Cluster File Systems

Dr. Steven Whitehouse

July 12, 2004

Abstract

Cluster file systems are a popular research area and fast becoming an important aspect of enterprise computing. This paper discusses their applications, performance characteristics and limitations and how these differ from a local filesystem. The concept of the GFS cluster file system is explained and its application in Storage Area Network (SAN) environments is discussed.

Introduction

In order to clarify the discussion below a definition of the term cluster file system should be given. It is reasonable to think that any file system can be used in a cluster environment given certain restrictions (usually, access by only one client at a time, or read-only access by several clients simultaneously). A suitable definition therefore might be a file system which allows the sharing of data across nodes in a cluster and that it must allow both read and write access (subject to any access restrictions placed by a particular site's security policy) in a symmetric manner and maintain data integrity and correctness at all times. In other words, all of the data stored on the file system should be available on all of the nodes. Using ext2, for example, mounted read-only across many nodes does not count!

A cluster file system should be scalable. Subject to some fundamental limits (as we'll see below) it should be possible to keep adding more and more nodes and more and more storage in order to scale the computing power of the overall cluster from a single node, up to many hundreds if not thousands of nodes.

Scalability in this case needs to apply not just to performance, but also to reliability. If it were a requirement that all nodes in a cluster must be functioning in order for any work to be done by the cluster, then with a finite reliability of each node, there would come a point beyond which it would be impossible to scale a cluster any further. There would also be a point, beyond which, it would not be economic to scale a cluster. In order to avoid this a cluster file system needs some built-in features to ensure that it can cope with the failure of other nodes in the cluster. Also scalability of administration is very important in that the less effort it takes to run the cluster the cheaper it is.

A cluster file system needs, far more than local file systems, to be upgradable whilst it is running. Having to shut down an entire cluster to perform an upgrade can be a financially painful experience for the cluster's owner.

A cluster file system needs to be a file system! Its obvious enough that if, in allowing access from many machines in a cluster, we compromise on functionality which is usually available to applications running on a local filesystem, then the result is that those applications will not run correctly on the cluster filesystem. Later it's shown where GFS doesn't quite live up to this and some hints and tips are given for application developers to consider when writing software which may run on a cluster file system.

There are a number of file systems which fall under the above definition and several more which almost fit. However in order to save this paper becoming a book, I will concentrate on just one, GFS, as an example.

The Buffer Cache

The buffer cache was developed as a method of speeding up access to files by keeping the most recently used blocks of data in memory. Linux has moved over to a page based model, however the principle of keeping data which is likely to be accessed in the near future in memory to speed up access times still applies.

This can naturally be extended over a cluster, so that a number of nodes can have a view of a file system and keep it in cache provided each node caches a part of the file system exclusively, or each node agrees that the part of the file system is read only.

The same performance rules apply on the cluster as on a the local file system: creating a system where data is always being written out to disk or read in from disk only once before it is ejected from cache (for whatever reason) will be much slower than one where the data is reused several times while it is still in cache.

In a cluster file system this is doubly true. To get access to data which is held by another node, a message is sent requesting that the data is written to disk by the node which is caching it. Once its written to disk by that node, it can then be read in again by the node which requires the data. So it takes a minimum of one network round trip time and two disk I/Os in order to move data from one node to another.

On the other hand, while data stays local to a single node, the file system is exactly the same in principle and operation as a local file system, and can run just as fast (potentially) as any other local file system.

The trick therefore in designing a cluster file system is to avoid bouncing data between nodes when not absolutely required. The same applies to application design as well, but usually there is less flexibility here since most applications are written without the considerations of a cluster file system in mind. For application designers who are lucky enough to be starting their design from scratch and know they are targeting a cluster file system, this is probably the most important point to bear in mind.

Network Attached Storage

Network attached storage can simply be a block device which is accessed by means of a network of some description. One or more client nodes access the server using commands which are very similar to those you would expect to send to a hardware disk drive. The servers are usually embedded devices running a standard operating system (Linux we hope!) or a modified version of it, even if thats not obvious from the configuration interface (which is normally web based or via a hardware serial

port).

The advantage of this over a direct connection to the disk drives is that the extra computing power of the server gives the opportunity to implement many useful functions on the storage server itself (snapshot, backup, authentication, encryption, etc) and the ability to present the same storage via several different protocols. The iSCSI protocols are becoming more popular as a cheaper alternative to fibrechannel. This can be run with standard Ethernet cards, but it is best when run on dedicated iSCSI hardware which can offload the processing of the SCSI protocol from the system's CPU.

There are also many NAS servers which present their storage at a higher level; SMB protocol (Samba) and/or NFS being the more popular ones at the moment. These are outside the scope of this paper however.

Storage Area Networks

Storage Area Networks (SANs) were introduced as a way to share large amounts of storage between computers in a flexible way. The usual example of SAN technology is fibrechannel which is also a very expensive technology. Its use tends to be limited to those with very large budgets. Although the fibrechannel NICs are not that expensive, fibrechannel switches are expensive and the alternative AL (arbitrated loop) topology doesn't give the required isolation between nodes for maximum reliability.

People are currently looking to protocols such as iSCSI with the hope that they can bring the features of fibrechannel storage at a price which is far less and we are starting to see that happening at the moment.

Another factor to take into account is that even when fibrechannel is in use, its not advisable to use it for intra-cluster communication. The cost in terms of latency of setting up a packet on fibrechannel is much greater than in sending traffic over Ethernet. Most clusters have a communication network which is separate from their storage network for this reason, as well as the more obvious one of increased bandwidth. Technologies such as Myrinet are able to combine high speeds with efficient processing of all sizes of message.

Some GFS History

A cluster file system doesn't exist in isolation on each node. There must be a method to synchronize the actions between each node in order to avoid chaos and data corruption. While it would be technically possible to create a synchronisation mechanism entirely by means of making atomic writes to sections of disk, the performance of such a scheme would be hopeless. An early attempt to solve the synchronisation problem without resorting to a separate communication mechanism was the SCSI D-Lock command developed by Steve Soltis and later by Sistina Software.

The availability of fast and gigabit Ethernet has meant that finding hardware and/or NAS with an implementation of D-Locks/hardware memexp is no longer a cost effective option by comparison.

Distributed lock managers have been available for many years, the most famous implementation being that on the VAX architecture. The concepts are rather different from the way kernel programmers normally think of locks (spinlocks/semaphores) in that six modes are available. Callbacks can be triggered to tell the

holder of the highest priority lock when a conflicting request is received. Also the concept of resources which have a hierarchical form allow the creation of a tree structure giving finer control at the lower levels.

The VAX lock manager could recover from any one of the nodes in the cluster failing through an algorithm which resulted from the way in which lock requests were granted.

Each resource was "mastered" by a single node in the cluster. By which we mean that a single node makes the decision about which of the lock requests for that resource will be successful, and which will be denied or queued. The idea is to try to ensure that if a node is requesting a lock repeatedly, that it becomes the master for that resource through a process called remastering. This way many lock requests can be granted without the requirement for any network traffic at all.

In the case of a failure, then one of several things might happen. If a lock holding node fails, this can be detected by the master and appropriate action taken (recovery followed by releasing the resource so iy may be used normally again). If the lock master fails, then its state can be recreated from the information on the lock holding nodes.

There is a special case in which the only lock holding node is also the lock master (for efficiency we'd like this to be the case most of the time) in which case all the resource's state is lost. In this case though since there was only a single user of this resource, and that user has crashed/broken or whatever, we know that it can be no longer using the resource and thus it doesn't actually matter that the state has vanished. This is a very important optimisation.

The correct recovery is subject to suitable measures being taken to protect the resources in the case of a failure from erroneous hardware. The original GFS implementation called this STOMITH "shoot the machine in the head" which has now been replaced by the more general term "fencing". This is usually done by isolating the failing machine either at the network level, with a fibrechannel switch for example, or more simply by using a network power switch to turn off the offending node. Pains must be taken to avoid any algorithm which can lead to domino effect, or similar problems. Site policy will determine whether automatic rebooting of nodes, of manual intervention is more suitable.

GFS

The GFS filesystem was born out of a research project being carried out at the University of Minnesota in Minneapolis into ocean currents. Matt O'Keefe's research group were looking for a solution to the problem of storing their (large amounts) of simulation data. Originally written for SGI's IRIX, it was ported to Linux, which sped up development considerably due to the open nature of the Linux source code.

The author's involvement with GFS dates back to early 2000 when he worked at a consultant for Sistina Software Inc. Since that time he has worked both on technical development and in pre/post sales support getting hands on experience of cluster file systems running real applications.

A number of the internals of GFS were based upon ext2 and offered, even on single hosts, an advantage for people with large numbers of files in single directories due to its hashed directory structure.

Another feature of GFS is the symmetric tree of metadata used to map the disk blocks to each inode. This means that no matter how

large the file, access times to every block is the same as every other block. GFS is fully 64bit and can handle very large files with few levels of metadata in the tree. Current versions of GFS are limited by the block layer in the Linux 2.4 kernel to a maximum file system size of 2Tb, but it will be possible to expand to much larger sizes with the 2.6 kernel without even unmounting the file system.

Application Optimisation

Although we stated above that cluster file systems should provide all the semantics of a local file system, it will always be possible to get greater performance by understanding the limits and possibilities provided by cluster filesystems.

In addition, this section could also be subtitled “VFS Annoyances” since it covers some areas where the Linux VFS as it stands does not quite provide all the functions required of it by a cluster file system.

File Locking

There are two kinds of file locking supported by the VFS, flock and fcntl type locks. The flock type allows locking of whole files and is relatively straight forward to implement using a lock manager (this is how GFS provides this function).

The fcntl style of locking offers greater functionality and is used more widely due to its support of range locking and (depending upon configuration) on NFS filesystems. It also has a feature whereby you can request the process ID of a process which is holding a lock which would cause another lock to block. This is often then used to send a signal to the locking holding process to request that it release the lock.

The problem with this approach is that in the cluster case, the process may very well not be running on the local node and therefore the signal may fail, or worse still, be sent to a completely unrelated process.

It is a difficult problem to solve. One possible way to do it might be to segment the process ID space so that each cluster node has its own range of PIDs. Then it would be possible to ”hook” the signal system call so that it too could work across the cluster. On the other hand, GFS is supposed to be a file system and not a complete cluster environment, so that the addition of an extra fcntl call which returns a node ID as well as a process ID might be more appropriate. The disadvantage of this approach being that it is not transparent to the application.

Directory Notification

The Linux VFS provides a set of functions whereby a process can register to receive notification of events in a certain directory, such as creation of a new file being created etc. This is used in applications such as graphical file managers which update their displays of the filesystem when it changes.

The question then is how to make this work across a cluster. The Linux VFS provides no way for a file system to know which process, or even whether there are any processes awaiting notification. The VFS handles all the notification logic itself in the upper layers.

There is no way for a file system to send a message to the VFS to say that an event has occurred, since this function is simply not required for local file systems at all.

As a result, directory notifications on GFS

will only work if the listening process is on the same node as the process which made the modification. It isn't entirely fair to blame the VFS here. Even if the functions were available, there is actually considerable difficulty in creating a system that would allow the notifications to work across a cluster and still remain scalable. My feelings are that the best way to achieve this particular goal would be to use a VAX style lock manager to generate a scalable notification mechanism.

Write access to executables

The Linux VFS contains functions called `get_write_access` and `put_write_access` which are used to deny write access to files when they are executable files which are being executed. The locking is taken care of by the VFS which is fine for local file systems, but means that it is not possible to make this function work across a cluster without some modifications.

Once the functions are exported though, it would be a trivial task to make this work across a cluster given that the locking functions required are already present in the filesystem.

VFS Locking and Hints

There are several system calls where the VFS calls into a filesystem more than once. This usually happens when a lookup is required for some operation followed by the actual operation itself.

As an example, consider the `truncate` system call. It looks up an inode, given a path and then later calls the truncate VFS method. During this time, the VFS provides protection from other processes accessing the inode, but this doesn't apply to operations occurring on other nodes. It is possible for the permissions on the inode to have been changed from another node between the initial look up and the actual truncate.

GFS gets around the problem at the moment by doing the permission check again after the VFS has done the preliminary checks. It is not possible for the file system to get a lock during the look up and hold it in anticipation because should the VFS tests fail, the file system will not receive any notification and the lock would be held forever.

This problem has been addressed in the 2.6 kernel by the addition of `struct nameidata` to the `lookup` call of the `struct inode_operations` (and indeed some of the other VFS operations). This allows the VFS to inform file systems the `intent` union about what the look up is for. The file system can then do all its checks and leave its locks in the correct state for the operation to be performed.

Memory Mapping

Linux's memory subsystem, like the VFS, wasn't designed with cluster file systems in mind. A short cut, which gives a performance advantage for local file systems, results in a page being mapped into memory read/write if the mapping is a writable mapping even if the memory access which triggered the page fault was only a read.

This can cause problems in a clustered situation where many of the accesses to the mapping might just be reads, but still cause write locks to be taken out across the cluster (and thus lead to undesirable cache behaviour).

It would therefore be useful to be able to return a read-only mapping of a page in the case

that the mapping is read/write so that the extra overhead of getting write locks is not taken on reads. It could of course also be argued that this is effectively an implementation of distributed shared memory (DSM) and therefore that we shouldn't encourage applications to use it, but many database and similar applications make extensive use of memory mapping to access their files, so providing them with a way to share data across a cluster without making extensive changes to them seems a sensible goal.

With the 2.4 kernel, a patch is required to the memory subsystem in order to allow the operation of writable shared memory mappings at all. This is due to the order of locking within the kernel as the currently provided hooks are not suitable. Work has recently been done to change that situation and a patch is in some vendor kernels at this time.

Conclusion

From the above discussion, you might get the impression that due to the number of areas of the VFS which currently do not provide for cluster file system support that such file systems are not very useful. In fact, there are very many applications for which none of the previously mentioned areas is a problem. GFS has been successfully deployed in many clusters, both large and small running applications for ISPs, calculation and simulation and other large scale computing applications.

The journaling and recovery code ensures that the overall reliability of a cluster is maintained even when a node experiences hardware or software failures. This is essential in large clusters in order to maintain cost effectiveness.

The comments about the VFS and memory management in the last section are intended to be partly a useful list of pitfalls to watch out for if you are an application developer, and partly as a starting point for further discussion in the kernel community. Some of them are known problems which are being worked upon and solutions may even have been created by the time of this conference. Others are slightly less well known and I haven't seen any discussion about them on the mailing lists.

GFS 6.0

Red Hat has recently announced the availability of the source code for GFS 6.0. Details can be found on the Red Hat web site at the following URL: `http://sources.redhat.com/cluster/` along with the other components of the Red Hat cluster project. There is a source RPM package for Red Hat Enterprise Linux 3.0 (2.4 kernel based) and a CVS server containing the 2.6 version of the source code. A snapshot of the CVS is included on the conference CD along with this paper.

Some differences will probably exist between GFS 6.0 and this paper as the source has only been on public release for a relatively short period of time before the submission of this paper. I will try to cover any major changes in the talk itself.

Acknowledgements

The author would like to acknowledge the assistance of Sistina Software Inc (now part of Red Hat Inc.), Red Hat and also UKUUG for inviting me to present this paper.