

Scalable Locking and Recovery for Network File Systems

By Peter J. Braam, peter.braam@sun.com

Petascaling computing systems pose serious scalability challenges for any data storage system. Lustre is a scalable, secure, robust, highly-available cluster file system that has been successfully deployed on some of the largest supercomputing systems in the world, including the BlueGene/L supercomputer at the Lawrence Livermore National Laboratory (LLNL), the Red Storm supercluster at Sandia National Laboratories and the Jaguar supercomputer at the Oak Ridge National Laboratory. This paper provides file system developers with insight into how network file system scalability is addressed in the Lustre file system through policies and algorithms that support distributed lock management and options for facilitating recovery after a compute node failure in a large scale cluster. These design approaches can be applied to the scaling of other file systems to support large clusters.

Locking strategies at scale. Protecting the integrity of cached data through the use of distributed locks 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 acquired by clients in a Lustre file system, 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 number 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 Lustre by skip lists, which make fast compatibility checks. First introduced by William Pugh, skip lists incorporate multiple pointers that aid in searching, as shown in Figure 1. When file extents are used, the lists of extents are replaced by an interval tree, making lock matching dramatically faster.

Figure 1. Skip lists showing linked lists with additional pointers

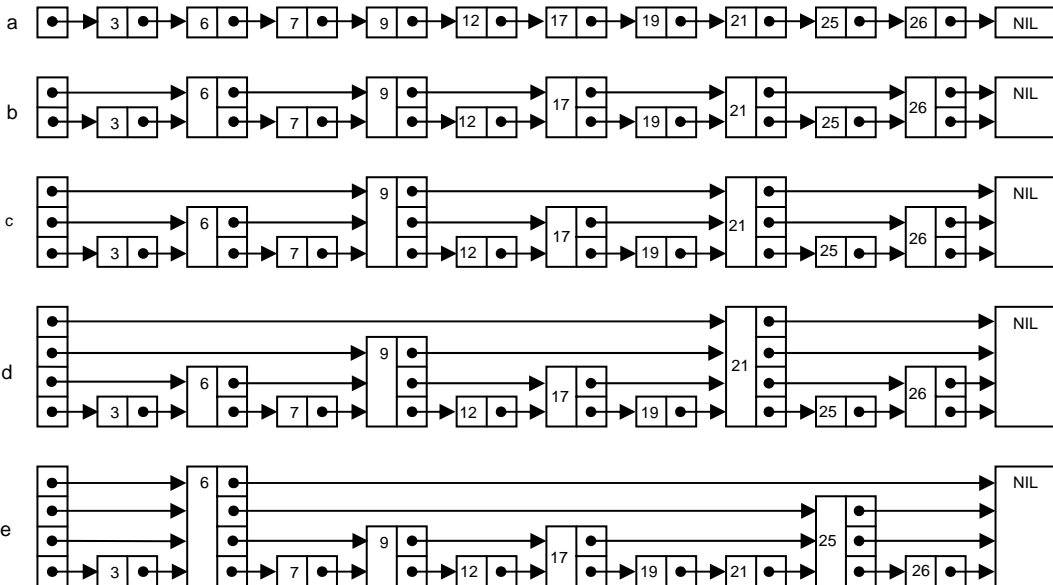
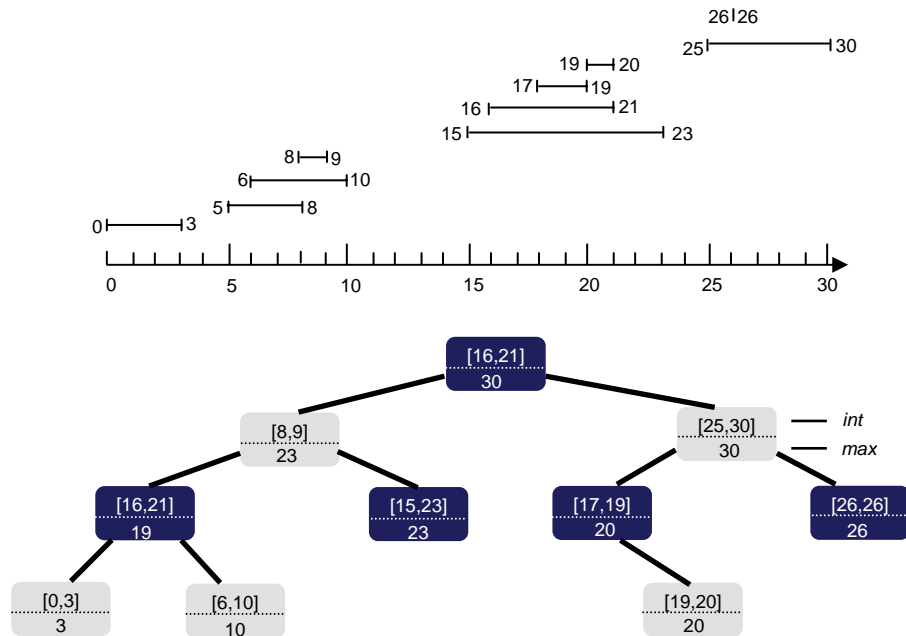


Figure 2 (top) shows a set of 10 intervals sorted bottom to top by left endpoint. Figure 2 (bottom) shows an equivalent interval tree.

Figure 2. An interval tree representing a sorted list of intervals

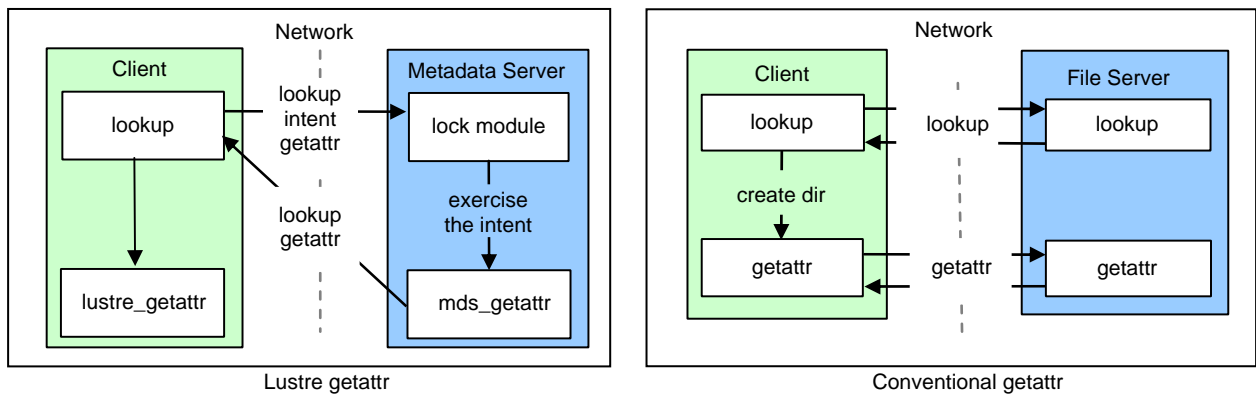


Lock contention. Lock contention occurs whenever one process attempts to acquire a lock held by another process.

- **Intent Locks.** For metadata processes, such as the creation of files, it is common for data to be heavily shared. For example, when all clients create files in one directory, all the clients need information about that directory. While traditional file systems granted each client, in turn, a lock to modify the directory, Lustre 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 may be fully executed without granting a lock. This allows Lustre 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 3 illustrates this behavior.

Figure 3. Single RPC metadata operations with intent locks



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

For the Catamount version of Lustre, called liblustre, which runs on the ORNL Jaguar system, adaptive I/O locks are mandatory for another reason. The Catamount nodes cannot be interrupted, and, as a result, lock revocations are not processed. Read locks can be revalidated upon re-use, but write locks are associated with dirty data on the client, so Catamount does not take write locks.

Recovery at scale. Like most network file systems, Lustre uses an RPC model with timeouts. On large clusters, the number of concurrent timeouts can 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.

Serialized Timeouts. Relatively early in the development of Lustre, scenarios in which timeouts happen repeatedly were observed to be commonly-occurring. Such a scenario may result when compute nodes in a group each create files in a directory and then 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 (let us call it the "listing" node) performs a directory listing with the "ls -l" command, problems can occur. The listing node tries 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 does a lock callback to the client, and, since the client is unreachable, the server callback times out. Even though callback timeouts take just a few seconds, with a few thousand clients, this leads to an unacceptable situation.

To address this problem, a ping message was introduced in Lustre. 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 the 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, therefore, safe for servers to hand out a conflicting capability.

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. Lustre has been adapted 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 pingging.

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 OSSs, 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:

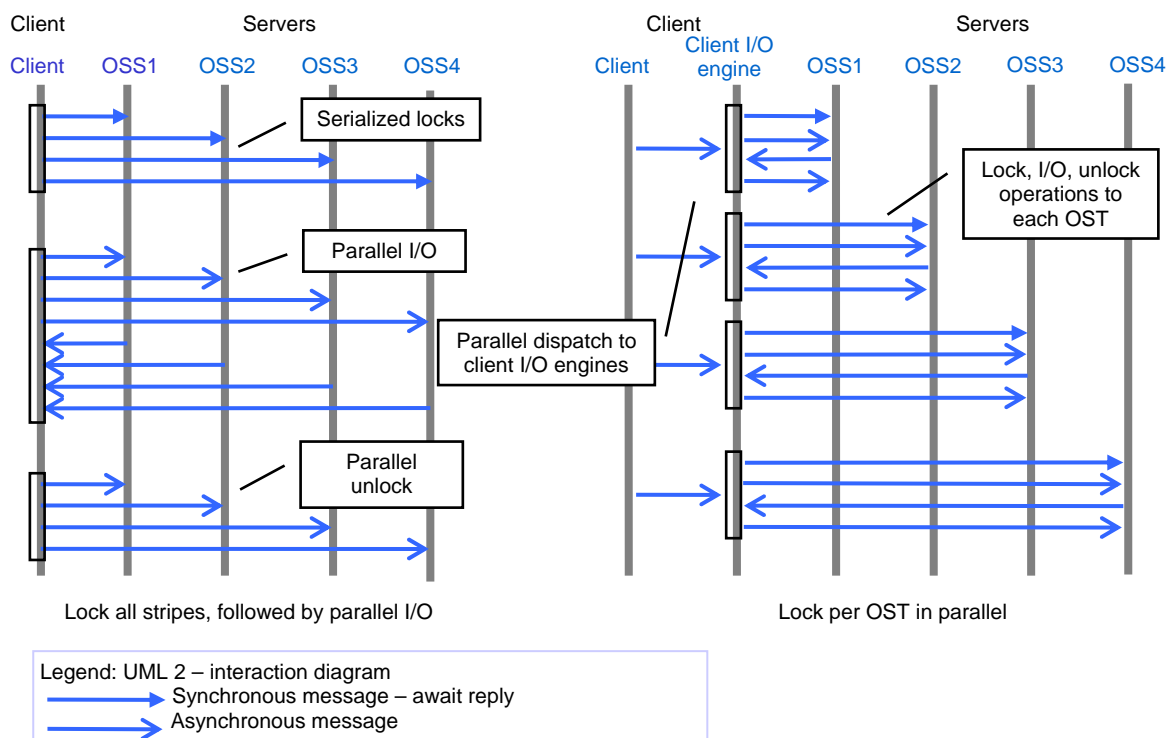
- The client receives a callback for a lock that has just been acquired.
- The client experiences a timeout while acquiring a further lock associated with the file extent.

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.

Lustre was changed 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 has greatly reduced the number of timeouts experienced on larger clusters, where problems with lock enqueues are relatively common.

A further change to Lustre introduced more parallelism to the I/O process to stripe objects. Figure 4 shows two interaction diagrams with this enhanced object locking mechanism shown on the right.

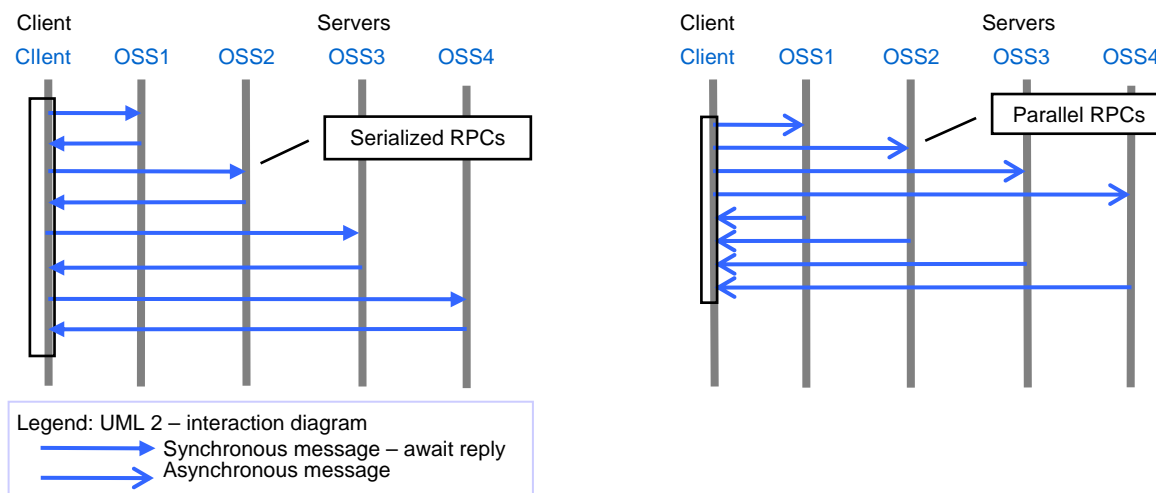
Figure 4. Object locking implementation in Lustre



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 Lustre locking behavior on the left is used.

This methodology also illustrates another scalability principle that was introduced for metadata, namely parallel operations. Whenever possible, the clients engage all OSS nodes in parallel, instead of serializing interactions with the OSS, as illustrated in Figure 5.

Figure 5. Parallel operations between clients and servers



Version Recovery. Lustre 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 aborts, leading to evictions.

Version recovery is being introduced into Lustre 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 retries the operation that was in progress and did not complete. After this retry, the client rejoins the cluster. Cached data is 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 appears to scale 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 re-integrate changes long after the initial recovery, for example, after a disconnection.

Adaptive Timeouts. Client requests see huge variations in processing time, due to congestion and server load. Hence, adaptive timeout behavior is very attractive and now available with Lustre.

When a server knows that it cannot meet an expected response time for a client request, it sends an early response to the client including a best guess for the required processing time. If the client does not receive such a message, the client assumes a failure of the server and, after the server recovers, resends the request.

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

Conclusion. Further challenges lie ahead as systems scale from petaflops to exaflops over the coming decade or two. Several aspects of Lustre file system infrastructure help make possible the scalability of a network file system to support very large clusters.