



Xyratex Lustre Architecture Priorities Overview

Dr. Peter J. Braam, Dr. Nikita Danilov & Nathan Rutman

Networked Storage Solutions

x y r a t e x •

Notices

The information in this document is subject to change without notice. While every effort has been made to ensure that all information in this document is accurate, the document is provided “as-is” and Xyratex accepts no liability for any errors that may arise. Reliance on this document and its content is at the sole risk of the viewer or reader.

©2011 Xyratex (the trading name of Xyratex Technology Limited). Registered Office: Langstone Road, Havant, Hampshire, PO9 1SA, England. Registered number 03134912.

Xyratex Permissions: this document, and portions thereof, may be copied unaltered, provided Xyratex is identified as the copyright holder. All uses of the document or portions thereof MUST provide attribution to Xyratex. In no circumstance does Xyratex approve of any use of the document or its content without proper attribution. The attribution must be legible to the average viewer or reader. Requests for exceptions will not be answered.

Xyratex is a trademark of Xyratex Technology Limited. All other brand and product names are registered marks of their respective proprietors.

For more information, please contact marketing@xyratex.com

Issue 1.0 | 2011

Introduction

This paper summarizes a technical approach for the Lustre community to consider implementing new features that will contribute to dramatically increasing Lustre’s stability, scalability, and performance.

What distinguishes this partial roadmap is that Xyratex proposes that the community collaborate to replace components that currently have implementation issues, some dating back to the original Lustre development roadmap, with more modern designs. Several features of this proposal to the community complement existing approaches, while other innovative aspects replace existing designs entirely.

The purpose of issuing this document is to provide some transparency and insight into Xyratex’s architectural thoughts to extend Lustre capabilities and to stimulate discussion within the community. We want to help drive an overall community roadmap that can be achieved through collaborative development with other organizations within the Lustre community.

1. File I/O

1.1 Changes in the I/O Path – Large I/O

It is well documented that large I/O transfers to disk drives reach peak performance when I/O transfer sizes are around 4MB. Figure 1 shows the bandwidth achieved (MB/sec), as a function of buffer sizes, when performing repeated read and write operations of a given buffer size to random locations on 1TB FAT SAS Seagate Barracuda ES.2 (ST31000640SS). Lustre does not achieve this optimal buffer size for reasons explained below. Several publications have confirmed a detrimental effect of Lustre’s 1MB disk transfers, e.g. see the LLNL poster study at SC 2010.

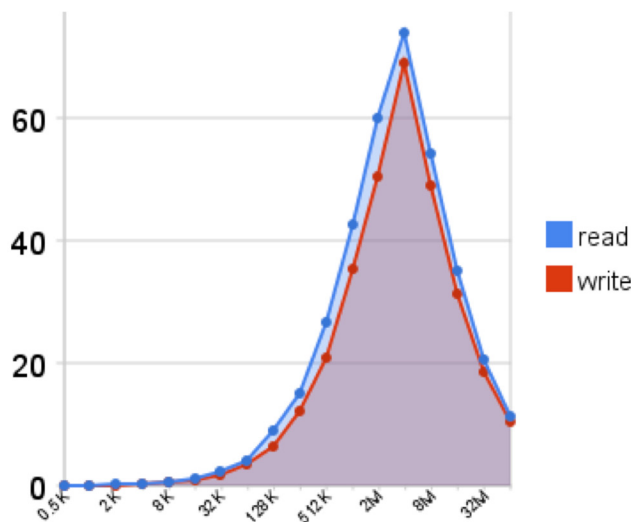


Figure 1: X axis: I/O size (bytes); Y axis: throughput MB/sec

To improve Lustre’s transfer rate, multiple challenges must be overcome. Lustre’s network RAID0 striping may distribute a large I/O stream over multiple OSTs. Further, the OST backend typically utilizes a RAID engine that may break up a 4MB I/O block over multiple drives. Additionally, Lustre’s maximal buffer size for a single network transfer is capped at 1MB, and metadata associated with OST data is typically small. Presently, when 4MB transfer sizes are achieved, this happens because the block device disk scheduler (by coincidence) aggregates sufficient data.

Yet in many cases, Lustre can deliver 4MB I/O transfers by aggregating several smaller transfers with 1MB of data in the block device scheduler. Our goal is to improve I/O throughput and make it more likely that 4MB I/O buffers will aggregate in the disk device drivers. The following small changes to Lustre will enable large I/O transfers and improve performance.

Large Network I/O. One change is to allow multiple 1MB chunks to travel from client to server¹ as part of a single, OSC-based read or write RPC. Rather than implementing fundamental changes to enable I/O buffers larger than 1MB, Lustre can conduct multiple transfers as part of the RPC, before the OST backend initiates I/O. Lustre patches to enable these transfers have already been developed, see Bugzilla [16900](#) (4MB sequential I/O RPCs). Figure 2 shows multiple network RDMA transfers initiated from a single write RPC.

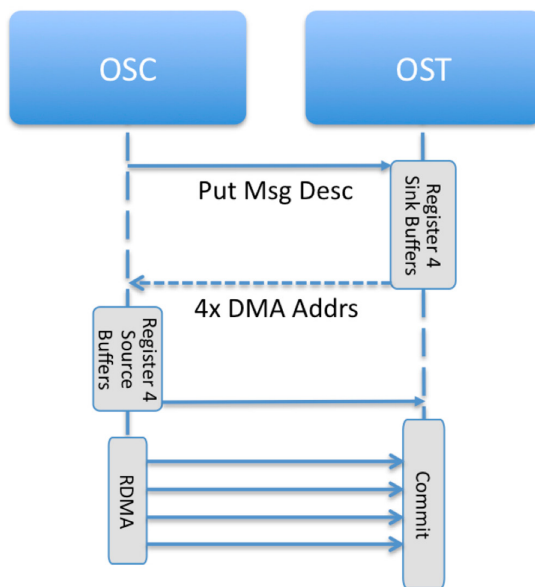


Figure 2: Multi-megabyte bulk I/O

Fast Caches. Second, a very fast persistent cache will be introduced in the OSS, to aggregate multiple writes in battery-backed memory or flash storage before writing them to disk. This cache, a tuned version of Linux Bcache or Flash Cache, will be shared or mirrored between controllers to retain the same semantics as Lustre without a cache. The implementation and tuning of these fast caches is beyond the scope of this paper and will not be discussed further.

Separate File and Metadata. Third, as is commonly known and was perhaps first implemented in a Cray file system well over a decade ago, storing data and metadata on separate volumes improves I/O throughput considerably. By leveraging virtual volumes in Linux and the format options of the Lustre disk file system on the OSS, one can format the file system so its metadata resides on a segment of the LVM backed by a RAID1 volume and the data resides on a segment backed by a customary RAID6 volume. The result is that small I/O for metadata does not incur a read-modify-write penalty (because RAID1 volumes do not have this I/O mechanism), and traffic to the RAID6 volume does not contain metadata and often writes full stripes.

¹ Because the OSS read cache feature provides read-only caching of data on the OSSs, the size of the network transfer from server to client, for read operations, is less critical.

1.2 Changes in the I/O Path – Small File I/O

Lustre performance with small file I/O will benefit from the changes described above. Lustre already benefits from a feature of the disk file system allocator that groups small files together. This advantage, combined with a fast persistent cache, will lead to improved performance for small file I/O because the cache will allow multiple small files to be aggregated into larger, contiguous extents before being written to disk.

1.3 Network Request Scheduler

In clusters, typically, many nodes together perform I/O to a single file. On a per-client basis, such I/O may seem to hit a near-random part of a file and the servers, therefore, perform random I/O by executing I/O requests from clients in a traditional request FIFO queue pattern. However, the collective effect of such I/O commands – i.e. the aggregate of the read or write commands – is often equivalent to writing an entire file sequentially, which is a significantly more favorable I/O pattern.

The purpose of the Network Request Scheduler (NRS) is to re-order such collective I/O into an effective, sequential stream while maintaining consistency with read-write ordering semantics and locking. The NRS optimizes throughput by implementing several basic policies, such as:

- Fair client I/O scheduling
- Prioritized clients

In ORNL's large-scale, Jaguar Lustre cluster, I/O traffic was routed to achieve fair load balancing of I/O across the Lustre clients. Using the NRS resulted in a near 2x increase in performance. The fair client I/O scheduling policy in the NRS may eliminate the need for such I/O routing. The prioritized clients policy can address the longstanding request for some form of I/O prioritization, particularly when visualization and compute clusters compete for access to the file system.

1.4 File Layouts – Wide Striping and Data Placement

Data Layout Descriptors. Currently, Lustre is limited to striping across approximately 160 targets. Increasing this stripe limit (wide striping) is needed to achieve maximal bandwidth to a single file when there are a large number of targets. A better approach than supporting ever-larger extended attributes in the inodes is to introduce a different file layout mechanism. First, a FID-based object store (already in use by the MDS) should be used for the OSS targets. With the FID-based object store, a group of FIDs can be formed that describes files with an identical FID on all targets. The description of a file using such FIDs is very compact. Second, a new "wide striped" layout type is defined that indicates the FID and the specific OSTs containing stripes for that file. This could include a bitmap, which should accommodate up to ~32,000 stripes within the current ext4 EA size limit, or the layout could indicate that the file is striped over all targets². Figure 3 illustrates the new layout types.

² Care needs to be taken to allow the addition of OSS nodes without invalidating existing stripe descriptors.

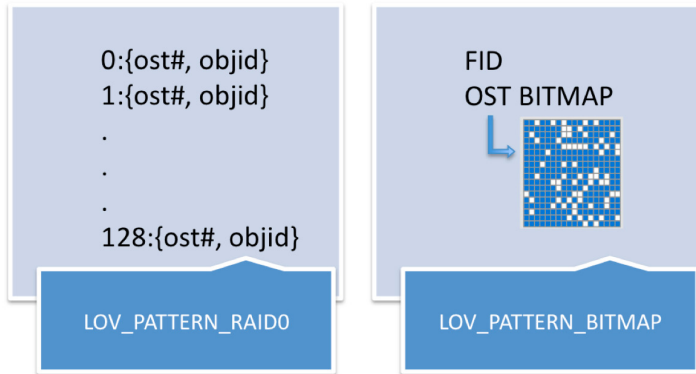


Figure 3: Bitmap layout type

Data Layout Policies. Lustre distributes file data to achieve even filling of OSTs and balanced I/O loading. This is a desirable policy in HPC environments, but if Lustre is used in an archival setting or in an implementation where locality of reference is important, then other data allocation policies may be more favorable.

In an archival setting, a policy that effectively fills a set of OSTs, one after another, can be useful. In this environment, it is likely that an OST, once full, will remain idle without additional activity.

Lustre already has a prototype implementation that joins two layouts. This can be leveraged to allow a file that cannot grow – because one of the targets on which it resides is full – to be extended by utilizing other OSTs.

As will be discussed later, it is also beneficial to introduce a layout for mirrored data that leverages the standard extended attribute mechanism for object location.

For cloud computing or general purpose “home directories”, where favorable locality of reference is needed, a layout policy that groups files by user ID is beneficial.

2. Metadata

2.1 Fast (Flash) Caches

Flash Storage. To improve Lustre’s metadata performance, Xyratex will utilize shared flash SSD storage in the drive trays, DRBD replicated PCI flash storage in the controllers, or RAM-disk devices in battery-backed RAM, replicated over a PCI inter-controller link. In each case, the storage device can be used for several purposes:

- Maintain ext4 recovery logs
- Store bitmaps and logs to speed up RAID recovery
- Allow fast access to other Lustre MDT files that see critical small I/O such as last_rcvd, last_objid, and delete operation logs
- Host a log, in conjunction with the size-on-MDS feature, that registers “file open” operations synchronously
- Provide a general purpose Linux Bcache or Flash Cache to aggregate small I/O

This use of flash storage relies on widely-tested Linux kernel software and involves minimal development in Lustre; the only significant changes are the locations of various files.

2.2 Client Concurrency

Single client node metadata performance is severely limited by a semaphore in the metadata client. This semaphore leads to a single, sequential stream of metadata updates between a client-mounted file system and the Lustre MDS.

To remove the semaphore requires changes to the recovery mechanism for RPC streams. For instance, each metadata update requires an entry in a reply cache, and presently there is one entry per client. This is similar to the NFSv4.1 design in which reply cache entries are called “slots”. With such changes, many threads on one client can make concurrent, metadata update requests.

This feature is very important in view of the high core count now commonly found in Lustre clients. With multiple update streams, single node metadata performance can be dramatically improved and should approximate the same metadata update rates as a small cluster.

2.3 Size-on-MDS

A significant improvement in directory listing is expected by replacing distributed storage of the file size (using each OSS involved in storing a file) with centralized storage of the size on the MDS (SOM). A new, simpler design for SOM should replace the 2002/2003 design. The old SOM model emphasized a design principle that no synchronous I/O should be undertaken. This specification caused several complications, such as logs on the OSTs to track writes, a difficult I/O epoch state machine, and the inability to use SOM altogether, when not even a single OSS is able to recover with the MDS.

With the utilization of flash storage, the MDS can now log operations synchronously in flash, as files are opened for writes. Having this database allows the MDS to query its own data and determine the files for which it can accurately provide a size, even if the OSS nodes have failed. Moreover, now there is just a single log on the MDT that tracks the opening of files. Figure 4 shows the simplified size-on-MDS RPC flow.

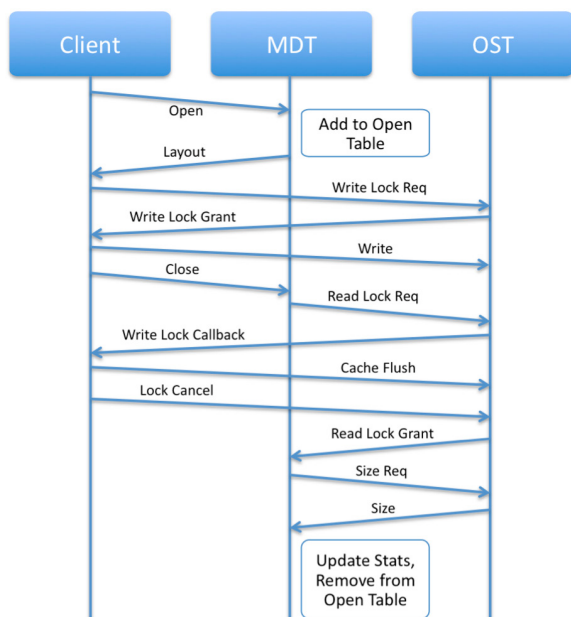


Figure 4: Simplified SOM based on flash storage

Using the new SOM model, in normal operation the MDS takes a lock on an entire file when it receives the last close for the file that was opened for write. As a result, the OSS nodes prompt the clients to flush their dirty pages for that file from the client caches. When the lock is granted, the MDS obtains the new file size from the OSS nodes and updates its metadata. The commit callback of the update cancels the entry in the open file database.

The SOM feature can be improved by making a small change to file size management. When a file is open, the MDS cannot use the file size in its metadata store. Instead of prompting the clients to obtain the file size from the OSS, the MDS should (at least optionally, but probably by default), request the file size directly from the OSS node itself. The benefit of managing file size requests in this manner is two-fold. Repeated acquisitions of a file size by clients do not require multiple calls to the OSS nodes and, in a wide area network, traffic over the WAN is dramatically reduced and replaced by MDS to OSS RPC messages. An additional benefit is that POSIX I/O HPC extensions (see, for example [POSIX I/O High Performance Computing Extensions³](#)) do not require a file size to be refreshed more frequently than necessary, as determined by an implementation. Implementing this enhancement to the SOM model as described is readily done by tracking the last time that the OSS nodes were asked for the file size.

If a client is doing I/O to the head of the file, it will already have knowledge of the file size and appropriate locks to guarantee its correctness. In this case, the client should not query the MDS for the file size.

In this model, recovery is very simple. The MDS iterates over entries in the table of open files which are not open after MDS recovery, and obtains file sizes from the OSS nodes, as described above. The MDS can provide file size information, except for files in its table, regardless of whether all OSS nodes have recovered.

2.4 MDS Threading Model

A fundamental problem with the Lustre MDS is that the metadata throughput of ext4 is almost an order of magnitude higher than that of the MDS. When this issue is addressed, significant improvements to metadata throughput become possible, without resorting to clustered metadata (even though that feature can readily coexist with this enhancement). Xyratex is implementing a threading model that better utilizes the backend file system, following guidance from the embedded database community.

Prototypes have shown that this threading model enables MDS performance close to ext4 performance on a local file system.

2.5 Maximum File Counts

In Lustre, each new file created in the file system consumes one inode on the MDS. To increase the maximum inode count, thereby increasing the maximum file count, few, if any, changes are needed to ext4 and Lustre because the 2.0 MDS no longer relies on Linux VFS code, nor on 32-bit inode numbers. However, various ext4 utilities may expect file systems to be formatted with fewer than 4B inodes. To complete changes removing ext4 and inode count limits, directory formats would have to accommodate 64-bit inode numbers.

³ J. Nunez, "POSIX I/O High Performance Computing Extensions", Carnegie Mellon University Parallel Data Lab, POSIX Extensions, LA-UR-05-8531, 2005.

While planning 4KB per Lustre inode on the MDT is recommended, in practice much less space is needed. A file system can be formatted with inodes of 256 bytes in size. These inodes are easily big enough to hold the striping of an inode over a small number of OSS nodes or a form of the new wide striping layout, as well as several other Lustre extended attributes such as the linkEA. The total space consumed by a Lustre inode is, approximately, the inode's 256 bytes, plus an entry in the FID lookup tables (about 16 bytes), and the name and inode number in the directory entry, adding up to approximately 300 bytes/inode. If a large storage volume is used for metadata storage, with, e.g. 20 pairs of 3TB drives to store MDS data, this allows storage of $60 * 10^{12} / 350 \approx 2 * 10^{11}$ inodes, i.e. 200 billion files. Using 512 byte inodes would lower the storage count to 100B files and allow more flexibility in the EAs stored in the inode.

To accommodate 1B files per directory, the current hash tree directory layout needs to be replaced with a Btree layout. We regard this as an optional, low-risk enhancement that would be accompanied by changes in the ext4 tools to check and repair such trees.

3. Availability

3.1 FSCK

A key challenge for large file systems is the ability to handle rapid consistency checking and repair. Since the initiative set by XFS, file systems have been hardened considerably. However, file systems still require an fsck utility to handle both dramatic failures and the consequences of software bugs. Time and time again, the absence of this utility has led to serious reputation problems for file systems.

The main problem with fsck is that the scanning time for large file systems becomes unacceptably long. While different file systems may have a more compact layout, the iteration over a similar number of inodes imposes a high lower bound for scanning. Initiatives like tilefs have introduced further parallelism, which greatly helps on disk arrays, but exploiting them on a single drive is generally not possible.

The traditional fsck architecture is roughly as follows. When the fsck utility scans the file system, it makes an alternative set of tables that contain critical metadata, i.e. information independent of the file system view through mounting. In important cases, the data in the alternative set of tables can be used to fix errors in the disk layout. A very clear semantic description of fsck is provided in [SQCK: A Declarative File System Checker](#)⁴.

Xyratex is working on a novel approach to overcome the limitations of file system checkers that is beyond the scope of this paper.

3.2 End-to-End Data Integrity

Since the introduction of ZFS and with the growth of peta-scale storage, much discussion has centered on the development of end-to-end data integrity mechanisms for file systems, see, e.g. [CERN's Data Corruption Research](#)⁵. This subject has seen many papers, messages, and blog posts offering strong opinions about the merits of a particular

4 H. S. Gunawi, et al., "SQCK: A Declarative File System Checker". In OSDI '08.

5 R. Harris, "CERN's Data Corruption Research", <http://storagemojo.com/2007/09/19/cerns-data-corruption-research/>.

approach. For example, contrast the conclusions in [About this Data Integrity Thing in Oracle ULK](#)⁶ with the summary in [End-to-End Data Integrity for File Systems: A ZFS Case Study](#)⁷. Yet there is no doubt that many different approaches can deliver very significant value.

In a network file system such as Lustre, data integrity checking should start by verifying that the copy of data from the client's application address space to the kernel address space does not corrupt the data. Ideally, the application has already computed a checksum and can request that the kernel verify this for the copied data. In the absence of a computed checksum, the kernel can compare source and destination memory. Note, however, that addressing all concerns about bad memory, instead of only data buffer issues, would require that most data structures, e.g. those containing inode and file structures in the kernel, contain integrity checks as well.

Lustre already has mechanisms (in the form of checksums) that verify data traveling over the network is not accidentally corrupted en route. (The checksums are not designed to intercept attempts at deliberate corruption, which would require a separate security mechanism.)

When data arrives on data servers, where it is stored in Lustre, the T10 mechanisms come into play. Lustre can leverage T10-DIF/DIX interfaces, which are now available in low-cost, high capacity drives and provide a low barrier to implementation. There are many choices involved in leveraging the T10 mechanisms, but great care is needed to cover most corruption cases, as explained in the paper, [Parity Lost and Parity Regained](#)⁸.

The T10 DIF/DIX interface, in conjunction with writes to a device with redundancy (e.g. a software RAID volume), can guarantee that the buffers and intended location of the buffers are identical in the drive firmware and in the OS. As explained in [Parity Lost and Parity Regained](#), by storing a data block version in the T10 APP field of data blocks in a stripe, and storing a checksum of these versions in the APP field of the parity blocks of the stripe, even lost writes can be detected.

If a T10 field is not consistent with the application's expectations (the application being the Lustre file server), then redundancy in the volume can be exploited to (attempt to) reconstruct the correct data. If all redundant copies of the data are corrupt, such as may occur because of a software bug, then the block device driver must pass up an error for the volume, which is subsequently handled by the file system or a file system checker. The paper referenced above demonstrates that most common file system corruptions are detected and repaired, although exotic ones (e.g. writing an entire stripe in the wrong place) are not successfully resolved. There is a cost to this approach – to detect lost writes, the parity block should be read whenever a data block from the stripe is accessed; this can be costly in terms of I/O. To detect lost writes, ZFS and BTRFS record a version of extents that are written to extent pointers in the index trees, possibly with less overhead than the T10 solution. For a discussion of the relative frequency of various corruptions, see [Parity Lost and Parity Regained](#).

6 J. Moellenkamp, "About this Data Integrity Thing in Oracle ULK", <http://www.c0t0d0s0.org/archives/6919-About-this-data-integrity-thing-in-Oracle-ULK.html>.

7 Y. Zhang, et al., "End-to-End Data Integrity for File Systems: A ZFS Case Study". In FAST '10.

8 A. Krioukov, et al., "Parity Lost and Parity Regained". In FAST '08.

3.3 Scalable File System Utilities

The ext4 utilities must be modified and verified to deal correctly with larger volumes; a considerable amount of this work is already done, but it requires careful validation. This is necessary to ensure that standard formatting, repair, and tuning utilities can handle the very large LUNs that are expected to become more common in the future.

4. Data Management

4.1 Replication Migration

Lustre 2.0 contains a changelog, which can be enabled on the MDS node to track changes to the file system. The changelog can, with care, be processed to collect metadata and file update changes for the purpose of fast replication, migration, and backup.

A key problem with such file system operations is addressed by the Xyratex daemon, which can perform these functions on live file systems, i.e. while changes are being made. Additionally, the Xyratex tool suite enables these features to be used even if the file system was created without a changelog. The impact of these operations on the design of the daemon is approximately as follows.

The daemon waits for new changelog entries and, after either a certain time has elapsed or a certain amount of change has been observed, opens a new epoch. The outcome of a backup, migration, or replication operation is that all namespace changes in the epoch are propagated. The namespace of the target, which is assumed to be replicated, migrated, or backed up to the beginning of the epoch, then has the state at the end of the epoch. Pathname calculations are done in an embedded database (in user space) to avoid excessive load on the MDS.

All files with data that changed during the epoch are transferred, backed up, or migrated. The consistency of this operation can follow one of two models. One model transfers data as is, regardless of whether the file is still undergoing changes. The second model transfers file data as it is encountered, and only completes data movement if the file data does not change further during the operation. If data is changed, then the transfer is retried in a later epoch. Figure 5 shows the changelog-based asynchronous replication model.

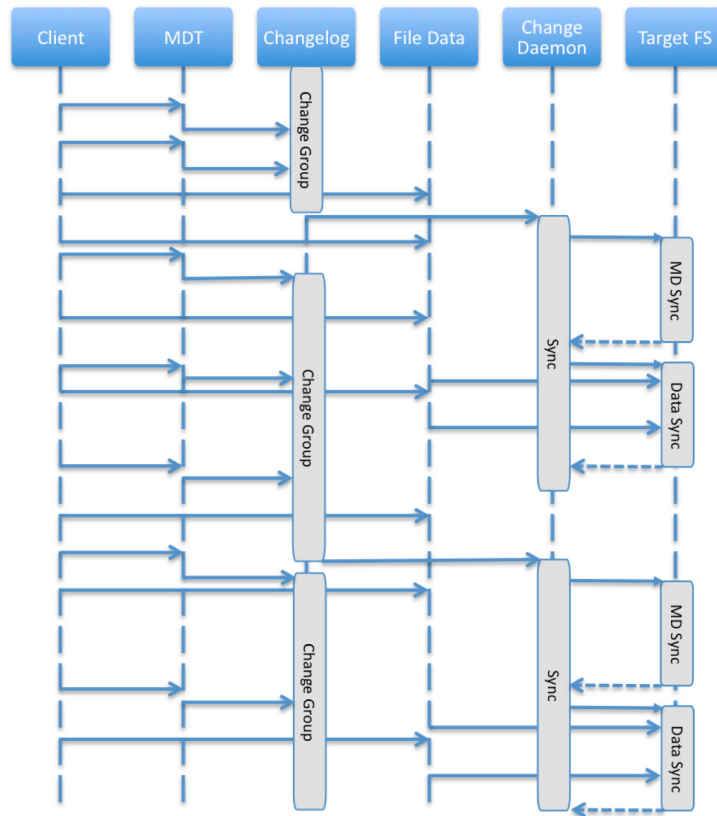


Figure 5: Changelog-based asynchronous replication model

The replicator handles a number of subtle features, such as the issue of files being renamed during or soon after they are modified. Moreover, the daemon moves data in parallel using a group of file system clients to propagate file data. This achieves a much higher bandwidth than offered by a single data mover. If only small parts of files are changed, then the data mover uses rsync-like algorithms to avoid rewriting entire files.

The replicator can replicate a file system that was created and populated without a changelog. For this, a scan is performed to record the contents of the populated file system. A changelog is created from the scanned contents, recording entries such as file creations with writes, and directory and link creations. Because the file system's data may change during the scan, great care must be taken to correctly treat the changelog entries created during the scan.

The replication infrastructure uses special APIs on the target file system to atomically record which changelog entry was the last replicated entry. This record allows the replication operation to be restarted without scanning or re-doing operations.

The APIs used by the replication, backup, and file system infrastructure are part of the Xyratex File Data Management Interface (FDMI). Because these APIs will be made available for multiple file systems, migration and replication between different types of file systems will be possible.

4.2 HSM

Xyratex plans to integrate the HSM infrastructure designed by Cluster File Systems in 2006, and implemented by CEA and Oracle.

The HSM interface also leverages the changelog, but in a different manner. The changelog is observed for certain conditions and when these are found, e.g. sufficient aging without use of the file, the policy implements a migration protocol to and from the backend HSM store. Cache misses are handled transparently.

This system has been documented elsewhere, see the [Lustre HSM Project](#)⁹, and will not be discussed in detail in this paper. Similar to replication, HSM involves coordinating and data moving nodes, as shown in Figure 6.

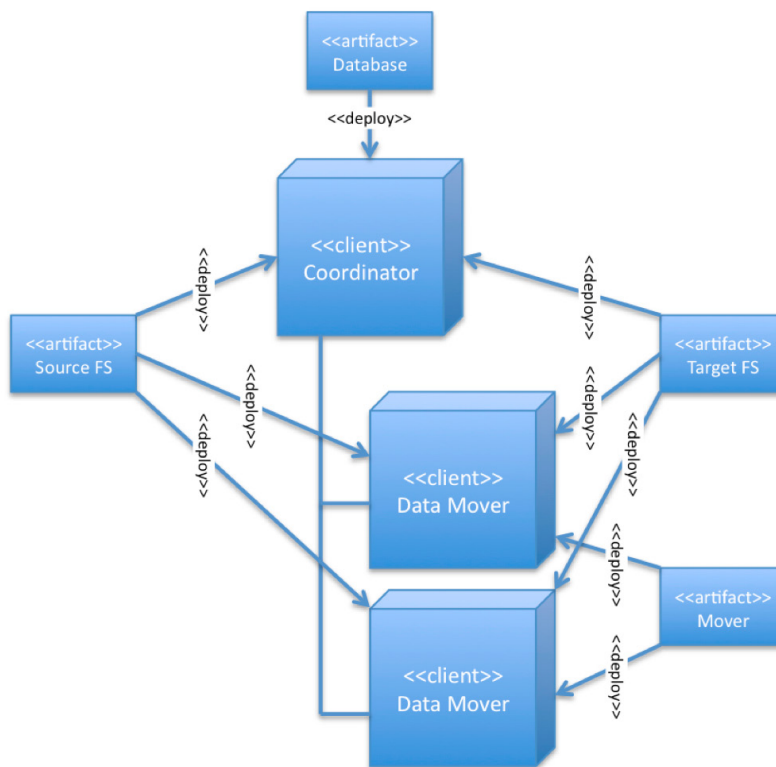


Figure 6: Deployment showing the coordinator and file data movers with source and target file systems

A shortcoming of the HSM system that will not be addressed in the near future is the difficulty of managing large collections of small files. One alternative is that the database maintained by the replication daemon (discussed in Section 4.1, above) may have sufficient information to collect groups of small files in small archives and migrate them to tape, while retaining the capability of transparent cache miss handling.

⁹ A. Degrémont and T. Leibovici, "Lustre HSM Project". In LUG 2010.

5. Comments on Alternative and Additional Approaches

5.1 Clustered Metadata

Clustered metadata allows for horizontal scaling of metadata throughput. It was first demonstrated in 2004, but full implementation has been significantly delayed. Xyratex considers the approach of significantly improving single-node MDS throughput, discussed earlier in this paper, to be a more favorable strategy that should be pursued first. There is no conflict between our approach and the clustered metadata feature, indeed they are complementary.

5.2 BTRFS

BTRFS and ZFS offer other approaches to data integrity and scalability of the file system than those proposed in this paper. Our approach demonstrates that changing the Lustre backend to such file systems is not a prerequisite to achieving significant scalability improvements and new features such as end-to-end data integrity. As file systems, device drivers, and firmware on adapters and drives will inevitably have bugs, a solid fsck repair tool remains a necessary utility.

Conclusion

We have presented an overview of Xyratex's Lustre architectural direction that we believe can significantly improve Lustre's scalability and performance, avoiding heavyweight implementation efforts upfront. We believe that these improvements will enable Lustre to reach common Lustre community goals within a relatively short timeframe, providing a valuable path forward to the next strata of performance.

About Xyratex

Xyratex is a leading provider of enterprise class data storage subsystems and hard disk drive capital equipment. The Networked Storage Solutions division designs and manufactures a range of advanced, scalable data storage solutions for the Original Equipment Manufacturer (OEM) community. As the largest capital equipment supplier to the industry, the Storage Infrastructure division enables disk drive manufacturers and their component suppliers to meet today's technology and productivity requirements. Xyratex has over 25 years of experience in research and development relating to disk drives, storage systems and manufacturing process technology.

Founded in 1994 in an MBO from IBM, and with headquarters in the UK, Xyratex has an established global base with R&D and operational facilities in Europe, the United States and South East Asia.



Xyratex Headquarters

Langstone Road
Havant
Hampshire PO9 1SA
United Kingdom
T +44 (0)23 9249 6000
F +44 (0)23 9249 2284

Principal US Office

46831 Lakeview Blvd.
Fremont, CA 94538
USA
T +1 510 687 5200
F +1 510 687 5399

www.xyratex.com



ISO 14001: 2004 Cert No. EMS91560

©2011 Xyratex (The trading name of Xyratex Technology Limited). Registered in England & Wales. Company no: 03134912. Registered Office: Langstone Road, Havant, Hampshire PO9 1SA, England. The information given in this brochure is for marketing purposes and is not intended to be a specification nor to provide the basis for a warranty. The products and their details are subject to change. For a detailed specification or if you need to meet a specific requirement please contact Xyratex.

No part of this document may be transmitted, copied, or used in any form, or by any means, for any purpose (other than for the purpose for which it was disclosed), without the written permission of Xyratex Technology Limited.

