File Systems for Clusters from a Protocol Perspective

Peter J. Braam
School of Computer Science
Carnegie Mellon University

Abstract: The protocols used by distributed file systems vary widely. The aim of this talk is to give an overview of these protocols and discuss their applicability for a cluster environment. File systems like NFS have weak semantics, making tight sharing difficult. AFS, Coda and InterMezzo give a great deal of autonomy to cluster members, and involve a persistent file cache for each system. True cluster file systems such as found in VMS VAXClusters, XFS, GFS introduce a shared single image, but introduce complex dependencies on cluster membership.

1. Introduction

Distributed file systems have been an active area of research for more than two decades now. Surprisingly, the design of some extremely well engineered systems such as Sprite and VAXClusters are not widely known and are not part of the Linux offering yet. Simple systems such as NFS have become common place, together with the complaints accompanying their behavior. However, the more complex systems have not yet achieved sufficient maturity to replace NFS. In this paper we will describe some protocols in detail and make brief comparisons with other systems.

We will first give a detailed outline of NFS and AFS functionality. There are very interesting differences between these, well described by their protocols.

We then turn to Coda and InterMezzo where client autonomy rules. Coda's protocols have been extended with features for server replication, disconnected operation and InterMezzo engages in write-back caching through client modification logs. The semantics of these file systems are different from Unix semantics.

SAN file systems such as VAXClusters (Lustre), XFS, and GFS allow systems to share storage devices. These systems provide a tightly coupled single image of the file system. Here lock management and cluster membership becomes a key ingredient.

Key differences between the design of these file systems can be understood through analyzing what is shared. For NFS sharing is at the level of the vnode operations. For Coda and InterMezzo sharing is almost at the granularity of files. All read and write locking to deliver semantics, is managed by the file servers. In contrast, SAN file systems share storage block devices and negotiate access through a distributed lock manager.

We conclude our paper by discussing the implications for use in clusters. Necessarily we are looking at all available systems, but merely highlighting a few systems with which we are more familiar.

Acknowledgement: This research was supported by the Air Force Materiel Command (AFMC) under DARPA contract number F19628-96-C-0061. Intel and Novell provided additional support. The views and conclusions contained in here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of AFMC, DARPA, Intel, Novell, Carnegie Mellon University, Western Washington University or the U.S. Government.

2. Sharing files and Caching

2.1 NFS

The simplest file sharing protocols, such as those in NFS, share files through an api on the client. This api is implemented through remote procedure calls and it is largely similar to the api used by an upper layer of a local file system when retrieving or storing blocks or querying directories. In particular, the protocol makes no assumption about server state, and exploits little or no caching.

The NFS protocol is easily described. The server answers requests for:

1. **Read only operations:** getattr, lookup, readdir, read, readlink, statfs, mount
2. **Tree modifications:** mkdir, link, symlink, rename, unlink, rmdir
3. **File modifications:** setattr, write

Most NFS implementations try to do a limited form of caching, with doubtful semantics such as "re-use a data block if less than 30 seconds old". Modifying operations must reach stable store on the server before returning on the client. Security in NFS is arranged by trusting the client and communicating the user id of the calling process to the server.

Implicit in the design of systems like NFS is a desire to provide Unix semantics among the systems sharing files. For tightly coupled systems on fast networks and not too many clients, this is a reasonable goal. With large numbers of clients or networks of lower bandwidth, this goal will severely compromise performance.

NFS v3 and v4 refined this protocol by making it possible to relax the synchronous nature of updates, and to fetch attributes for directory entries in bulk. Among other variations on NFS Spritely-NFS is worth mentioning - it has a much-improved caching scheme.

NFS is widely available. Large specialized NFS servers are available from corporations such as Network Appliances, and most free Unix clones come with NFS version 2 and 3.

2.2 AFS

We now turn to AFS. AFS was the first file system to exploit a persistent cache on clients, which caches both file and directory data. AFS is a commercial file system available from Transarc, but a free clone named ARLA is available and inter-operates with deployed AFS systems.

During normal operation, clients can assume that data is valid, until notified by the server with a CallBack, or InitCallbackState RPC asking the client to discard one or all cache entries associated with the calling server. This will happen when other clients are about to make updates. AFS has “last close wins” semantics, as compared with Unix’ “last write wins” semantics.

AFS caches directories on clients and once objects have been cached with callbacks read only access does not involve any server traffic. The AFS cache is persistent. Even after a reboot of a client or server, cache entries can be re-used by exploiting an attribute named “DataVersion” associated with each file.

The namespace of all data made available by AFS servers is unified under a single directory “/afs”. Client administration is minimal, it is merely necessary to inform the client of the names of AFS cells that need to be accessed. Each cell is a group of AFS servers falling under a single administrative domain. Each cell exports volumes, (a.k.a. file sets) which have “volume mount points”, in the /afs tree. When a client traverses a mount point in the /afs tree, it contacts a volume location server in the appropriate cell to find the servers storing the volume.

The AFS protocol is simple and powerful. Unlike NFS both clients and servers must answer RPCs. The client does this to keep informed about currency of data.

A. Client interface:

Semantics: Probe, Callback, InitCallbackState.

B. Server interface:

1. **Read only operations:** FetchData, FetchACL, FetchStatus
2. **Tree modifications:** RemoveFile, CreateFile, Rename, Link, Symlink, MakeDir, RemoveDir
3. **Storing:** StoreData, StoreACL, StoreStatus
4. **Volume calls:** GetVolumeStatus, SetVolumeStatus, GetRootVolume
5. **Liveness and semantics:** GetTime, BulkStatus, GiveUpCallBacks
6. **Locking:** SetLock, ExtendLock, ReleaseLock

AFS has powerful security. Clients are not trusted, and must connect with credentials obtained from authentication servers, usually in the form of Kerberos tokens. An ACL based protection scheme supplements this authentication.

2.3 DCE/DFS

Further developments from AFS led to the DCE/DFS file system part of OSF and documented in detail at [DCE/DFS].

On the servers a complete re-engineering of AFS took place, with VFS+ support in the file system layer, to export local Episode file systems through the kernel level file exporter interface. Fileset support was added to Episode and usage of the local file system does not compromise semantics of DFS. The server interfaces were extended with several key ingredients:

1. **Token management:** clients can obtain tokens on objects such as files and extents. Single system Unix semantics are available through DFS.

2. **Translators:** by adding readdir and lookup calls to the server interface, DFS file exporters can serve NFS and Novell clients.

3. **Volume level callbacks:** just like in Coda volumes can be validated with a single version vector if no changes have taken place.

4. **File level access control lists:** to refine security the AFS directory level ACLs were supplemented with ACLs per file.

File space on servers is not directly accessible in AFS and Coda, as it is in simple systems like NFS. The cache file and directory data exploited by AFS are stored in a private format. Ficus and DCE/DFS do this for server space only through stackable layers and VFS+ respectively, see [FICUS], [DCE/DFS].

On the client side, DFS resembles AFS with the main differences coming from token management allowing for fine-grained sharing.

DFS 1.2.2 is available in source form. It is an extremely large system and has not been ported yet to Linux.

3 Coda, Ficus and InterMezzo

3.1 Coda

Coda descends from AFS version 2. Coda adds several significant features to AFS offerings: disconnected operation with reintegration, server replication with resolution of diverging replicas and bandwidth adaptation. A multi-RPC protocol that allows multiple requests to be dispatched and responses to be collated without serializing remote procedure calls supports Coda replication.

To support disconnected operation Coda performs operation logging when servers are not available to make synchronous updates, or when bandwidth makes this unattractive. The operations are written in a client modification log, the CML, which is reintegrated to the server when connectivity allows. During reintegration, the server replays the operations it finds in the CML. When the server has processed the directory modifications in the CML, it back-fetches the file data from the client. Coda's bandwidth adaptation adapts the rate and delay at which the CML is reintegrated to the server based on perceived network performance. To deal with client, server and network failures during reintegration, the Coda version stamps have been extended with a host id of the client performing the update. Upon reconnection Coda can rapidly validate cache-state using file and volume version stamps. To prepare for disconnection, Coda keeps collections of files hoarded in client caches.

Server replication in Coda is achieved by assigning a volume storage group to a newly created volume, consisting of the servers holding the volume data. Coda uses a write all, read one model. To guarantee consistency the basic invariant is that equality of the version-stamp of individual replicas of files and directories guarantees their equality. Before a client knows the version stamps of all available replicas are equal it will not use the data. However, a client will accept an incomplete available volume storage group (AVSG), i.e. availability of a subset of the full volume storage group (VSG).

During modifications with an AVSG smaller than the VSG, servers will log the modifications made to the file system, much in the same way as clients use a CML while disconnected. To keep replicas in sync, Coda uses two phase update protocol. First each server holding a replica performs the update and reports success or failure to the client. To complete the update the client installs a final version-stamp and vector on each of the AVSG members with a COP2 multi-RPC. Usually, these COP2 messages are piggybacked on other RPCs, but they will be delivered within 10 seconds.

If servers in the AVSG return different version-stamps, clients initiate resolution. Resolution is a server to server protocol exploiting version vectors in conjunction with version stamps and server modification logs.

The version vector of an object count the number of updates seen by individual servers and allows for simple reconciliation of differences in important cases such as objects missing from some of the servers ("Runts"). This is based on partial ordering between version vectors, which can lead to a dominant replica of an object. A single WeaklyResolve RPC can resolve these cases.

More delicate divergences of data need a 5 step resolution protocol between the servers. A randomly chosen coordinator first locks volumes. During phase 2, server modification logs and version stamps are obtained from all subordinates, including the coordinator itself for the so-called closure of set of diverging objects. The coordinator merges these logs and sends them out to the subordinates.

Subordinates parse modification logs and deduce operations missed and proceed to perform these missing operation in phase 3. Subordinates now return list of inconsistencies, if any, that arose. If inconsistencies arose these are marked in Phase 4, and a user will use a repair tool to choose which replica of the object to retain. Where no inconsistencies were found each subordinate installs a final version-vector during Phase 5, and logs are truncated if the set of subordinates equals the VSG. The resolution is now complete.

Coda extends the AFS interface with several calls:

A. **Client interface**:

Backfetch

B. **Server interface for clients:**

Reintegrate, Repair, Validate, Resolve, COP2

C. **Server resolution interface**:

- **Simple Runt resolutions:** WeaklyResolve
- **Full resolution:** Lock, FetchAndMerge, ShipAndPerform, MarkInconsistencies, InstallFinalVV

UCLA's Ficus project shares many features with Coda. We refer to [Ficus] for details.

3.2 InterMezzo

InterMezzo is a prototype file system exploiting Coda style reintegration as a write back caching mechanism, as well as for disconnected operation. InterMezzo uses the local disk file system for cache and server space through a filtering file system (a.k.a. stackable layer) and does write-back caching at kernel level, with very little overhead. See the references [InterMezzo], for details.

InterMezzo is implemented with very little code and while in early stages of development it looks promising.

The write back caching protocol in InterMezzo is based on permits. These are obtained before modifications can be made, and must be surrendered, together with the modification log when a server revokes the permit. Future versions of Coda will support a similar permit mechanism. Like DCE/DFS InterMezzo performs bulk lookups of directories.

4 Cluster File Systems

4.1 VAXClusters (and Lustre)

A true cluster file system was available from Digital Equipment Corporation as early as 1985. VAXClusters have a magnificent architecture, which is extraordinarily well documented in [VAXFS] and [VAXCL].

VAXClusters maintain membership based on quorum, which prevents cluster fragmentation. Cluster members can share designated storage devices, residing either in systems or in network attached storage devices. A variety of interconnects can be used.

The VAXClusters file system is derived from the local file system by annotating the access to data to closely interact with the distributed lock manager. The lock management protocol is what mostly controls the features and semantics of the file system.

Locks are obtained for resources, which are named and organized in trees. To acquire a lock, ancestors must be locked first. Each lock hierarchy, i.e. the tree of resources under a root lock, has a mastering node, which is usually the first system to acquire a lock in this tree. To find the node mastering a resource, querying a distributed resource directory, based on a hash value of the resource name, follows self-inspection. If the resource does not yet exist, the system looking up the resource becomes the mastering node.

Locks can be held on cluster members by processes or by the operating system in a variety of modes: exclusive, protected read or write, concurrent read or write and unlocked. A simple bitmap defines compatibility between these 6 modes. When a member requests a lock in a new mode, all nodes holding the lock in a mode not compatible with this request are notified and execute a callback function.

Each system can obtain a great deal of autonomy, through locks allowing write back caching, quota usage and reading without contacting the lock manager. The organization is broadly similar to the design of AFS callbacks, and many details are given in the quoted references.

Several important points should be made about this design. When a member leaves the cluster, for example through a failure, a complex recovery process is needed. Unlike systems like AFS and Coda, there is no client server relationship between the cluster members, and recovery from failed members is much more involved.

Secondly a major difference between the design of VAXClusters and the systems discussed in the previous section lies in the assumption that the cluster members trust each other, and have a secure network for intra cluster communication.

All systems engaging in locking of file objects must worry about deadlocks. There are two ways to deal with these: one can acquire locks using an ordering, in which deadlocks will not happen. If an ordering cannot be used, deadlocks can arise and must be detected. A general-purpose distributed lock manager like the one found in VAXClusters can detect deadlock.

The Lustre project (read Linux clUSTRE) aims to build a similar cluster file system for Linux.

4.2 The Berkeley family: Sprite, Zebra and XFS.

Sprite [Sprite/Zebra] was a distributed operating system with an extremely interesting design. Sprite supported a Unix API, process migration and a totally transparent shared file system. Sprite typically ran in clusters consisting of several dozens of systems. The Sprite file system made extensive use of caching in VM and provided Unix semantics, as well as extensive facilities for process migration, remote device access and communication with user level servers. Sprite allows caching only as long as there was no read/write or write/write sharing on a file. By forwarding open calls to the server, caching was disabled when a non-exclusive writer appeared. The Sprite client/server interface for ordinary file service superficially resembles that of NFS and AFS, but precise timing and semantic constraints render Sprite's file system quite different.

It is interesting that the SMB file system used in Windows [CIFS] has many semantic similarities with Sprite. Unfortunately the SMB protocol is extraordinarily elaborate and exists in rather many dialects and doesn't resemble any of Sprite's clean design.

Zebra [Sprite/Zebra] was developed towards the final phases of the Sprite project and provided striping across disk arrays. Zebra did not stripe individual files, but instead used a per client log, and striped the log. Zebra relied on a single file manager to locate the servers holding data and to manage cache consistency.

The XFS file system [XFS] evolved from Zebra towards a file system for storage area networks. In XFS the distinction between clients and servers is blurred. In XFS there is no centralized manager, and the motto is "anything anywhere".

Perhaps the most transient feature of XFS is that the authoritative copy of the data may be fetched from the buffer cache of a client, eliminating the server as a point of central control.

Frangipani-Petal [Frangipani/Petal] is a Compaq project to build clustering for shared storage, somewhat related in its goals to XFS.

4.3 GFS

The Global File System [GFS] is a SAN file system project at the University of Minnesota. GFS worked with industry to exploit shared locks, implemented on the disks, so called dlocks. GFS can exploit logical volumes, has no central point of management and is not log-based. GFS manages its own disk layout, with support for filesets and tree-based directories. Initially the file system followed a write through model. The system is not fully featured as of the time of this writing, but its results are promising. GFS is reported on elsewhere in this workshop.

5 Conclusions for Clusters

From the earlier sections in this paper, it is clear that different file systems offer different level of sharing:

- informal sharing: NFS
- sharing across file closes: AFS, Coda, InterMezzo
- single image sharing: Cluster file systems and DCE/DFS

To achieve various forms of file sharing, there are other solutions apart from file systems, such as using a user level library. PVFS [PVFS] is a library that provides applications with further I/O capabilities.

For Linux clusters it would be most desirable if all choices were available and offered the highest levels of robustness and performance. Unfortunately at present this is far from true. Linux needs better NFS, as well as totally mature AFS, Coda, InterMezzo as well as GFS and Lustre file systems.

6 References

[CIFS] See: http://www.cifs.com/

[Coda/AFS] See: http://www.coda.cs.cmu.edu, http://www.cs.cmu.edu/, http://www.stacken.kth.se/project/arla/

[DCE] See: http://www.opengroup.org/dce/.

[Ficus] See: http://ficus-www.cs.ucla.edu/travler/

[GFS] See: http://gfs.lcse.umn.edu/

[InterMezzo] See: http://www.inter-mezzo.org

[PVFS] See: http://ece.clemson.edu/parl/pvfs/index.html

[Sprite/Zebra] See: http://www.cs.berkeley.edu/projects/sprite

[XFS] See: http://now.cs.berkeley.edu/

[Franginpani/Petal] See: http://www.research.digital.com/SRC/personal/thekkath/frangipani/home.html

[VAXFS] VMS File System Internals, Kirby McCoy, Digital Press 1990.

[VAXCL] VAXcluster Principles, Roy Davis, Digital Press, 1993.