Lustre: A SAN File System for Linux

Peter J. Braam & Michael J. Callahan,
Stelias Computing Inc.

Abstract: We outline a design for a SAN File system for Linux. The file system will allow fine-grained sharing of data in a cluster environment. It has many components and is heavily inspired by Digital's VAXclusters. We discuss in some detail the interaction between such a file system and object based disks.

Motivation

Large-scale enterprise and technical information processing can benefit significantly from **clustering** software, which allows a network of workstations to exploit shared resources. Clustering software allows coordinated concurrent access to shared resources by the systems making up the cluster membership. For example, multiple computers can share a disk array.

This proposal focuses on exploitation of shared storage, in the context of a cluster file system. The benefits of a storage area network **(SAN) file system** like we are describing here are significant. A SAN file system:

?? gives higher performance, scalability and availability
?? de-couples storage and computational resources
?? can exploit current and future networked attached storage and interconnects
?? decreases system administration and TCO
?? supports advanced applications like Oracle Parallel Server and large-scale computation

Advanced applications such as cluster database and WWW servers, as well as large-scale scientific computation, like that performed in the National Laboratories, leverage shared storage to provide higher performance and scalability. These applications depend on cluster infrastructure both for hardware and software. This infrastructure provides basic services such as interconnects, storage and system communication protocols, cluster membership management, distributed resource and lock management, and access to shared storage volumes and file systems. Clusters can also give higher availability through redundancy and replication.

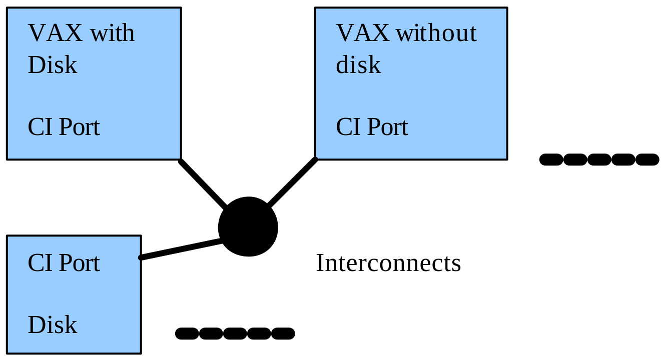

Figure 1: Vax Cluster Layout

Digital Equipment designed VAX Clusters in the early 80's and started shipping them around 1984. Digital provided custom hardware with the clusters in the form of interconnects (CI), network adapters (CI Ports) and network attached storage in the form of disks and tapes. The design of the system was documented in detail, and 15 years of research have only suggested minor improvements to Digital's design. Figure 1 shows a simple VAX Cluster layout.

The VAX Cluster product was a commercial success: many systems were sold, and their performance made them leaders in TP benchmarking contests. IT managers loved these systems, because the de-coupling of storage and computation made it extremely convenient to deal with growing demands on their systems, and to provide redundancy in critical situations.

The Linux operating system is now mature enough to add clustering software as a core component, and among principal file system designers there is much enthusiasm for an open design based on DEC's proven system.

Early involvement of storage and interconnect vendors in this development can enable the system to easily support new interconnects and new storage devices such as object based disks (OBD-s) and network attached storage. This could make such hardware successful, at least for Linux systems, independent of the cooperation of OS software vendors such as Microsoft and Veritas.

Design Goals

Before entering into more detail on our SAN file system design, we need to review the goals and constraints we accept in the design. Our design goals are tuned to cover a typical cluster environment. We want our file system to operate under the following circumstances:

1. A limited number of workstations have the shared file system mounted.
2. The workstations in the cluster fall under a single security domain; i.e. they trust each other.
3. The workstations are connected by a secure net, which is not connected to the Internet.
4. The file system can exploit shared resources such as disks in the workstations and network attached drives.
5. The file system will integrate with the buffer cache of cluster members.
6. The file system will use the disks with a protocol suitable for use with future object based disks.
7. Different types of interconnects among systems and storage can be exploited simultaneously
8. The file system has cluster-wide Unix semantics
9. If the cluster membership changes, due to adding or removing systems, the cluster file system will continue to operate after automatic reconfiguration.

Notice that these goals are very different from those of wide area network file systems such as Coda, AFS, DFS. Those systems address security as a concern, and make a strong distinction between clients and servers. Our cluster is to be regarded as one large server. The file system is available to software on cluster members and through this software running on the server can form a massive server, with high availability.

A limited number of workstations typically means a few to a few dozen. We would like to scale into the 100's, possibly 1000's to cover the case of large scientific clusters used for computation. However, it is likely that on such large clusters a less tightly coupled file system such DFS, Coda or InterMezzo is more suitable.

To achieve the security of the network, one would expect cluster members to have more than a single network interface. One would be for the outside networking through which clients of servers running on the cluster can access the services. Typically this will be a TCP/IP based network. Within the cluster we use a second network, which will typically be much faster. Fiber Channel, Memory Channel, Gigabit Ethernet and Myrinet are all good candidates. System to system communication is required, as well as system to storage communication, and the interconnect should support bulk transfers with large packet sizes.

Most cluster file systems to date have not integrated with buffer caches (the Berkeley XFS file system is an exception). We feel that for best performance it is necessary to allow file data to be fetched from the buffer cache of a cluster member, instead of always fetching it from a member mastering the disk.

To integrate with different storage devices, we need a storage device class driver that can perform I/O with local as well as remote disks. To access a remote disk, either in a remote host or in a networked attached device, a storage protocol should be used. Again, Digital set an example. VAX clusters exploited a Mass Storage Communications Protocol (MSCP), which could be used with network attached storage. MSCP is a higher level protocol than SCSI (but likely somewhat lower than the OBD protocols will be) and allows bulk transfers into contiguous ranges of virtual memory, with very few interrupts. The disk class driver would either access a local disk, or use the MSCP pr otocol over the interconnect to address disks in other systems or dedicated storage devices. For best performance, Digitals storage devices sometimes included a 32MB buffer cache on the device.

Unix file sharing semantics mean that writes are mutually atomic, and that the result of a write is immediately visible to all readers. Such semantics allow fine-grained sharing, suitable for processing database files. For this, distributed locking and notification mechanisms are needed.

Maintaining cluster membership is primarily necessary because after a network partition the systems in the cluster need to distinguish between being a member of a smaller valid cluster and being disconnected from the cluster. Such software is well documented, but necessarily complicated.

System Components

A fairly large collection of components is needed to build Lustre, our Linux cluster file system. Figure 2 gives an overview of the interactions between these subsystems.

At the lowest level we need drivers for network interfaces for all interconnects and storage devices. Partly such drivers currently exist, but one can expect that new drivers or enhanced capabilities are needed for a successful cluster. System to system communication and bulk transfer facilities involving DMA with few interrupts to hosts are vital for performance.

In conjunction with this we need a communications architecture capable of sending messages with **reliable delivery, broadcast traffic, bulk transfers, and unreliable datagram messages**. Digital's SCA (systems communication architecture) is a system with such capabilities. It does not seem complicated to build such as system on an existing infrastructure (like UDP), but retransmission algorithms have to be added to deal with reliability and so called **multi-rpc** facilities to avoid serialized waiting for multiple timeouts are very desirable.

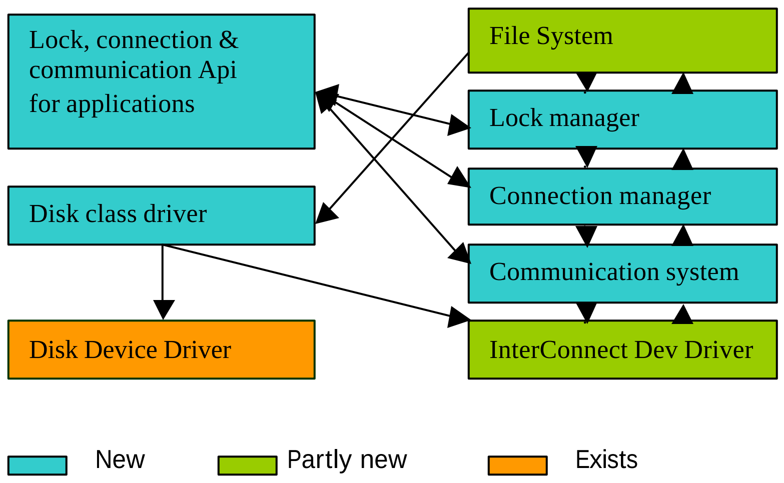

Figure 2. Lustre Component overview

The connection manager subsystem is responsible for maintaining the **cluster membership**. If any communications fail, due to a failed network or cluster member, the communications layer is responsible for supervising the recovery of the cluster. In outline this proceeds as follows. A non-recoverable communications failure for any subsystem raises an event in the connection manager, which immediately starts suspending all ongoing transactions in the entire cluster. The next task is to determine the current membership (through so called **quorum** mechanisms), to make sure that the system on which the event was raised is part of a valid cluster and not part of a breakaway group of failed systems. When cluster membership has been re-established other subsystems are brought back online and perform their recovery. This is done in an order of priority and the lock manager likely has the highest priority. Gradually all cluster subsystems resume operation.

The lock manager provides several pieces of functionality. First, it allows **resources** to be named, and it can appoint a **mastering node** for a new resource. Resources span a wide collection – some will be systems resources such as disk extents, free bitmaps, quota tables. Others will be created by cluster applications. Resources can be locked in many modes, for example a system can require an exclusive read/write lock on a resource. The acquisition of such a lock involves first finding the system mastering the resource. If the lock doesn't exist, it will be created as a new resource and it will be mastered on the node requesting the lock, thereby eliminating network traffic for locking the resource. If the lock already exists, the lock manager is asked to break existing locks. This is done by raising events in all nodes currently holding a lock, to notify them of the acquisition of the new lock. When these events have taken place, the lock is granted to the requesting system. Sometimes this takes time, and the request has to be queued before it can be honored.

There are two very challenging aspects to a distributed lock manager. The first is **deadlock detection**. Deadlocks occur when cycles exist between lock dependencies, the simplest being that A is waiting for B and B is waiting for A before can proceed. Now neither can go forward. In distributed lock management such cycles of dependencies can span multiple machines and are hard to detect. Detailed mechanisms about deadlock detection are given in "*VAXcluster principles*", by Roy Davis (Digital Press, 1993).

The recovery of the lock database after a cluster reconfiguration is another challenge. The problem here is that locks might be held for which the mastering node is no longer available. Again, the team at Digital has described in detail how their lock databases could be recovered.

(An interesting aspect of these descriptions is that they show how the evolution towards larger and larger systems led them to improve the algorithms between the early eighties and early nineties.)

The **disk class driver** is an area of particular interest for storage. Here we will house the software that can address disks on remote machines. It will be important that the disk class drivers address **logical volumes**, not just disks. Logical volumes can take several forms, such as locally attached devices, raid devices, devices on remote systems, or standalone attached storage devices. Linux already has network block devices that will probably be a good starting point to build a mass storage protocol for logical volumes. File systems would regard such devices as the devices hosting their data, and will want to acquire locks on extents in the volumes. Digital built very advanced storage devices that contained a buffer cache of up to 32MB of RAM. The MSCP protocol in conjunction with these buffer caches provided an interface to storage not very different from what object based disk array could handle. However, combining individual object-based disks into a logical volume is potentially tricky.

The **distributed file system** will be layered on top of this infrastructure. It will provide each system in the cluster with a consistent view of files on the devices. They three key aspects in correct design of such a file system are **pre-allocation, locking for currency** and **locking for exclusivity**. Pre-allocation will assign cluster members certain amounts of resources, in the form of disk blocks, quota, and inodes, which they can use without further communication. Locking for currency will allow cluster members to proceed on a *call-back* basis, that is, they can continue to use data until notified it is no longer valid. Such notifications are done with the lock-trigger mechanisms contained in the distributed lock manager. Locking for exclusivity enables *write back caching*. A node with an exclusive lock can continue to modify data until a reader on a different system declares interest in the data. At this point the lock manager breaks the lock and the system writing back must make the data available. Kirby McCoy's "*VMS File system Internals" (Digital Press, 1990)* has a detailed description of these mechanisms as they are done in the VAXcluster systems. Coda and other advanced distributed file systems have similar mechanisms.

Finally there will be applications that will exploit the cluster architecture. For these applications an interface to most subsystems must be made available. This applies in particular to the connection manager, the lock manager and the communications architecture.

Cluster File systems and object based disks

Professor Satyanarayan suggested an interesting design constraint on the file system: it should use **object-based disks**. An object-based disk has not completely been defined yet, but it would certainly be capable of creating objects similar to files on the drive, manage free space, and have an interface to read blocks out of files. It would manage the block allocation for its objects and hide this from the user. If the object interface were to include distinction between files and directory objects, the object-based disk would become an entire file system, and by reducing the interface to one where a single object resides on the drive, we return to a commodity disk. The value of the design constraint lies in the fact that it will prepare our file system to deal with future storage devices and it will give early indication as to what object based disk interfaces would need adaptation to be suitable for use in a SAN file system.

Lustre should aim to run on commodity hardware in the first instance, but be designed for all hardware. The relation ship between commodity file systems and object based file systems is explained in Appendix A.1.

The purpose of a shared file system is that data in the file system is shared between hosts. While many hosts may have read access, modifying an object will be a privilege exclusive to a single host at a time. For sharing a database file in a cluster, and modifying a record contained in a page of the file, the page would be *write locked* before the modification. For operations involving metadata such as removing a file, things are more involved.

When removing a file, we will be modifying several objects. First, a name will be removed from a directory file, secondly the blocks of the file we are removing will be freed, and third the file inode will be freed. It is desirable that locks naming the directory data and the inode, and the data blocks of the file are held by the remover, in order for the file system to migrate from one consistent state to another. So, file data may be shared at the level of an individual block, but meta-data changes must be carefully check-pointed to reflect consistent state in the file system.

It is important to realize that these locking constraints are managed through the distributed lock manager and they apply, for the most part, equally well for file systems for object-based disks and for traditional file systems. In fact the object-based disk is taking over a small part of the concurrency management from the file system. By adding locking semantics to a file system for object-based disks, a cluster file system can achieve correct sharing semantics when using object-based disks. However, such an object-based disk file system still needs to be designed and built – we have described a path towards that goal in Appendix A.2.

For concurrency the disk should allow multiple transactions to be in progress, but it can leave the interaction between transactions to the lock management in the file system [i.e. the file system will make sure not to start two transactions on the same object.]. Likely some markers are needed to clearly indicate which host initiated certain transactions, in order to deal correctly with cluster transitions.

Finally the object-based disk should allow multiple pre-allocation requests. Each of the systems in the cluster should be able to request a list of object id's it can allocate without contacting the disk. Similarly, each system should be able to reserve quota on the disk, so that it can proceed producing data in the knowledge that the disk will not be full when its data is finally written back.

Very smart object-based disks, for example those that know about the entire file system layout, become in effect a standalone file server, like a SNAP server. With these, the high availability and concurrency benefits of a SAN file system for a cluster are replaced with a single authoritative file server.

To summarize, object-based disks provide a fine platform for cluster file systems provided that they have a few minor features not yet introduced: multiple concurrent transactions initiated by different systems and support for multiple pre-allocation requests. Finally the disks should not be excessively smart, since then the advantages of clustering will be lost.

Data sharing and recovery

A key issue in file system design is recovery from crashes. Advanced Unix file systems such as IBM's JFS, Irix XFS and future versions of Linux Ext2 file system all handle this with journaling techniques. Journaling means that the file system and buffer cache have knowledge of when exactly the blocks in the file system are in a consistent state. As we have seen this is not an issue for a data blocks in a file, but for blocks describing the metadata, consistency only arises when well defined sets of operations have all been performed (as for removing a file). Figure 3 shows this in a diagram.

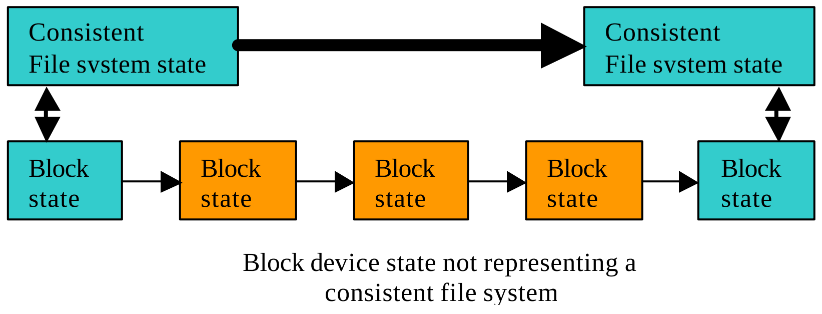
Consistent File svstem state
Consistent File svstem state
Block state
Block state
Block state
Block state
Block state
Block device state not representing a consistent file system

Figure 3: Metadata transitions between consistent file system states involve block device state not conforming to a consistent file system.

A journaling file system arranges that writes to disk always reflect a consistent file system state. It is important to note that the original BSD UFS file system did not do this, nor does ext2 for Linux. The price one pays by not doing this is a file system recovery process. Journaling does so even in the presence of failures, by first committing the blocks that need to be changed to a journal (on disk or in NVRAM). It then writes the data blocks to their final location. In case of a crash, all that needs to be done is to scan the journal and put the write the data blocks to the correct location – an almost instantaneous operation even for very large file systems.

In Lustre we want each cluster member sharing a device to have a journal on the shared storage device. The act of writing the journal will need careful coordination among the cluster members sharing the device. The updates to a single metadata object must retain the ordering they acquired through the lock manager.

With journaling present in the file system, it is also known exactly at which point data can be transferred from the buffer cache in one system to that in another. The lock manger will be building up a small lock hierarchy during metadata changes, and when these hierarchies have disappeared, the data is clean and can migrate. Several delicate details remain to be specified with regards to which system can journal and the precise detection of a hierarchy capable of migrating data.

The distributed lock manager and the file system

We envisage that we will build a lock manager much along the lines of the VMS lock manager. The VMS lock manager as well as its use for VAXClusters is described in great detail in the VAXCluster Principles publication.

The locks are equipped with versions, call-back functions and provide crucial features necessary for good file system performance. For read sharing **callback based caching** will be used. For modifications **write back caching** will grant exclusivity and eliminate write through bottlenecks**.** In each of the cases a cluster member can continue using a resource as if it were the only user, until notified by the lock manager. When the lock manager services a new lock request, it will break existing locks, at which time members either have to refresh the data, or make data it has modified available to other systems.

The implementation of such a file system can proceed through annotating an existing file system with lock requests. It might be beneficial to have a buffer cache with advanced features such as described in appendix A.2.

Connection management

Stephen Tweedie (RedHat Software) has designed a beautiful connection management system. We want to use this for Lustre. Stephen's system deals with multiple interconnects, and determines cluster member ship and recovery functions.

We would envisage that the most complicated components, the lock manager, distributed file system and recovery mechanisms, would be written by us - hopefully involving experts like Stephen Tweedie in the community. Disk class drivers, regression tests, communication software, and drivers can hopefully be written by others, but designed with our help.

Appendix: Object Based Disks

During the writing of this document it became apparent that more thinking needs to go into the use of Object Based Disks, possibly especially in the case of a cluster file system. This section contains a summary of our findings.

A.1 The relationship between object based disks and file systems

Object based disk interfaces can be constructed by dissecting existing file systems. Linux file systems interact with the remainder of the kernel though several **VFS interfaces**: the **file** operations, the directory cache (**dcache**), the **inode** operations and the **superblock** operations. There is also a well-defined API internal to the file system to manage free space, create new file inodes etc. What could be regarded as the lower half of the file system gives a good model of an object-based disk. In figure A.1, we have schematically drawn the interfaces that make up the object-based disk.

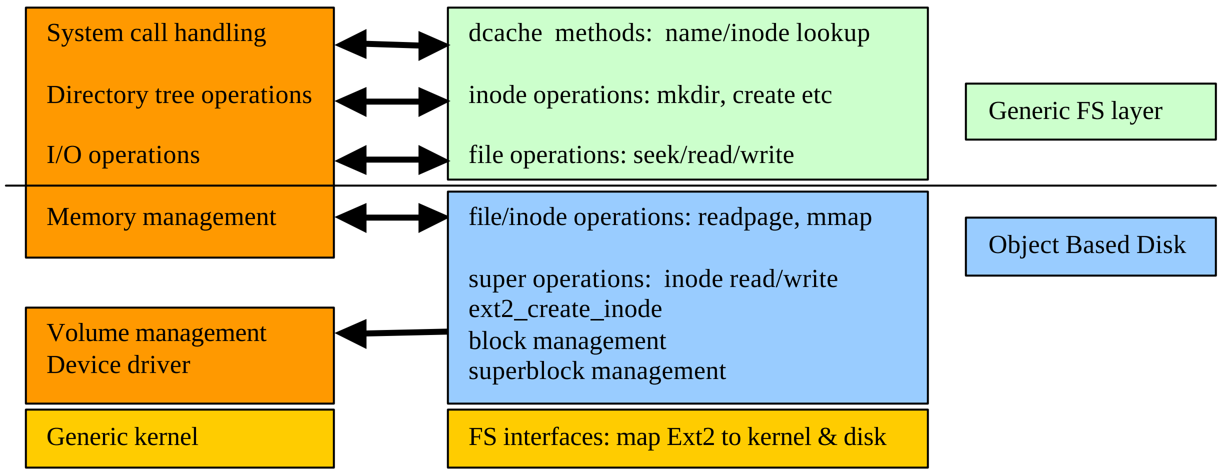
System call handling
Directory tree operations
I/O operations
Memory management
Volume management
Device driver
Generic kernel
dcache methods: name/inode lookup
inode operations: mkdir, create etc
file operations: seek/read/write
file/inode operations: readpage, mmap
super operations: inode read/write
ext2_create_inode
block management
superblock management
FS interfaces: map Ext2 to kernel & disk
Generic FS layer
Object Based Disk

Figure A.1: Object-Based disks within a Linux file system

A.2 File systems for object-based disks

A file system exploiting an object-based disk as its storage medium needs special support from the kernel, which is currently not present. This section briefly describes such support.

In a traditional Linux media file system, the *file_write* routines operate as follows. The file_write function is passed the inode that needs to be written, the offset from the beginning of the file where the data needs to be written, and the buffer with a known length. First the file system determines in which logical block of the file the caller wants to write. The file systems *bmap* routine is then used to map these to physical blocks and these are read into the buffer cache with the getblk function. Finally the data is copied from the user's buffer into the kernel buffers just allocated. The buffer cache is responsible for writing this back to the disk. This happens when the data has aged a little or when space is needed and buffers need to be freed. The buffer cache is designed to support standard block devices with this service. These devices perform I/O request based on a device and block number. At present network file systems, like NFS, cannot use generic buffer cache functions to do the I/O.

It is instructive to look at this routine in the context of an object-based disk. The key benefit of the disk is that the block allocation routines would not be needed, so we should be able to instruct the disk to transfer logical blocks into the buffer cache and from the buffer cache into the disk. In particular, we do not know the target disk blocks, and we could no longer rely on the existing buffer cache routines for writing back to object based disks.

The NFS file system faces similar issues and gives a preliminary solution, whereby the file system itself manages writing back the dirty buffers. The NFS solution is not really satisfactory since it may lead to unnecessary and premature flushing of buffers, and complicated race conditions could arise if memory were tight.

Likely the correct way forward is to modify the buffer cache to become knowledgeable about different ways of writing back buffers. A *buffer_head* could contain a pointer to private data and have methods to write back buffers. For an object-based disk, the private data would be a pointer to the inode, and the method for writing back would be the store_object method the disks offer. For NFS the private data would point to the file server, and perhaps also describe which part of the block would need to be sent back to the file server.

The generic mechanism would benefit both NFS, an object-based disk file system and other network file systems. Buffers can be kept around until they have aged sufficiently and when the bdflush runs it calls routines in the buffer method. It should be emphasized that changing the buffer cache will be a desirable but somewhat delicate modification to the Linux kernel. Ultimately, the object-base disk file system will be much simpler than a traditional file system.

Another interesting consideration in designing a file system for OBD's is the creation of new inodes. Here it is important to prevent synchronous calls to the OBD to get new inode numbers (object numbers as the disk sees it). The disk should be able to give the kernel a *pre-allocated* list of say 32 inodes that the file system can allocate for new objects, and similarly the file system should be able to request a certain amount of quota from the disk. In this way the file system can proceed to create files without contacting the device, and it can have certainty about the available space on the device.