Lustre: Building a File System for 1,000-node Clusters

Philip Schwan
Cluster File Systems, Inc.
phil@clusterfs.com, http://www.clusterfs.com/

Abstract

Lustre is a GPLed cluster file system for Linux that is currently being tested on three of the world's largest Linux supercomputers, each with more than 1,000 nodes. In the past 18 months we've tried many tactics to scale to these limits, and the first half of this paper will discuss some of our successes and failures. The second half will explore some of the changes that we plan to make over the next year, as we scale towards tens of thousands of clients and petabytes of data.

1 Introduction

The Lustre cluster file system has been designed and implemented with the goal of removing the bottlenecks traditionally found in such systems. Lustre runs on commodity hardware and provides a cluster storage layout that is efficient, scalable, and redundant. Metadata Servers (MDSs) contain the file system's directory layout, permissions, and extended attributes for each object. Object Storage Targets (OSTs) are responsible for the storage and transfer of actual file data, and already scale to many dozens of OSTs and hundreds of terabytes of data. Both types of service node can operate in pairs which automatically take over for each other in the event of failure. Each also runs an instance of the Lustre distributed lock manager, access to which forms the core of the Lustre protocols.

Although Lustre's design dates from 1999, development began in earnest in early 2002. In the time since, surprisingly few of the major points have changed from the original plan, and the implementation has undergone fewer false starts as a result. Our distributed lock manager has weathered the storm and remains largely as it was a year ago. The choice of a system designed around object protocols has proven to be correct, and Lustre has so far scaled to the limits of available hardware. Lustre's internal networking has shown itself to be relatively flexible and high-performance, and network abstraction layers exist for TCP/IP, Quadrics Elan, Myrinet, and SCI.

Not all of our original decisions were ideal, however. One source of bugs continues to be the interaction between Lustre and the Linux VFS layer, which is not very well suited to network file systems that want a great deal of control. This interaction had a significant impact on one of Lustre's major metadata architecture choices, the concept of "intent-based" locking operations, described in more detail later. Ultimately, we had to make significant changes to our intent-based metadata implementation.

The last year of working with government and industry has suggested which activities are most important to pursue in the next year. First, and most significantly, two major caching im-

provements will be made beginning this summer: a write-back metadata cache, and a persistent data/metadata cache. The write-back cache will be enabled in times of low concurrency, and allows for metadata updates which can be made in local memory and later replayed on the server. Making this cache persistent for both metadata and file data will enable features such as disconnected operation and server replication. Finally, our collaborative read cache will reduce the load on primary OSTs for the most frequently accessed files, removing a very common bottleneck in distributed systems.

2 Distributed Lock Manager

All of Lustre's consistency guarantees are enforced, in one way or another, by the Lustre distributed lock manager (DLM). Core operational decisions, such as when to switch between writeback caching and synchronous metadata updates, will be delegated to the DLM.

The design of the Lustre DLM borrows heavily from the VAX Clusters DLM, plus extensions that are not found in others. Although we have received some reasonable criticism for not using an existing package (such as IBM's DLM[1]), experience thus far has seemed to indicate that we've made the correct choice: it's smaller, simpler and, at least for our needs, more extensible.

The Lustre DLM, at just over 4,000 lines of code, has proven to be an overseeable maintenance task, despite its somewhat daunting complexity. The IBM DLM, by comparison, is nearly the size of all of Lustre combined. This is not necessarily a criticism of the IBM DLM, however; to its credit, it is a complete DLM which implements many features which we do not require in Lustre.

In particular, Lustre's DLM is not really *distributed*, at least not when compared to other such systems. Locks in the Lustre DLM are always managed by the service node, and do not change masters as other systems allow. Omitting features of this type has allowed us to rapidly develop and stabilize the core functionality required by the file system.

Next, we feel that our extensions to the basic DLM API and protocol have been quite successful. File range locking is managed internally as part of the regular lock matching and compatiblity functions. Through the use of a small policy function, the lock manager is able to grant larger locks than originally requested. In this way we avoid the bottleneck found in some other file systems, for which a client must lock each page or block individually. For the very common case of a file being accessed by only one user, Lustre's DLM will grant exactly one lock for the entire file.

Finally, intent locking is designed around the concept of allowing the lock manager to choose between different modes depending on its view of resource contention. In a directory with very little contention—a user's home directory, for example—the DLM can grant a *write-back* lock, allowing the client to cache a large number of metadata updates in memory. In this way it will avoid an interaction with the server for each request and batch them at some later time. In a directory with very high concurrency—such as `/tmp`—the DLM will refuse to grant any lock at all. Instead, it will perform the operation on the client's behalf, notify it of the result, and avoid bouncing the directory lock between hundreds or thousands of simultaneous users.

3 Object Protocols

Of all of the concepts that went into Lustre's architecture, the use of *object protocols* is by far the most pervasive. Although Lustre is certainly not unique in its use of storage objects, we have also designed many of the internal APIs to allow for additional layering (RAID 0 as one example) or short-circuiting (a client running on an OST with no networking layer between them). This symmetry between internal APIs and network protocols has served us well.

During the initial design it became quite clear that a shared block file system would absolutely not scale to the required limits for many reasons. First, shared disk arrays on anything but the smallest clusters quickly become cost ineffective for even the largest customers; this would certainly violate our goal of running on inexpensive commidity hardware. Second, high-level object protocols remove a key bottleneck for scaling beyond a dozen or two nodes: locking and allocation of metadata.

In a traditional shared-block file system, those blocks which store inode and block allocation information are subject to incredible contention. By organizing the protocol around objects instead of blocks, the OSTs remain responsible for the internal metadata allocation. Parallel file I/O to a single file has been shown to scale to more than 1,100 nodes, the limit of available hardware.

For those customers who have already invested in a large storage area network (SAN) based around shared disk, Lustre is still an option. In the SAN mode, OSTs are still responsible for managing the object locking and shared storage metadata, but clients can read and write individual data pages directly from the SAN.

4 Networking

Lustre's networking layer has not changed significantly from its original form more than a year ago. It uses a simple message-passing package called Portals[2], which has from an API standpoint served us fairly well. Importantly, it provides the right abstractions for enhancements such as remote DMA as supported by the networking hardware. We've made a few relatively minor API changes to accomodate the different needs of the filesystem, as opposed to the scientific community from which Portals emerged.

The original implementation of Portals, however, caused many serious problems. The port to run in the Linux kernel and userspace was fairly straightforward, but Portals had never been run in a multi-threaded environment and had absolutely no internal locking. Given that we had to rewrite more than 80% of the code and put up with serious race conditions for many months, it would likely have been a better choice to keep the API and start the implementation from scratch.

The API and the internal abstraction layers, however, have been both simple enough to understand and modify, and flexible enough to cope with the needs of many networking drivers. Lustre (and therefore Portals) needs to support a variety of interconnects, including kernel TCP/IP, TCP/IP offload cards, Quadrics Elan 3, Myrinet, and SCI.

For each network type we have a Portals Network Abstraction Layer (NAL), approximately one to two thousand lines of code each. Although they are small, they are generally quite complicated, and may depend on a fair bit of wizardry to get the most out of a particular interconnect. Nevertheless, the Lustre network regression test running on our Elan NAL between two nodes is bottlenecked by the PCI bus

at more than 300 MB/s.

5 Intent Operations Explained

Most distributed file systems perform metadata operations in the same way all the time, regardless of contention. Some systems choose to give locks on objects to clients, and some choose to perform all operations synchronously on the server.

In the first mode, a client wanting to perform metadata operations will first take a lock on the parent directory, download the applicable directory information, and make many changes locally. This is extremely efficient in times of low contention; it can perform as many operations as it wants locally without contacting the server, as long as no other nodes try to acquire the lock. In times of high contention, however, it is a disaster: imagine users on 1,000 nodes all running `touch /tmp/foo`. The lock on `/tmp` will have to be given to each node in turn, and the cluster will grind to a halt.

In the second mode, the client sends a message to the server for each operation, and the server performs the operation without giving any locks out. Not only is this much simpler to code properly, it also avoids the problem with lock ping-pong. When this mode is used, however, even directories with no contention have this behaviour, and you suffer the effects of a server round-trip for each operation.

Lustre currently executes all operations as if there were high concurrency, with exactly one RPC per metadata operation. With the completion of the writeback metadata cache later this year, the DLM will be able to make the choice between giving the client a writeback lock on a subtree or performing one RPC per op.

6 Intents Gone Wrong

Our first attempt at writing the client-side VFS code to support the intent mechanism was roughly as follows. Consider the case of a mkdir operation: normal filesystems will lock the parent directory, lookup the new directory to see if it already exists, create it, then release the lock. Lustre added a *lookup intent* structure to each lookup call, to tell the lock manager on the server why we asked for the lock (in this case, to mkdir). If the server decided not to give out the lock, it would perform the operation on the client's behalf and return a success or error code.

When the Lustre client received this reply, it would do complicated things to cooperate with the VFS. If the mkdir succeeded, for example, it needed to create a *negative* directory entry (dentry) before returning from lookup (if we returned a new positive dentry, the VFS would return `-EEXISTS`). Later, the VFS would call us back to do the "actual" mkdir, at which time we would instantiate the dentry based on the reply stored in the lookup intent.

This turned out to be a disaster of race conditions, both on the server and on the client. On the server, the lock manager would perform these operations before the lock was granted, so that it could give the client a lock on the new file. By the time the lock was actually granted, however, anything could have changed. On the client, our ability to control the dcache, particularly in the window between lookup and final creation, proved insufficient.

Our final solution was to make two fairly major changes to both sides. Instead of the lock manager performing an operation before locks are granted, the metadata server is able to specify an already-granted lock to give to the client. This allows the MDS to perform the operation and then return the still-granted lock on

the new file without races. The client has been simplified to call directly into the filesystem and return the result immediately: no dcache, no VFS code, no races.

7 Client Metadata Caching

When a node asks to lock a directory for reading or writing, the Lustre DLM will soon be able to grant a *subtree lock*, if the directory has not recently seen conflicting activity. This allows the client to keep a cache which can be filled and selectively revalidated as necessary.

As a client with a subtree lock fills its cache from the MDS, the MDS may revoke locks on other objects. If during this process the MDS encounters an opened file or a file with hard links, it flags this file for special attention by the client. Specifically, the client also flags these as shared objects which cannot be cached locally and must use the intent path for updates.

Once a client has a subtree lock, it can begin to keep a local journal of updates. Each update is a short record which describes one logical filesystem operation on an object, for example "create directory, mode 0755, parent inode 12, new directory name foo". Because all update operations are now reduced to the creation of a single record in client memory, they are incredibly fast.

When another client attempts to perform a conflicting operation beneath a subtree lock, that lock must be found and revoked. The MDS server code can easily walk the dentry tree, looking at each path component of the affected object, and revoke subtree locks as necessary. It is now easy to see why we flag hard linked files for special handling, as they have more than one path by which they can be reached.

When a subtree lock is revoked, any accumulated updates must be flushed to the MDS and replayed on stable storage. This is exactly the same mechanism which already exists for intent locks, except that they are grouped into pages of operations which are all guaranteed to succeed by virtue of the subtree lock. Now a single network exchange can contain hundreds of records.

8 Persistent Caching

To this design there is one particularly attractive extension, which is a persistent cache in the style of AFS[3], Coda[4], and InterMezzo[5]. Two new pieces are required for such an extension: the local cache itself, and a way to revalidate its contents after locks are lost and re-acquired.

Lustre's stackable object protocols allow a very symmetric design for the persistent cache by adding a metadata server to the client. Today the file system code interacts with a metadata client (MDC) via function calls, which executes network commands to an MDS. In this new model, the MDC can be replaced with a *caching MDC* which can talk to both a local and remote MDS. The local MDS is responsible for maintaining the local cache, either in memory or on disk; the caching MDC resolves cache misses and replays updates to the real MDS as before.

At some point, after a client has lost and reacquired a lock, we need a way to validate the data that already exists in the cache. Unix filesystems already provide the concept of a *change time* (ctime), which is updated whenever the inode changes. For Lustre directories we will add a new *subtree change time* (stctime) which will be updated whenever any inode in the subtree is changed. These stctimes have a nanosecond granularity and will allow a client to very quickly establish whether large portions of a cache are up to date.

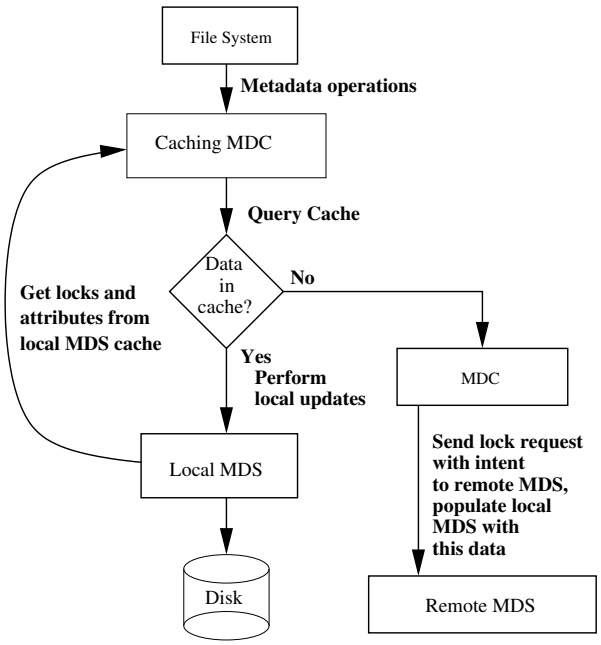

Figure 1: Persistent Caching

Updates to the sctime will of course dirty more (possibly many more) inodes during each update. For customers pushing their metadata server to the limits, they have the option of disabling the sctime and revalidating each object individually, or going without a persistent cache.

9 Collaborative Caching

A very common load pattern found in industry filesystem installations is one where read traffic vastly outnumbers write traffic. One such example is a cluster of web servers serving mostly static content.

Unless some effort is made to distribute the load in these situations, the servers will be completely overwhelmed, based purely on the raw bandwidth that a single server can provide. Consider the load placed on central servers if workstations have remote root filesystems and are all booted simultaneously following a power outage; or a lab of students all loading large applications at the beginning of a class.

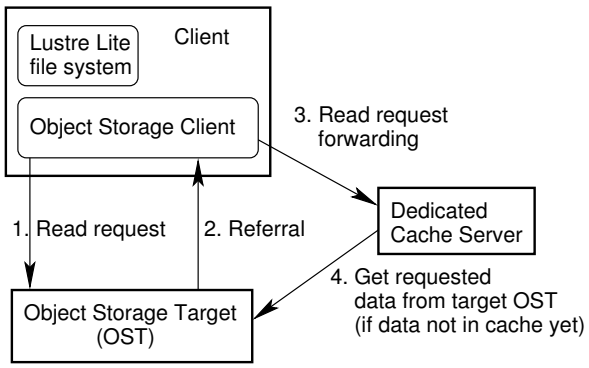

Figure 2: Collaborative Caching

Lustre is once again leveraging the object protocols into a symmetric collaborative caching device. Instead of working directly with an OST, a client communicates with one or more caching devices which can take locks on the client's behalf, locally service cache misses, and provide an alternative path to the high-demand data. These caching devices can also run on the clients themselves, which is the *collaborative* part of the collaborative cache.

With the addition of caching devices, recovery becomes somewhat more complicated. Normally the decision to give up on a particular OST following a network timeout is a very simple one. However with a cache in the middle it's important to distinguish between a failure of the cache node and a failure of the OST itself.

10 Conclusion

After more than a year of development, the Lustre framework has deviated surprisingly little from the original architecture. The lock manager and object protocols have served us well and will continue to form the centre of the design. Despite the fairly serious setbacks with

the VFS and client caching, our intent locking strategy has been shown to be successful on very large clusters and will be the primary mechanism for dealing with high-contention directories.

Today Lustre scales comfortably to more than 1,000 nodes and is running on 3 of the 8 largest clusters in the world[6] (at Lawrence Livermore and Pacific Northwest National Laboratories). We have every reason to believe that today's Lustre code will scale to 2,000 nodes without serious difficulties; the next two years of development are planned to address scalability and performance issues on a completely new scale of clusters which are only just beginning to be designed.

11 Acknowledgments

Lustre has benefitted significantly from the experience, guidance, and funding of several US Government national laboratories, notably Lawrence Livermore National Laboratory and Pacific Northwest National Laboratory. We have also received the support of the National Nuclear Security Administration ASCI Path-Forward Program, which provides funding for many of the advanced features in Lustre's future. Amongst our partners we are grateful for the support of and opportunities with Hewlett-Packard and Dell. The views and conclusions contained in this paper are those of the author and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of our partners or the US Government.

12 Availability

Lustre is released under the terms and conditions of the GNU General Public License, and can be downloaded from our FTP site or checked out of our public CVS repository. More information can be found at `http://www.lustre.org/`

Founded in 2001 by Dr. Peter Braam, Cluster File Systems, Inc. is a privately held company headquartered on the internet, with developers in five countries. CFS is focused on high-end storage solutions, including the development of advanced file systems, novel architectures, and the storage industry as a whole. More information about how we can improve your storage offering, business, or laboratory can be found at `http://www.clusterfs.com/` or by writing to `info@clusterfs.com`

References

[1] IBM, *Programming Locking Applications*, Version 4.3.1, Second Edition, 1999.

[2] Brightwell et al, *The Portals 3.1 Message Passing Interface*, Revision 1.0, 1999.

[3] J. Howard, *An Overview of the Andrew File System*, In Proceedings of the USENIX Winter Technical Conference, 1988.

[4] Satyanarayanan, M, Kistler, J. J., Kumar, et. al., *Coda: a Highly available File System for a Distributed Workstation Environment*, IEEE Trans. on Computers, 39(4): 447-459, 1990.

[5] Braam, Callahan, and Schwan, *The InterMezzo Filesystem*, In Proceedings of the Ottawa Linux Symposium, 1999.

[6] `http://www.top500.org/list/2003/06/`

Proceedings of the Linux Symposium

July 23th–26th, 2003
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
Stephanie Donovan, *Linux Symposium*
C. Craig Ross, *Linux Symposium*

Review Committee

Alan Cox, *Red Hat, Inc.*
Andi Kleen, *SuSE, GmbH*
Matthew Wilcox, *Hewlett-Packard*
Gerrit Huizenga, *IBM*
Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*
Martin K. Petersen, *Wild Open Source, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*