large scientific data clusters for relatively long periods of time.

If no response is received within a fixed time window (e.g., 5 seconds), the manager looks for a node

⁶ Parallel Root Facility

⁷ This is rarely done since it is exceedingly difficult to define a topological name space that is uniformly reachable across a B-64 tree.

that has declared itself capable of hosting the file. Nodes declare themselves capable when they connect to the manager or supervisor. Such nodes typically are interfaced with Mass Storage Systems and can retrieve the file from a remote location. The additional wait time is easily hidden by the time it takes to retrieve the file and does not represent a significant latency burden.

The use of a just-in-time query means that the name space is built dynamically and can change whenever the circumstances warrant. No persistent data is maintained which eliminates synchronization problems and administrative overhead. Nodes are responsible for notifying their immediate superior node should any significant changes occur. For instance, when a file is created or an existing file is removed the leaf node notifies it's parent node that of file deletion. Nodes can also temporarily rescind their file hosting declaration.

While the majority of time is spent performing file queries, the olbd nodes are also responsible for monitoring each other's health. Each node periodically sends a heartbeat message to its sub-nodes and also asks for load information. Load information is automatically sent by any node whenever a significant change occurs in order to maintain timely information. This allows a manager or supervisor to avoid overloaded and non-working nodes. The process is optimistic in the sense that a manager or supervisor will defer queries for a limited amount of time in the hopes that a non-functioning node will come back to life should it have a file of current interest.

Conversely, supervisor and server nodes monitor that heartbeat messages are in fact being sent. If heartbeat messages are not received, the supervisor or server automatically tries to find another manager or supervisor to subscribe to. This mutual monitoring allows the complete B-64 tree to be virtually always connected to within the provided level of fault-tolerance.

The olbd protocol does not provide transactional consistency. As the B-64 is potentially always in flux and no persistent information is maintained, it is impossible to know exactly the state of all files in the system. For instance, determining all the locations of a file is not possible since not all nodes may be connected at the time the question is posed. This effect is minimized in that when the configuration of the tree changes on a particular path, all superior nodes are notified that any cached file information along that path should be discarded. A partial cache invalidation scheme is used to avoid resending queries along healthy paths.

Since managers essentially concentrate file location information, a fixed window cache algorithm is used to place a bound on the size of the cache. Typically, cached information that is more than 8 hours old is discarded whether or not the file has been recently used in a query.

Using a trivial algorithm to deterministically limit memory use greatly outweighs the overhead of occasionally re-broadcasting a query for an already known file.

The olb protocol also allows for certain name space operations (e.g., remove, rename, etc) to be forwarded to all nodes in an attempt to maintain a consistent name space. However, as previously mentioned this is not semantically effective in a non-transactional dynamic system.

3.3: The Name2Name Component

The Name2Name component, XrdOucN2N, is a run-time selectable plug-in for the olb protocol component. This component is responsible for translating a logical file name to a physical file name. This component is identical to that used by xrootd.

3.3: The xmi Component

The olb protocol allows one or more of its methods to locate files as well as manipulate the name space to be overridden by the xmi component. This is a run-time plug-in that, when loaded, tell the olbd which method calls are to be forwarded to the xmi component. This component is used to implement other types of clustering mechanisms while still maintaining the capability of supporting an xrootd-based cluster.

4: Relationship between xrootd and olbd

An xrootd server provides a uniform interface to a client using the xroot protocol. It is exclusively responsible for providing data and name space operations. Clients always connect to an xrootd server with the expectation that the server will perform the requested operation. However, an xrootd server is always free to redirect the client to another xrootd server⁸. This is the mechanism used to maneuver the client through a cluster of xrootd servers until the client reaches one that can actually perform the requested operation.

Xrootd servers rely on olbd servers for information on how this redirection is to occur. Thus, each xrootd is paired with an olbd in 1-to-1 correspondence. A simple configuration is shown in figure 4.

Data node xrootd servers pair with a server role olbd. The olbd is responsible for monitoring the health of

⁸ Redirection may even occur in the middle of a request; allowing for dynamic reconfiguration and real-time client load redistribution.

the xrootd and also responds to file query and heartbeat requests from its superior olbd. The xrootd also sends the olbd information on any name space manipulations that may have occurred. The olbd is responsible for relaying that information to its superiors.

A supervisor node xrootd pairs with a supervisor role olbd. Since supervisors are almost identical to managers, the relationship between a manager xrootd and olbd applies here.

A manager node xrootd pairs with a manager role olbd. When, for instance, a client connects to a manager xrootd and attempts to open a file, the xrootd asks its olbd for the location of the file. When the xrootd receives a response, it simply redirects the client to the xrootd server the olbd selected. This can be an actual server or merely a supervisor that will, in turn, redirect the client to a node lower in the tree.

A small variation occurs at the manager level since there may be more than one manager. In order to provide full redundancy and allow for load balancing, a manager xrootd connects to *all* of the manager olbd's.

This fully symmetric relationship simplifies configuration since there are no exceptions regardless of the role each server plays in the cluster. Furthermore, since a redirection model provides clients only point-to-point connections, I/O operations are never slowed.

There is no limitation on how many xrootd/olbd pairs can run on a single system. The cluster maintains its integrity without special configuration by each server using an arbitrarily available port number and affiliating

with a logical network name specified at start-up time. This allows for any number of self-cohesive but otherwise overlapping clusters with respect to hardware. The only restriction is that the manager node xrootd and olbd must be assigned well known port numbers for each respective cluster.

5: Client Access

Access to xrootd servers is provided by a medium-weight client-side object called XrdClient. The choice to use a medium-weight client was driven by the need to optimize data access in ways that only was visible to the client. For instance, the client is responsible for caching the appropriate data, providing pre-read hints, managing multiple parallel paths, pipelining, parallelizing, and aggregating requests. All of these allow the client to optimize both latency and throughput based on the conditions relevant to the client. This strategy is far more effective than having the server attempt to guess what an optimum strategy might be.

The XrdClient object provides all of the usual POSIX-like methods to access data, create files, and manipulate the name space. Additional application level methods provide ways for the client to specifically optimize data access (e.g., request aggregation).

A fully functional true POSIX interface is also available. This is accomplished by a preload shared library that intercepts the usual POSIX calls and routes them either to the local file system or to XrdClient, based

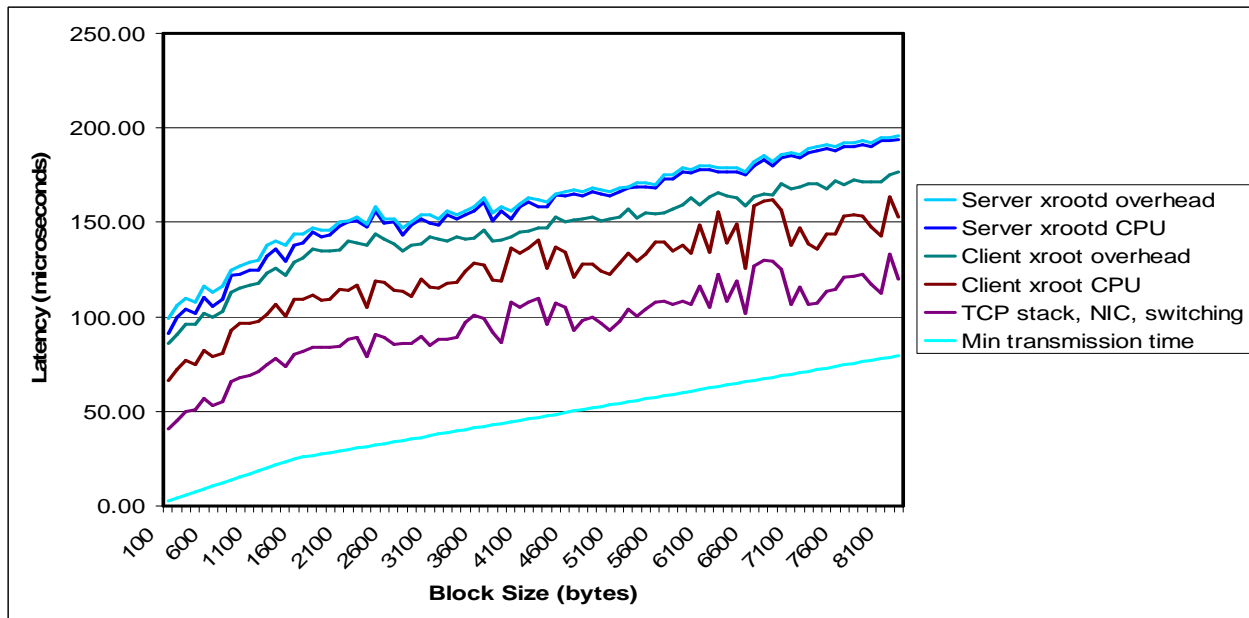
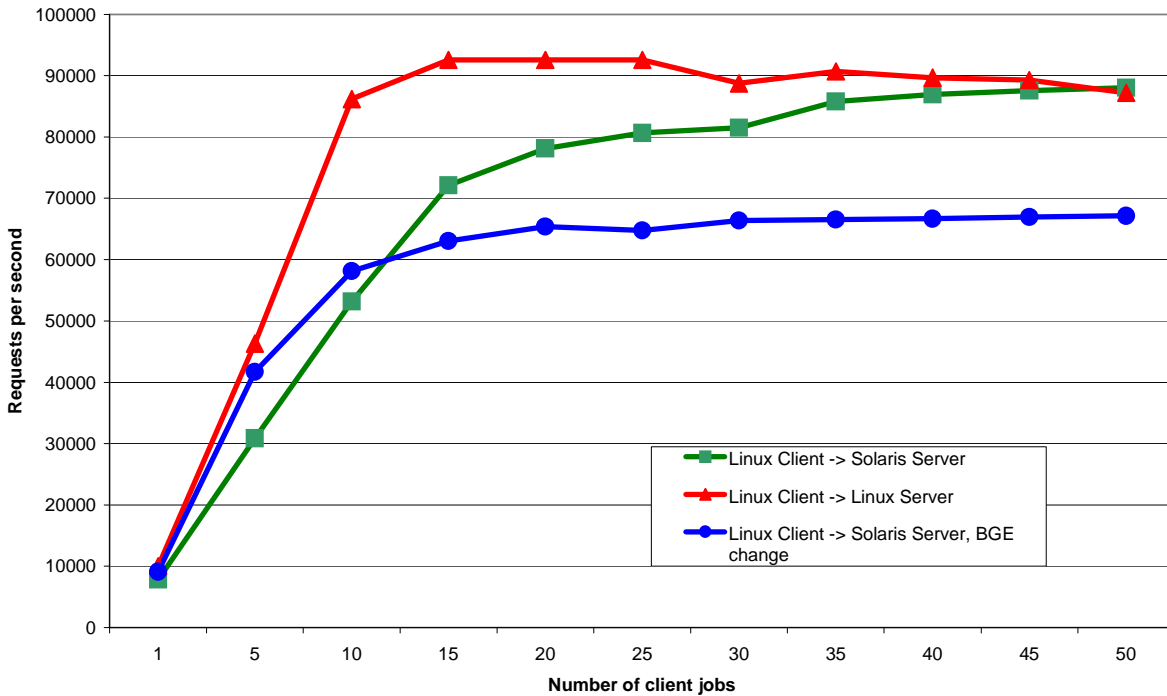


Figure 5: Stacked Latency vs Block Size for Linux client/server. Fluctuations are artifacts of measurement technique.

Figure 6: Single Server Throughput



on the file name. This interface does not provide application specifiable optimizations. It does, however, allow applications to use Scalla without change.

Finally, a linkable POSIX interface library is provided for those cases where a preload library is not acceptable.

6: Performance and Scalability

The following measurements were done on a cluster composed of 64 dual-CPU systems consisting of commodity V20z's[2] from Sun. Each system contains dual Opteron 244 (1.8GHz) processors, a 36GB system disk, a 73GB persistent-data disk, and eight memory slots with 2GB DIMMs, for a total of 16GB. Each system has two gigabit Ethernet ports, of which only one is used.

Both Solaris 10 x64 and Red Hat Enterprise Linux 3 are installed on these systems in a dual-boot configuration.

The cluster is connected to a Cisco 6509 switch with copper gigabit Ethernet. The switch has four bonded gigabit Ethernet connections to a batch farm. The batch farm consists of 2,200 nodes (3850 processors) ranging from 440MHz Ultrasparc II to 2.0GHz Opterons. Each of these nodes has a 100 megabit connection to one of ten Cisco

6509 switches which are interconnected with multiple gigabit Ethernet backbones.

6.1: Latency Measurements

Figure 5 shows the latency, for access to objects in already open files, being dominated by the network transmission time. The highly optimized server code contributes very little to the overall transaction time, while the less optimized client code shows some room for improvement. In these measurements with a single client, the elapsed time for xrootd execution ranged from 13 microseconds for small transactions to 21 microseconds for 8kByte transactions. This test demonstrates that we can effectively use server DRAM storage with little overhead and at network speeds.

Measurements were also made of the much higher

Client: Server:	Linux Linux	Linux Solaris	Solaris Linux	Solaris Solaris
1st Open via xrootd only	7.7	7.7	4.5	4.5
2nd Open via xrootd only	5.5	5.5	1.1	1.1
1st Open via olbd+xrootd		11.0		5.6
2nd Open via olbd+xrootd		8.0		1.2

Table 1: Millisecond latencies for xrootd file open operations on files that are memory resident on the server

latencies involved in the first access to a file. These are shown in Table 1 for clients accessing files via xrootd alone, and for clients accessing files via both olbd and xrootd. The relatively poor performance of Linux clients is striking. We do not have any obvious explanation for this difference. However, open operations are currently a small contributor to overall latency.

6.2: Throughput Measurements

To test the throughput and robustness of the xrootd-based sever under heavy load, an increasing number of concurrent clients was run against a single server. The client code did no computational work on the retrieved data, but just read data as fast as possible; issuing a new request for data as soon as the previous request has returned. The Solaris “bge” network driver suppresses the attempts to bundle processing of network packets. The results are shown in Figure 6. A saturated transaction rate of over 90,000 transactions per second could be obtained with a Linux server. There was no sign of any pathological behavior under heavy load. The maximum transaction rate represents 22 elapsed processor microseconds per transaction. This can be compared with the 13 elapsed microseconds per transaction consumed by xrootd when serving a single transaction stream.

6.3: Scalability Measurements

Figure 7 shows how a single server scales with increasing number of clients. This view is different from a latency perspective in that it shows CPU consumption vs. number of clients, as well as service delivery. The graph clearly shows linear scaling allowing for deterministic server-sizing.

Cluster scaling is equally important. There are two aspects to the scaling: 1) setup time, and 2) run-time overhead.

To test setup time, a large Linux-based servers cluster was created by taking 890 machines out of the SLAC batch data-processing system and configuring them as identical xrootd/olbd servers. The machines started running the xrootd/olbd software over a period of 30 seconds (the LSF batch system had problems trying to start 890 machines faster than this). It took 86 seconds from the time the first machine started until all 890 machines had self-organized into a data-serving cluster. In comparison with the earlier test using only 280 machines indicates that the self-organization time

increases very approximately as the square of the number of machines. This effect was traced down to the fact that servers were competing with supervisors for cluster slots; causing significant collision delays. Had all the supervisors been started before the data servers, we would expect to see $O(\log_{64}(N))$ scaling in setup time. Since it is not always possible to sequence the start-up order; randomly starting 10,000 servers would require over one hour for them to self-organize. We are currently working to improve the scalability to allow fast construction of clusters of this size.

Based on Table 1, run-time clustering overhead introduces approximately $100 \cdot \log_{64}(N)$ microseconds per open request; where N is the number of servers. This is an excellent number and the scaling factor comfortably accommodates very large clusters.

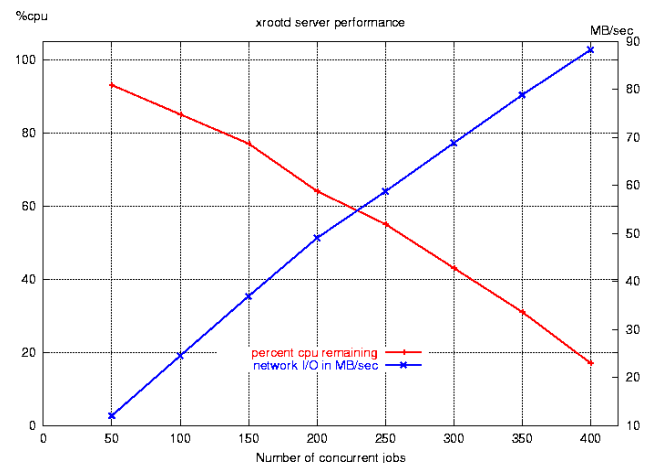


Figure 7: Single Server Scaling

7: Acknowledgement

Work supported by the U.S. Department of Energy under contract number DE-AC02-76-SF00515.

References

- [1] Maarten Ballintijn, Gunther Roland, Rene Brun, Fons Rademakers; [The PROOF Distributed Parallel Analysis Framework based on ROOT, CHEP Conference](#), La Jolla, California, March 2003
www.slac.stanford.edu/econf/C0303241/proc/papers/TULT003.PDF
- [2] <http://www.sun.com/servers/entry/v20z>
- [3] Fabrizio Furano, [Andrew Hanushevsky](#): Managing commitments in a Multi Agent System using Passive Bids. [IAT 2005](#): 698-701
- [4] <http://xrootd.slac.stanford.edu/>