PETA-SCALE I/O
WITH THE LUSTRE™ FILE SYSTEM

An Oak Ridge National Laboratory/
Lustre Center of Excellence Paper
February 2008

Abstract

This paper describes low-level infrastructure in the Lustre™ file system that addresses scalability in very large clusters. The features described deal with I/O and networking, lock management, recovery after failure, and other scalability-related issues.

Table of Contents

Chapter 1

Introduction

The Lustre™ file system first went into production in Spring 2003 on the Multiprogrammatic Capability Resource (MCR) cluster at Lawrence Livermore National Laboratory (LLNL). The MCR cluster was one of the largest clusters at that time with 1100 Linux compute nodes as Lustre clients. Since then, the Lustre file system has been deployed on larger systems, notably the Sandia Red Storm deployment, with approximately 25,000 liblustre clients, and the Oak Ridge National Laboratory (ORNL) Jaguar system, which is of similar scale and runs Lustre technology both with client-node Linux (CNL) and with Catamount. A number of other Lustre system deployments feature many thousands of clients. The servers used with these configurations vary considerably, with some clusters using fast heavyweight servers and others using very many lightweight servers.

The scale of these clusters poses serious challenges to any I/O system. This paper discusses discoveries related to cluster scaling made over the years. It describes implemented features, work currently in progress, and problems that remain unresolved. Topics covered include scalable I/O, locking policies and algorithms to cope with scale, implications for recovery, and other scalability issues.

Chapter 2

Networking and I/O Protocols in Very Large Clusters

This section describes features included or proposed for inclusion in the Lustre file system to deal with I/O and networking in very large clusters.

The quality attributes the Lustre file system aims to achieve are easily described. First, to achieve a high degree of efficiency in the pipeline between a client computer and server disk systems. Doing this has implications for how I/O requests are handled on the server and for the network protocol. Second, when the number of clients in a cluster increases, congestion must be avoided. And finally, the issue of fairness must be addressed. When many clients are executing I/O, it is desirable that all nodes make progress in sensible proportion to one another.

Utilizing client-to-server pipes

Disk system throughput is extraordinarily dependent on dispatching large requests over the storage network from the server. To efficiently utilize the pipe between the client and the server disks, the Lustre file system takes several measures.

Near the disk, the elevators in the device drivers merge requests to achieve efficient utilization of client-to-server pipes. One level up from the device drivers sits the server file system, which performs allocation aimed at putting large logical extents in files on disk as contiguous extents on the block device. The allocator must avoid concurrent allocations that interfere with this process while managing reservation windows in anticipation of further I/O. Small files need to be kept very close together to assure good I/O.

Figure 1 shows how the block allocator in the Lustre file system compares to a default block allocator in Linux ext3.

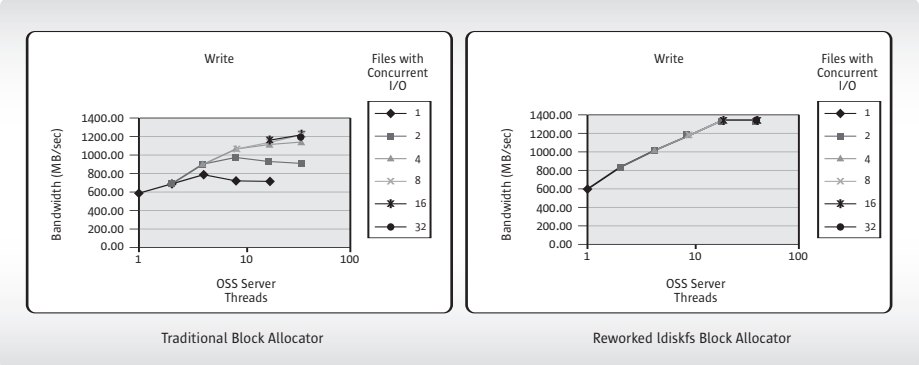

Figure 1. Concurrent I/O to multiple files

Figure 2 shows how 64 concurrent dbench threads perform with the ext4 file system.

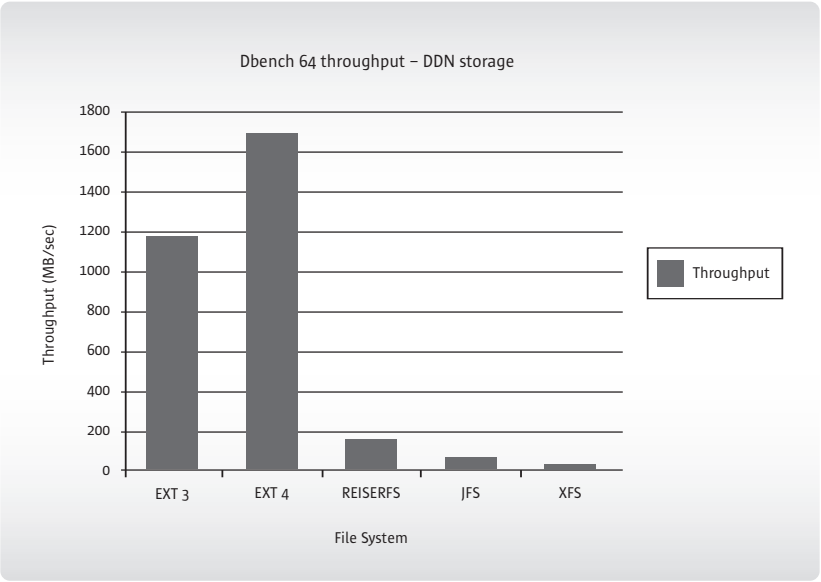

Figure 2. Dbench64 performance

All of the mechanisms in Figure 2 aim to send large I/O requests from the server to the disk systems while avoiding seeks and disk commits of small amounts of data. One might expect that network transports are also sensitive to small packets. While this is certainly the case, networks achieve very good throughput at much smaller packet sizes than disk systems. For example, Elan networks achieve near maximum bandwidth with 64K packets, whereas DataDirect Network S2A8500 Series Fibre Channel disk arrays require I/O in 1-MB chunks. Figure 1 shows that 512K chunks for read operations do not suffice when many files are written by many threads, even though they are adequate for write operations.

A second issue related to efficient use of the pipeline is keeping enough requests in flight. This is controlled by the client request dispatcher, which targets to keep up to eight packets in flight, by default.

Avoiding congestion

At the lowest level, Lustre Network Drivers (LNDs) in the Lustre networking (LNET) software stack support message passing through flow control credits. The credits are consumed and returned, assuring that packets do not overflow buffers. For example, when a packet is sent, the sender consumes a credit on the source node, allowing it to place the outgoing data in the sender's buffer. It also consumes a receiver credit, which provides a guarantee that the receiver has buffer space available when the packet arrives. When the credits are not available, the packet transmission has to wait. Credits are returned when the sender and receiver can free up the buffer space. To avoid overhead, credit information is piggybacked on all packets.

Figure 3 illustrates how this mechanism is implemented.

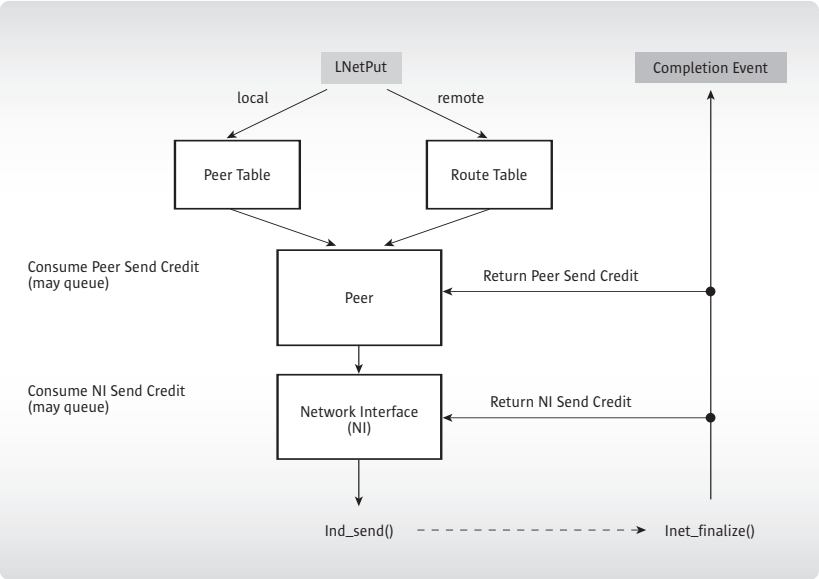

Figure 3. Sending packets with the LNET API

The file I/O protocol in Lustre technology separates bulk transfer from the initial I/O request. This means that bulk data is moving over the network at the discretion of the server, which significantly limits the amount of congestion.

Figure 4 shows how the server initiates remote direct memory access (RDMA) requests.

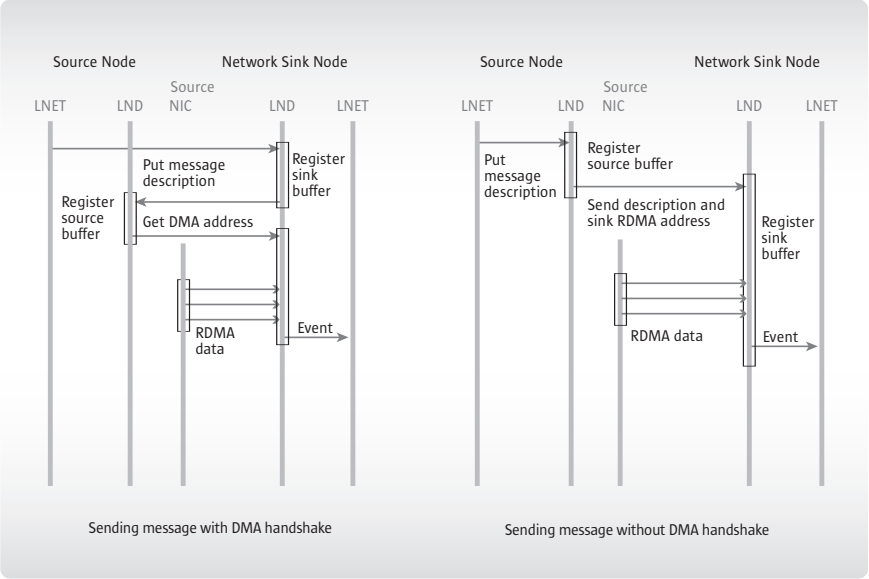

Figure 4. Server-driven RDMA for file I/O

The default limitation of up to eight packets in flight described in the *Utilizing client-to-server pipes* section minimizes the number of smaller packets in the pipeline. Metadata requests are similarly limited. This further reduces congestion.

Open issues

The I/O throughput of the Lustre file system is excellent, but further improvements are possible.

First, disk arrays with SATA disks are much more sensitive than other types of disk arrays to correct ordering of I/O requests to avoid seeks. There are two changes that could address this concern. One simple change is the addition of a cache on the object storage server (OSS) to collect small writes before dispatch and retain file data to help with rereading. A second change would enable the servers to use a local request scheduler (LRS), which would group the I/O requests they receive from different clients in a suitable order for dispatch to the file system. Currently, requests are dispatched with a First In, First Out (FIFO) policy that fails to address important I/O optimization issues.

Second, large clusters sometimes exhibit uneven I/O behavior when many clients are involved. This behavior can be very visible when a single shared file is written by many clients to many I/O servers because, in this instance, uneven distribution of client throughput will slow the entire process down. However, it appears that for the Lustre file system, uneven I/O behavior is simply related to uneven throughput of requests at any point between clients and disk storage. Examples of possible causes of uneven throughput are networks that favor some clients over others, servers that are too heavily loaded to respond to requests, and disk file systems that slow down I/O due to fragmentation.

To eliminate uneven I/O behavior, the LRS must enforce a collective flow control policy, which assures that, over time, some clients do not receive unfavorable treatment as a result of these imbalances. Such resource management issues can become highly complex and may, for example, involve redirection of I/O requests to different, less-loaded servers with associated changes in the file layout. However, at present, all distributed file systems do little or nothing at all, and a basic policy to try to provide even throughput from clients will probably make a major difference. This issue is also closely related to server-driven quality of service (QoS) as described in the *Lustre Networking* white paper, Chapter 4: Anticipated Features in Future Release. Figure 5 shows how a server policy can influence an LRS to enforce a policy.

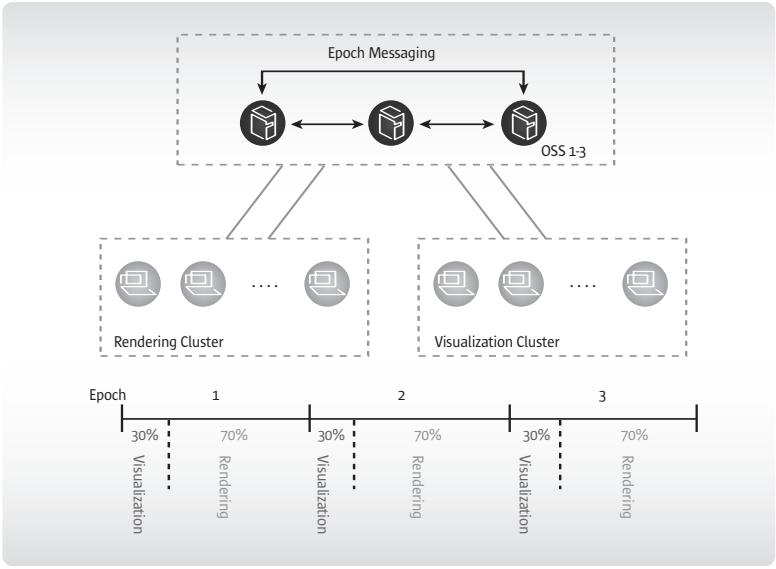

Figure 5. Interaction of the LRS and server epoch handler to enforce QoS policy

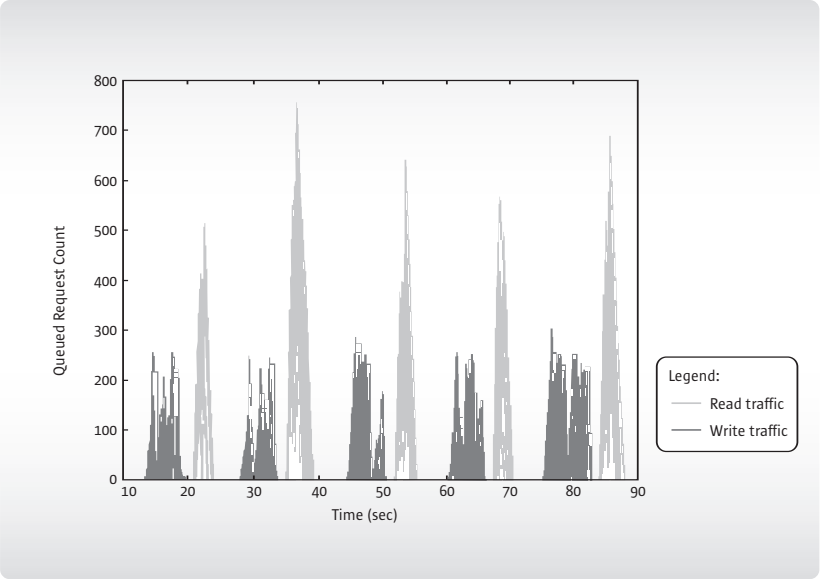

Figure 6. Queued I/O requests on servers with a suboptimal load versus time Queued Request Count

These concerns are further motivated by the results from detailed measurements made on the very large Jaguar cluster at ORNL. Read and write requests were observed on 72 servers while they handled a load generated by 8000 clients. Figure 6 shows the results of testing with this suboptimal load over five write and read phases.

Several issues can be inferred from these graphs. First, the clients are not able to issue an adequate number of write requests, likely due to insufficient parallelism when writing to many servers.

Secondly, during reading and writing, the servers handle approximately 8 to 12 GB/sec in aggregate throughput while they should be able to deliver approximately 25 GB/sec. A server-side request scheduler could arrange for more sequential I/O to increase the throughput.

Finally, a detailed study of the gaps between the read and write phases of the test reveals that during writing the opening and closing of files is taking too much time.

Chapter 3

Locking Strategies at Scale

The Lustre file system protects the integrity of cached data through the use of distributed locks. Such lock management goes back at least to the VAX cluster file system and possibly even further. When distributed locks are used in a very large cluster with extremely large files, several interesting new issues appear. The locking model has many subtle variations, but for the purpose of this discussion, it may be assumed that resources are protected with single-writer, multiple-reader locks.

Lock matching

When locks are required by clients, they are checked against already granted locks for conflicts. If conflicting locks exist, they have to be revoked through a callback to the lock holder. If no conflicting locks exist, the lock can be granted immediately. It is very common for a very large amount of compatible read locks to be granted simultaneously, for example, when a client wants to perform a lookup in a directory or when many clients read from one file or write to disjoint extents in one file.

To optimize the checks for lock compatibility, the linear lists that traditional lock managers use were replaced in the Lustre file system by skip lists, which make fast compatibility checks. Skip lists, first introduced by William Pugh, are lists with multiple pointers that aid in searching, as shown in Figure 7.

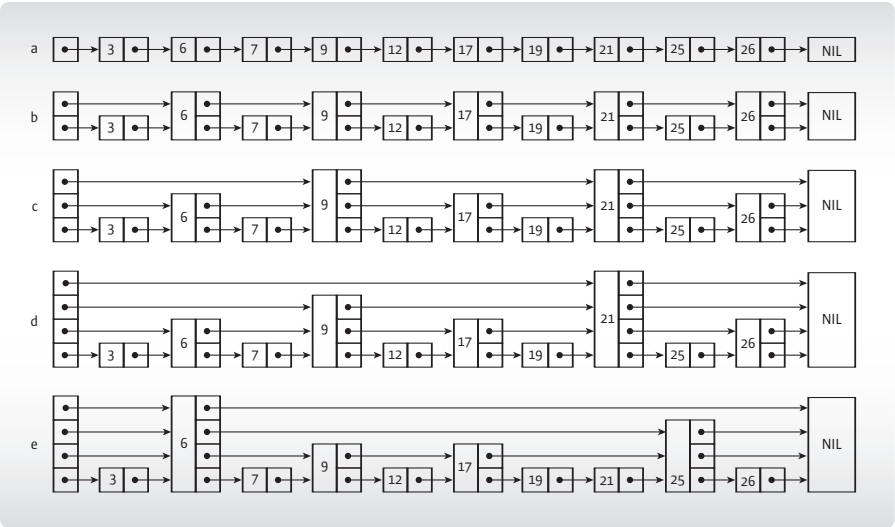

Figure 7. Skip lists showing linked lists with additional pointers

When file extents are involved, the lists of extents are replaced by an interval tree, making lock matching dramatically faster. Figure 8 (top) shows a set of 10 intervals sorted bottom to top by left endpoint. Figure 8 (bottom) shows an equivalent interval tree.

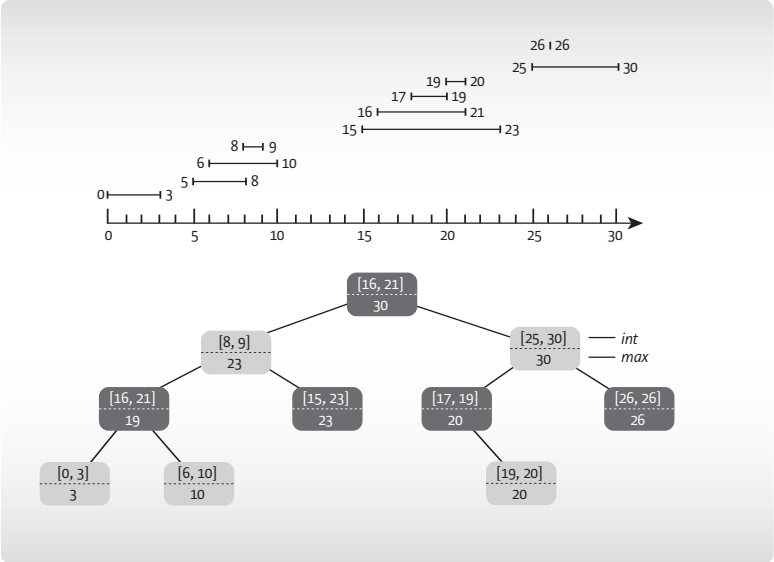

Figure 8. An interval tree representing a sorted list of intervals

Lock contention

Intent Locks

For metadata processes, such as the creation of files, situations where heavy sharing of data occurs are common. For example, when all clients create files in one directory, all the clients will need information about that directory. While traditional file systems granted each client in turn a lock to modify the directory, the Lustre file system has introduced "intent locks."

Intent locks contain a traditional request together with information about the operation that is to be performed. At the discretion of the server, the operation is fully executed without granting a lock. This allows Lustre technology to create files in a shared directory with just a single remote procedure call (RPC) per client, a feature that has proven robustly scalable to 128,000 clients on the BlueGene/L system at LLNL. Figure 9 illustrates this behavior.

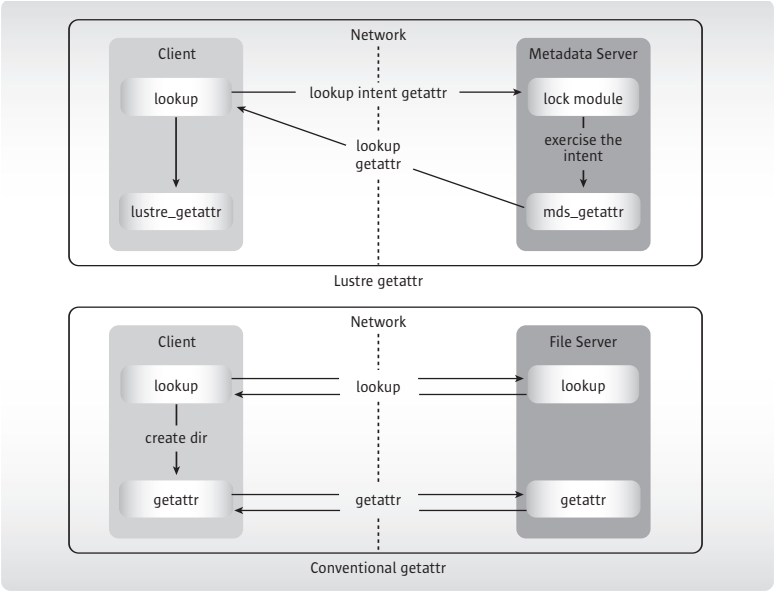

Figure 9. Single RPC metadata operations with intent locks

Adaptive I/O Locks

Although some applications involve heavy I/O sharing, these scenarios are rare. The Lustre file system adapts when heavy I/O sharing occurs, but normally assumes that clients should be granted locks and that these locks will not be frequently revoked. When the Lustre file system detects, in a short time window, that more than a certain number of clients have obtained conflicting locks on a resource, it will deny lock requests from the clients and force them to write through. This feature has proven invaluable for a number of applications encountered on very large systems.

For the Catamount version of the Lustre file system, called liblustre, adaptive I/O locks are mandatory for another reason. The Catamount nodes cannot be interrupted, and, as a result, lock revocations are not processed. To avoid corrupting cached dirty data, Catamount does not take write locks. Read locks can be revalidated upon reuse, but write locks are associated with dirty data on the client and cannot be taken.

Chapter 4

Recovery at Scale

Like most network file systems, the Lustre file system uses an RPC model with timeouts. On large clusters, the number of concurrent timeouts will be significant because uneven handling of requests can lead to significant delays in request processing. Timeouts may result when clients make requests to servers or when servers make requests to clients. When clients make requests to servers, a long timeout is acceptable. However, when servers make callback requests for locks to clients, the servers expect a response in a much shorter time to retain responsiveness of the cluster. This section discusses some of the interactions between timeouts and scale that the Lustre file system has had to overcome.

Serialized timeouts

Relatively early in the development of the Lustre file system, commonly occurring scenarios where timeouts happen repeatedly were discovered. Such a scenario may occur when compute nodes in a group each create files in a directory, execute I/Os to the files, and then, subsequently, the compute nodes are unreachable due to a circumstance such as a network failure. If another node in the cluster (call it the “listing” node) performs a directory listing with the “ls -l” command, problems can appear. The listing node will try to get attributes for each of the files sequentially as it makes its way through the directory entries it is listing. For each request for file attributes, the server will do a lock callback to the client, and because the client is unreachable, the server callback will time out. Even though callback timeouts take just a few seconds, with a few thousand clients, this leads to an unacceptable situation.

A ping message was introduced in Lustre technology to address this problem. A ping request is a periodic message from a client showing it is alive and a reply demonstrating that the receiver is alive. If the ping message is not received by a server, the server cancels the client's locks and closes its file handles, a process known as eviction. The affected client only learns that this has occurred when it next communicates with the server. It must then discard its cached data. Thus, a ping message is similar in function to a leased lock (a lock with a timeout), except that a single ping message renews all Lustre locks, while a separate lease renewal message would have to be sent for each leased lock.

Such ping messages are not only important to avoid serialized timeouts, they are also vital when three-way communication is involved. For example, a client may be communicating with two servers and server 1 grants a capability, such as a lock, to the client to enable the client to communicate with server 2. If, due to a network failure, server 1 is not able to communicate with the client holding the capability, it is important that such a capability loses its validity so that server 1 can grant a conflicting capability to another client. Clients and servers can agree that when a ping message is not received, or no reply is received in response to a ping message, the capability has lost its validity. After the expiration of the ping message window, it is safe for servers to hand out a conflicting capability.

Of course, a ping message also allows a client to detect that a server has failed when no response to the ping message is received from the server. However, ping messages in large clusters can cause considerable traffic. To resolve this, the intent is to adapt the Lustre file system to use ping services that gather status from all servers so that a single ping message from a client can convey all required information. Using this in conjunction with a tree model for communication, for example, by using all server nodes to handle pings, can lead to very scalable pinging.

Interactions among multiple locks

The Lustre striping model stores file stripes on object server targets (OSTs) and performs single-writer, multiple-reader locking on file extents by locking extents in each object covered by the file extent. When I/O covers a region that spans multiple OSS systems, locks are taken on each of the affected objects. The Lustre lock model is one that acquires a lock and then references it on the client while the client uses it. After use, the reference is dropped but the lock can remain cached. Referenced locks cannot be revoked until the reference is dropped.

The initial implementation for file data locking took locks on all objects associated with a given logical extent in a file and referenced these locks as they were acquired. In normal scenarios, the locks were acquired quickly in a specific order and each lock was referenced immediately after it was acquired.

A problem arises with this implementation in situations in which both of the following occur:

- The client receives a callback for a lock that has just been acquired.
- The client experiences a timeout while acquiring a further lock associated with the file extent.

The problem is that the first lock is not released in time because the timeout for the second operation is longer. As a result, the client that is experiencing problems with the server that has given out lock (2) is now evicted by the server associated with lock (1). This leads to cascading evictions, which are undesirable.

The Lustre file system was modified to avoid this. The locks on extents in objects are referenced, while I/O to the object is taking place, but do not remain referenced when I/O takes place to an extent in another object. The locking of objects is effectively decoupled. This greatly reduces the number of timeouts experienced on larger clusters, where problems with lock enqueues are relatively common.

A further change to the Lustre file system is planned that will introduce more parallelism to the I/O process to stripe objects. Figure 10 shows two interaction diagrams with the current implementation on the left and the planned implementation on the right.

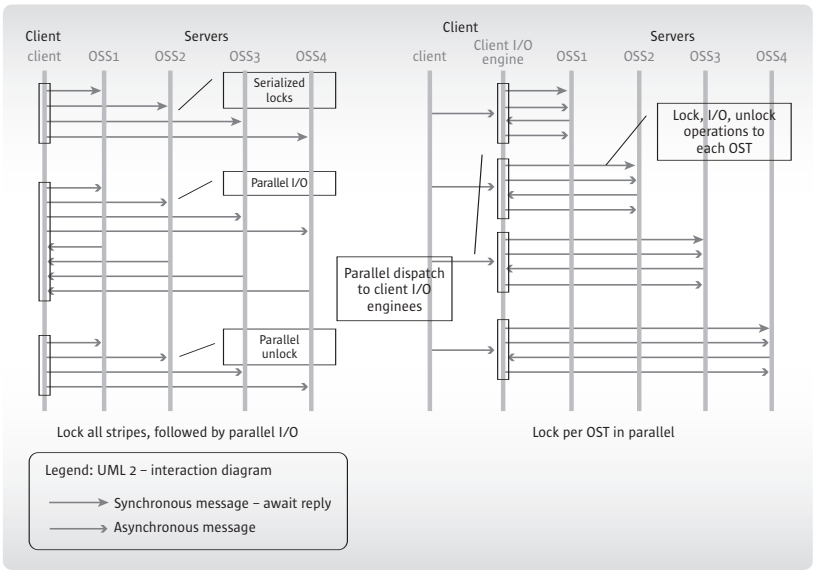

Figure 10. Object locking implementation in Lustre

On a single client, this approach does not affect POSIX semantics because the Linux kernel enforces POSIX semantics with different client-only locks in the virtual file system (VFS). However, in a cluster, one must be careful with operations that have strict semantics, such as truncates and appending writes. For such operations, the previous Lustre locking behavior is retained.

This methodology also illustrates another scalability principle that was introduced for metadata, namely parallel operations. Whenever possible, the clients will engage all OSS nodes in parallel, instead of serializing interactions with the OSS. This is illustrated in Figure 11.

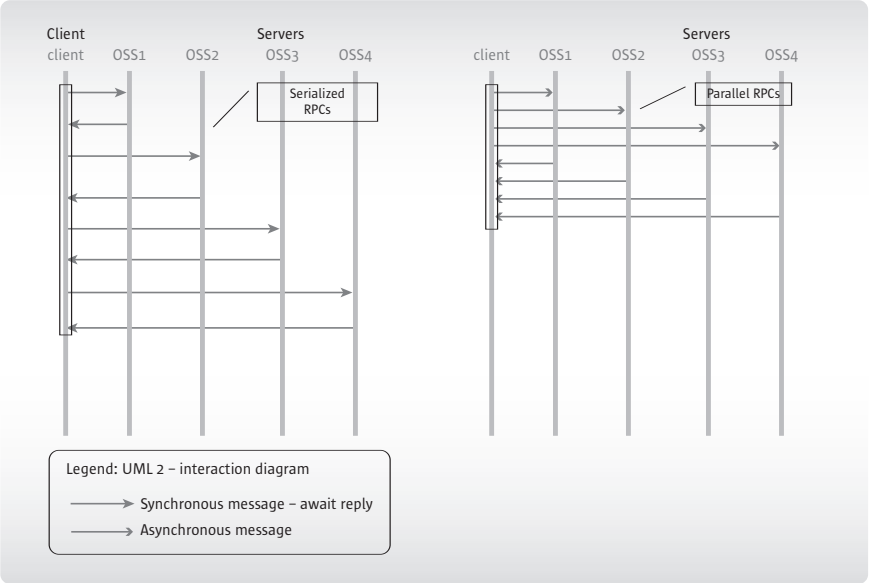

Figure 11. Parallel operations between clients and servers

Future directions — version recovery

Lustre file system recovery provides transparent completion of system calls in progress when servers fail over or reboot. This recovery model is ideal from an application perspective, but requires all clients to join the cluster immediately after the servers become available again. When a cluster is very large, or contains Catamount clients that may not have noticed a server failure and subsequent restart, some clients may not rejoin the cluster immediately after the recovery event. Consequently, the recovery will abort, leading to evictions.

Version recovery is planned to be introduced into the Lustre file system to more gracefully handle the situation where a client reconnects but misses the initial recover window. If the client reconnects later and finds that no version changes have taken place since an object was last accessed, the client will retry the operation that was in progress and did not complete. After this retry, the client rejoins the cluster. Cached data will only be lost if an object was changed while the client was not connected to the cluster, a decision again based on the version.

This model scales well and the compromises it includes for fully transparent recovery are acceptable. However, it has limitations. For example, versions are determined per object, not per extent in the object. Hence, version recovery for I/O to a shared file may continue to result in eviction of cached data.

Version recovery is of particular importance because it enables a client to reintegrate changes long after the initial recovery, for example, after a disconnection.

Future directions — adaptive timeouts

Client requests see huge variations in processing time due to congestion and server load. Hence, adaptive timeout behavior is very attractive and is planned for the Lustre file system 1.6.5 release.

When a server knows that it cannot meet an expected response time for a client request, it will send an early response to the client indicating a best guess for the required processing time. If the client does not receive such a message, the client will assume a failure of the server and, after the server recovers, resend the request.

With adaptive timeouts, failover time on lightly loaded clusters will drop from minutes to seconds. Adaptive timeouts are expected to be highly effective when server loading is involved, but it is harder to adapt to all cases of network congestion.

Chapter 5

Other Scalability Features

Additional features have been implemented in the Lustre file system to address issues related to scalability.

Lists, hashes, and other small scalability issues

A variety of small scalability issues have been addressed. In a few cases, lists have been replaced by hashes, for example, for connections. In other cases, results from searches, such as a search for a quota master system, are now cached to avoid repeating the search when many clients connect. A more interesting issue involved synchronous I/O, which the Lustre file system used only during the connect phase. Synchronous I/O was eliminated because it caused timeouts at a scale of 25,000 clients.

The fsck system utility and Solaris™ ZFS

A file system consistency check may be required when multiple disks fail or when RAID controllers violate write-ordering requirements, for example, when a power outage causes a cache to be lost. In these cases, the system utility fsck must be run, which is very time consuming. The Lustre file system includes dramatic changes in the efficiency of fsck by limiting the scan for inodes that are allocated, as illustrated in Figure 12. The file system also features scalable formatting by dynamically formatting more block groups as more inodes are used.

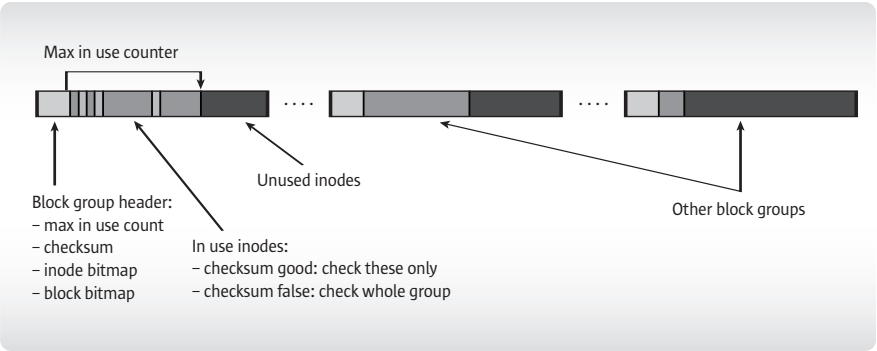

Figure 12. Fast fsck and dynamic formatting

Beginning with the Lustre file system version 2.0, the Solaris™ ZFS file system data management unit (DMU) will be used, replacing Linux disk file systems. The ZFS DMU is scalable and requires no file system consistency check.

MPI-I/O

Message Passing Interface I/O (MPI-I/O) can play an important role in alleviating scalability issues, as demonstrated by research done at ORNL and Argonne National Laboratories. MPI-I/O provides the Lustre file system with a more sophisticated analog-to-digital I/O driver with stronger support for collective operations.

Scaling on SMP nodes

The Lustre file system has been optimized for use with Symmetric Multiprocessing (SMP) and Non-uniform Memory Access (NUMA) systems. Several small changes have significantly improved the efficiency of the Lustre file system when used on nodes with many CPU cores. On clients, frequently used data types called Lustre handles are placed in read-copy update (RCU) lists, which greatly reduces contention. Moreover, Lustre caches use per-CPU pages that are lazily updated. For these clients, I/O does not degrade with high core counts on client nodes.

On metadata server (MDS) nodes with many CPUs, the distributed lock manager (DLM) locks are now per resource, not per namespace. This change alone has allowed an up to 5x speedup of metadata operations on servers with 8 to 16 cores.

Similar fine-tuning is under way to improve software RAID performance.

Chapter 6

Conclusion

Further challenges lie ahead as systems scale from petaflops to exaflops over the coming decade or two. Enabling the agility to adapt when scalability issues arise is one of the greatest strengths the Lustre file system can bring to high-performance computing (HPC) storage.

Sun Microsystems, Inc. 4150 Network Circle, Santa Clara, CA 95054 USA **Phone** 1-650-960-1300 or 1-800-555-9SUN (9786) **Web** sun.com

SunWIN #526918 Lit. #SYWP14018-0 02/08