

Survey Project: A study on distributed file-systems

Subhradyuti Sarkar, Siddhartha Saha
Project Report, CSE 222, Fall 2003

1. Introduction:

The tremendous increases in the processor speeds have rendered the I/O subsystem of any computer as the bottleneck of the system. Advances in communication technology, in the other hand, have allowed connecting a number of machines to each other to build a cluster of systems to achieve far more processing power and storage capacity than that can be achieved in a single machine. The inability of the I/O systems to cope with the virtually unbounded processing power is more apparent in the clusters. Moreover, the demand for a fast and efficient network file system is always there. It is interesting to note, even though a distributed LAN and a cluster computer serves completely different kind of applications, all efficient distributed file system implementation that is suitable to either of these two scenarios, share some common features. In this project, and subsequently, in this report, we will explore some of the distributed file systems and their salient features. We begin with some earlier file systems like Zebra and AFS and move on to other interesting works in the field like the Petal/ Frangipani duo and Expand. We conclude with one recent and commercially used file system: GPFS from IBM, and one that is being used for quite some time now: XFS from SGI.

This report is organized in several sections. The next section highlights some of the common features of distributed file systems in general. After that, over several sections, we present brief overview and key features of few file systems. Next, we present the conclusion and there we give a very brief view of the current work that is being done.

2. Common characteristics of distributed file and storage-systems:

In this section we shall highlight a few issues (roughly in the order of relevance), which concerns the designers of any distributed file-system. Most of the file-systems discussed in the report have incorporated some policies/mechanism/optimizations etc. to address those issues.

1. Uniform name-space: One of the chief motivations for using distributed file-system is sharing and accessing files in a *uniform* way across the network. Hence all distributed file-systems create an illusion of virtual directory hierarchy of the shared storage which is visible from any client workstation. Some file-systems like Andrew require that all user files should be stored in the shared storage accessed through `/afs/...` (Thus enforcing strict uniformity in the shared name-space). Widely popular NFS is more flexible in this respect – any NFS client can store files in local as well as remote storage, and the shared directory structure can be mounted as an arbitrary subtree in the local file system.

2. File replication and backup: Distributed file-systems are usually deployed in such areas where reliability is a key issue. Users of a distributed file-system want seamless access to their files even when some file server fails or a magnetic disk crashes. Almost all distributed file-systems replicate and backup files to enhance reliability. There is a key difference between backup and replication. Backups create a checkpoint in the file-system, so that the system could be restored to that checkpoint if some disaster strikes in the future. Replication is the policy of keeping redundant copies of a file, to make sure that the file remains accessible even if one/more components of the distributed file-system fail.

3. Support for configuration change: Distributed file-systems must be able to balance itself when the physical or logical configuration of the computing environment changes. By configuration change we mean that disks and file-servers can be added to/removed from the file-system or the network connection between the file-servers might be upgraded or disrupted. The file-system should adapt itself to the new scenario and deliver the best possible performance with minimal intervention of a human administrator.

4. Maintaining cache-coherency: Distributed file-systems are used in a client-server computing environment – where the file-servers store the files and the client-workstations access them as a when required. However, since most of the modern workstations possess significant amount of disk space, they can cache files to minimize network overhead and exploit temporal locality. Local caching introduces the problem of maintaining cache-coherency: if client A updates a file in its local cache, client B might get a stale copy of the file. Distributed file-systems can use either optimistic or pessimistic policies to enforce some file access semantics (which might be different from the standard UNIX semantics).

5. Managing network partitions: Many distributed file-systems are used over wide-area networks. In such cases it is not uncommon that the network might get partitioned into a few clusters, and two replication of the same file might be accessible in two different partitions. If this occurs the file-system should either disable read-only access to those *stranded* files, or must run conflict-resolving programs to merge different versions of the same file after network operations resume.

6. Increasing throughput by parallel access: Maximum bandwidth of a disk is bounded by its hardware interface (IDE/SCSI/Fiber-channel), and usually much less than the processor-memory bandwidth. We can exploit the *distributiveness* of the file-system by storing different parts of the files in different disks and accessing them parallel. This technique, called *File-Striping* is widely used in distributed file-systems.

7. Support for synchronization and locks: Supporting robust synchronization and locking primitives is a vital issue in the design of any file-system, and it is more so in the distributed case. Most distributed file-systems support single-writer/multiple-reader access through multiple instances of dedicated lock servers.

8. Robustness: Distributed file-systems must recover from system crashes with minimum downtime, so they normally use clever techniques like read-ahead logging that can quickly detect and fix corruptions in file-system data structures.

3. Zebra Striped File System:

Zebra[4] is a network file system that applies two ideas those were originally developed for managing local disk subsystems and applies them to a network file system in order to increase the throughput. The ideas that are borrowed are the log structured file system (LFS) and striping with parity calculations. Each client using a Zebra file system organizes its file data in an append-only log structure like LFS and stripes this log across several file serves. The file servers in a Zebra system are usually implemented using RAID. The following points summarizes the essential features of Zebra:

- **Per Client File Striping Instead of Per File Striping:** There are some disadvantages of striping in a per file basis in a network file system that is trying to provide high performance throughput. Per file striping performs really badly for small files, and there is an additional overhead of parity computation. Zebra solves this problem by using per client striping. Each client maintains an append-only log, and stripes it across the servers. This log file idea follows that of LFS. This way

the servers are used efficiently regardless of file sizes, and small writes are batched together to perform efficient writes.

- **Central File Manager:** Since each client writes its own log, it is necessary to have some mechanism to share files across clients. Zebra introduces a central file manager, which essentially manages the metadata of the files and supervises interaction between the clients. The file manager stored all of the information in the file system except for the file data. In Zebra, file manager, being a centralized resource, can be a bottleneck of the system. All the file open requests come to this file manager, which then returns back the address of the blocks where the file is present in the network. When large files are being accessed sequentially, Zebra optimizes the performance by prefetching large amount of data.
- **Delta:** Deltas are small data structures in the log file that identifies the changes to the blocks in a file, and different types of deltas are used to communicate these changes between the clients, the file manager, and the stripe cleaner. The deltas provide a reliable and simple way for the various system components to talk to each other.
- **Stripe Cleaner:** As in the case of LFS, the log files of the clients that are striped across the file servers will be fragmented over prolonged usage. The Stripe Cleaner in Zebra does exactly the same work as the cleaner in LFS. But unlike LFS, in order to have the file system up and running even when a stripe cleaning is in progress, all the modifications done by the Stripe Cleaner is recorded in the special file called *stripe status file* in terms of Deltas. The Central File Manager consults this file if any new file system modification requests come in the interval in which the Stripe Cleaner was running. This approach makes the system usable even when a stripe cleaning is going on.
- **Crash Recovery:** Essentially, most of the crash recovery of the system is done using the log replay mechanism. If a storage manager crashes, then the data can be recovered by using normal RAID data recovery schemes, as all storage manager disks are configured using RAID. But if the Central File Manager crashes, then the system can be in jeopardy. In this way, the file manager is a central point of failure of the system.

One more interesting observation regarding this system is that it does not talk of network partitioning. But considering the fact that this is one of the early approaches for high performance network file system implementation, we may conclude that this problem of partitioning was not apparent at that time.

4. Andrew File System (AFS):

Andrew[5] was designed to be a ‘scalable, secure and highly available’ distributed file-system. It was a long-term project, and modified over a number of generations (AFS-1, AFS-2, AFS-3, and Coda). The chief design goals of AFS were to provide a uniform name-space to every client, secure and authenticated access to files and directories, excellent scalability and graceful degradation on the face of untoward situations like excessive load or network partition. The following are the salient features of Andrew File System and its evolution:

- **Vice and Venus:** *Vice* is the server component of the AFS which creates the illusion of uniform name-space. Each AFS client would run a process called *Venus* which would contact *Vice*, and mount the distributed directory structure in a local path called */afs/*. Only temporary, boot-time or cached files are stored in the local storage system.
- **Caching Policy:** AFS-1 used a pessimistic caching policy where every cached file was suspected as stale. Thus every file access would require a *Venus-Vice* communication, even if the file is cached. It turned out that pessimistic caching is detrimental to AFS-1’s performance, and AFS-2 implemented a form of optimistic caching. In AFS-2 whenever a client cached a file, the server *promised* to send intimation if another client updated the file.

- **File Access Mechanism:** In AFS-1 all files were referred by their entire pathname, there were no notion of UNIX-like *inode* or *inumber*. If an AFS server did not store a requested file, the search would eventually hit a *stub* directory which points to the actual server. This simple scheme was significantly improved in AFS-2, and the concept of unique identifier for each file/directory was introduced. Also, in AFS-2 files were stored in a new logical entity called *Volume* – which was also the chief mechanism to enforce user-level quota management. Fast, read-only replications of Volumes were possible as a potential back-up solution.
- **Security:** File level security was enforced through access-control list which could contain both access and deny permissions. Identity of the users was first verified by a password and then the corresponding Venus client was given a unique token to prove its authentication in subsequent transactions. This two-level authentication policy ensured that passwords are exchanged a minimum number of time over insecure network.
- **AFS-3:** AFS-1 and 2 were built keeping the local area network in mind. AFS-3 was introduced to support a distributed file-system over wide-area networks and decentralized administration.
- **Coda:** AFS users were very much inconvenienced in the case of network failure. Coda is improvement over AFS that can handle network disconnections more gracefully. Coda users can specify a set of *favorite* files and folders which can be cached in local disk and seamlessly accessed even when the network fails. A comprehensive conflict management policy in Coda ensures the distributed file-system would be in a coherent state after normal network operations resume.

5. Petal and Frangipani:

Petal[2] and Frangipani[1] are closely related to each other. They take a two-tiered approach to implement reliable, high performance distributed file-system. In a distributed storage environment, typically there are many file server having one or more storage devices. Petal virtualizes the actual hardware configuration to a set of virtual disks. Those virtual disks can be addressed like normal disks by a <virtual disk id, block offset> tuple. Frangipani is a distributed file-system that uses the service of Petal, and can provide uninterrupted access to user files irrespective of changes in the hardware configuration of the storage servers. The key design features of Petal and Frangipani are as follows:

- **Modules:** Petal consists of five modules - Liveness, Global state, Translation, Data Access and Recovery. For normal operation we need the service of Translation and Data Access module. The Liveness and Global state modules exchange periodic ‘I am alive’ packets to detect whether a Petal server has crashed. The Global State module is also responsible for updating the address-translation data structures when a Petal server is added to/deleted from the network. The Recovery module is activated during crash recovery.
- **Mapping Structures:** Petal keeps three mapping data structures: Virtual Directory, Global Map and the Physical Map. Using them it can translate a <virtual disk id, offset> tuple (supplied by a Frangipani server) to actual <server id, disk id, offset> tuple.
- **Snapshots:** Petal supports taking exact snapshots of any virtual disks at any time. Snapshots are labeled with an epoch number, and can be accessed with any other virtual disks but only with read-only access. Frangipani file system can be backed up by just taking a snapshot and copying it to tape.
- **Fencing and Reconfigurations:** Disks and server can be added to a Petal server ‘online’, and Petal automatically adapts to the new configuration. When a new server is added, the existing data is incrementally redistributed amongst all servers by a novel technique called *Fencing*.
- **Chained-declustering:** Petal replicated data using a scheme called *Chained-declustering* which can tolerate single server failure. In this scheme, if a Petal server fails, two neighboring servers

can automatically divide its workload. Also, data can be retrieved even in the rare case of site failure by grouping the *even* and the *odd* servers in two different physical locations.

- **Frangipani and Petal:** Frangipani runs on top of Petal, and serves the request of the user processes. Each Frangipani server can access only a single Petal virtual disk, though. Frangipani servers may run in each client machine (better load-balancing) or in the Petal servers (better security). Petal exposes a virtual disk interface having a huge 64-bit address space. Frangipani uses this address space wisely to create separate areas for file-system logs, inodes, allocation bitmaps and file data blocks. Each file can contain about 1 TB data, and there can be almost 2^{24} such files. It is very unlikely that these limits would have to be revised in the near future, even if we consider high growth of data density on magnetic disks.
- **Locking:** Frangipani implements elaborate locking protocols using *Lock Servers* and *Lock Tables*. There are multiple instances of Lock Server in the system, thus eliminating any single point of failure. Locks can be obtained only as *leases*, which must be renewed periodically. Hence if an entity holding a lock crashes, the lock is revoked automatically and recovery module is initiated. Frangipani solves the infamous *deadlock* problem by ordering the locks and enforcing a two-phase locking.
- **Read-ahead Redo Log:** Frangipani keeps a read-ahead redo log in a specified portion of the Petal server. Thus, if a Frangipani server crashes, another server can read the log and repair the file system metadata.
- **Adding and Removing:** Adding a new Frangipani server requires minimal administrative overhead, since the new server only needs to be told which Petal virtual disk and lock server to use. Shutting down a server is all the action required to remove the server.

6. Expand:

Expand[6] is a parallel and fault-tolerant file-system based on the popular NFS protocol. The main design goal of Expand is to use standard software components, maintain compatibility with existing systems, implementation with minimal modification of the operating system (on both server and client side) and achieving high throughput by allowing parallel access to both data of different files as well as data of same file. Expand incorporates a few simple yet novel design decisions, which are discussed below:

- **Structure of Expand files:** Expand stores only a part of the file in a physical disk-partition, and the whole file is accessible from a logical, *distributed* partition. On a distributed partition the user can create either striped file with cyclic layout or fault-tolerant file according to RAID5 scheme. Each subfile of an Expand file has a header that can store key information like the stride size and actual cyclic layout. The NFS server which stores the first block of the file is called *base node*.
- **Metadata management:** With striping, amount of metadata to be managed and processed increases significantly. To balance the load of processing metadata amongst Expand servers, one subfile of a file is marked as 'master node' and metadata is kept only at the header of the master node. The 'master node' is determined by applying a uniform hash function to the file-name.
- **Parallel Access:** Expand references to a file using virtual file-handles, which are aggregate of k NFS file-handles (provided the file is striped across k partitions). When a file request is submitted to Expand, it resolves the address of the actual NFS servers from the virtual file-handles and issues k simultaneous requests to the actual servers using parallel threads.
- **Fault-tolerance:** Fault tolerance in Expand is provided by the RAID5 technique. If specified by the user, NFS can transparently insert a parity block after appropriate RAID5 prefix. Such configuration can tolerate the failure of a single I/O node or a NFS server.

7. XFS:

XFS[3] is a general-purpose file system from SGI. It was designed to scale to very large file systems and to provide very high performance I/O. The primary factor contributing towards the scalability of the file system is the use of B+ trees to keep track of free space, to index directory entries, to keep track of dynamically allocated inodes scattered throughout the file system. XFS was designed to address several problems apparent in older existing file systems like EFS. XFS was designed for high performance file and file system access, and it can run well on large striped disk arrays. Some of the essential design features of XFS are mentioned below:

- **Allocation Groups:** XFS is a 64-bit file system. The file system is divided into partitions called allocation groups (AG). Each AG has its own separate data structures for managing free space and inodes. The free space management in an AG is done by using a pair of B+ trees instead of traditional block oriented bitmaps. One of the B+ trees is indexed by the starting block of free space and the other is indexed by the length of the free extents.
- **Large File and Directory Support and Extent Map:** XFS provides a 64-bit sparse address space for each file. The files can have hole in between, without any disk space allocated to them. Since there will be huge number of blocks in one such file, XFS uses an *extent map* rather than the block map. An *extent* is a contiguous range of blocks allocated to the file. The extent map is also managed using a B+ tree. XFS can also support a large number of files in one directory. An on-disk B+ tree structure is used to implement the directories. To make the management of large amount of contiguous space in a file efficient, XFS uses vary large extents and extent descriptors.
- **Crash Recovery:** In a large file system, examining all the metadata after a crash takes too long. Thus, in order to support fast crash recovery, XFS logs all structural updates to the file system metadata. User data is not logged. The transactions are logged asynchronously in XFS in order to improve performance.
- **Contiguous File Allocation:** XFS uses *delayed file extent allocation* techniques to allocate file data contiguously. Delayed allocation is a kind of lazy evaluation algorithm, which buffers data blocks in memory and builds up a virtual extent. Real disk blocks are allocated only when the data in the memory buffer are flushed to the disk.
- **Direct I/O:** Direct I/O is a mechanism similar to DMA transfer which lets a process read/write data bypassing the buffer cache. But the applications using direct I/O need to make the read/write requests respecting the block boundaries of the system.
- **Transaction Log and Parallelism in I/O:** XFS is designed to run well on large-scale shared memory multiprocessors. In order to support parallelism, the transaction log is the only centralized resource in XFS. This is the most contentious resource of the file system. All the metadata updates pass through this log. For each transaction, it provides a buffer space to copy the metadata update, writes the update to the disk, and notifies the transaction when the log writes complete. XFS allows multiple processes to access a file simultaneously in parallel. When using Direct I/O mode, multiple writers can write to the same file simultaneously

8. GPFS

GPFS[7] is a modern high performance and scalable parallel file system for cluster computers and it is presently being actively used in real implementations of high performance cluster computers. Parallelism is the key word in the design goals of HPFS. The high throughput guarantees of this file system also come from extensive parallelism inherent in the design of the file system. Even the administrative tasks of adding or removing disks, or, rebalancing files across disks can be done in parallel in GPFS. Even with

this amount of parallelism, this file system tries to follow the behavior of a general purpose POSIX file system. This is a highly scalable system. Currently GPFS disk data structures supports file systems with up to 4096 disks, each up to 1 TB of size each.

- **Striping and shared disk architecture:** GPFS system consists of the cluster nodes, connected to the disks or the disk subsystems over a switched fabric. All nodes in the cluster have equal access to the disks. Files are striped across all the disks in the file system, this provides load balancing as well as achieves full throughput of which the disk subsystem is capable.
- **File allocation:** Large files in GPFS are divided into equal size blocks, and consecutive blocks are placed on different disks in a round robin fashion. The block size is usually high (~ 256k) to minimize seek overhead. GPFS stores small files in smaller units called sub-blocks, which are as small as 1/32 of the size of a full data block.
- **Prefetching:** Prefetching is widely used in GPFS. The system issues I/O calls in parallel to several disks and prefetches the data into the buffer pool of the client. One interesting aspect of the system is that GPFS recognizes sequential, reverse sequential as well as various forms of other access patterns to make prefetching more efficient. If some application has some special data access pattern, GPFS has an interface through which the application can notify GPFS its data access pattern.
- **Directory Hashing:** To make lookups in a directory containing millions of files GPFS uses extensible hashing.
- **Metadata Journaling:** For crash recovery and to check file system consistency, GPFS does metadata journaling. Each node has a separate log file that is readable by everybody. So, any node can perform recovery on behalf of the failed node. Dirty metadata are periodically flushed to the disk, hence the size of the log file is bounded.
- **Distributed Locking:** The GPFS architecture is fundamentally based on distributed locking. Byte range locking is used for user data that provides very fine grain parallel and shared access to files, both read and write. The implementation of byte range locking is also very efficient. This allows parallel applications to write concurrently to different parts of the same file. Byte range locking is managed by tokens. A token represent a right to a lock for a particular object. A token manager is responsible for distributing tokens.
- **Parallel Access to Metadata:** Metadata access for a file in GPFS is synchronized by electing a *metanode* for the file. All clients accessing the file will modify the inode of the file locally and send the modifications to the metanode, which is responsible for merging all the updates in a proper way.
- **Free Space Allocation Map:** The free space allocation map is divided into fixed, large n number of separately lockable regions and each region contains the allocation status of $1/n^{th}$ of the disk blocks on every disk in the file system. One of the nodes is designated as the allocation manager and it is responsible for the coordination between systems regarding free space management.
- **Partitioning:** A communication failure may cause a partitioning of the network. To ensure error free operation, GPFS allows access to the file system data only by the group containing a majority of the nodes in the cluster.

9. Conclusion:

It is evident that the technology and ideas that goes in implementing an efficient, scalable file and storage management system for distributed computing environments and cluster computing has come a long way. But there are still problems, for which solutions are not efficient, or, do not scale. Research is going on to provide more and more throughput, performance guarantees, scalabilities etc. Other high performance file systems like Hurricane [8], Clusterfile [9], Hermes [10] etc which specializes for a particular type of

application and system. [8] is tuned especially for shared memory multiprocessors, [10] uses magnetic RAM enabled storage. Along with the scalability and performance issues, some other issues need to be investigated. With the advent of Storage Area Networks (SAN), the need for a high throughput, reliable, and scalable file system for SANs is required, in which there will not be any dedicated file servers. Intelligent disks will be attached to clients through a switching matrix. The complete file system will be managed in a distributed fashion. Amongst the file system that we discussed in this report, GPFS provides some promising schemes that can be incorporated in such a system.

One possible research improvement in which some work is going on is efficient buffer space management and cache management for a distributed file system. There has been a tradeoff between multiple parallel access and cache complexity. Most implementations try to avoid cache coherency problems due to the increased complexity. There are definitely some areas of improvement in this aspect of distributed file systems.

The way technology is growing nowadays - the rate at which the processing power is increasing is far more than the rate at which the performance of the storage systems is growing. With the phenomenal increase in processor speed, in order to alleviate the problem of the disk system being a bottleneck, file system scalability will be an active area of research in foreseeable future.

Acknowledgement:

We would like to thank our instructor Geoff Voelker for his valuable guidance in helping us choose the papers to review in this project. Without that, we would simply be lost in the vast literature available in the field of distributed file system.

References:

1. C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: A Scalable Distributed File System. In Proceedings of the 16th ACM Symposium on Operating Systems Principles, Oct. 1997
2. E.K. Lee and C.A. Thekkath. Petal: Distributed Virtual Disks. Proc. ASPLOS-7, Oct. 1996
3. A. Sweeney, D. Doucette, W. Hu, C. Anderson, M.Nishimoto, and G. Peck. Scalability in the XFS FileSystem, Proceedings of the USENIX 1996 TechnicalConference, pages 1-14, San Diego, CA, USA,1996.
4. J. Hartman and J. Ousterhout. The Zebra Striped Network File System. Proc. 14-th Symposium on Operating Systems Principles, pages 29--43, December 1993
5. Mahadev Satyanarayanan. Scalable, Secure, and Highly Available Distributed File Access, IEEE Computer 23 (1990)
6. F. Garcia, A. Calderon, J. Carretero, J.M. Perez, J. Fernandez. A Parallel and Fault Tolerant File System Based on NFS Server. 11-th Euromicro Conference on Parallel Distributed and Network based Processing Genoa - Italy PDP2003February, 5-7, 2003
7. F. Schmuck and R. Haskin, "GPFS: A Shared-Disk File System for Large Computing Clusters," In Proc. of the First Conference on File and Storage Technologies (FAST), Jan. 2002
8. O. Krieger and M. Stumm. HFS: A performanceoriented flexible file system based on buildingblock compositions. In Fourth Workshop on Input /Output in Parallel and Distributed Systems, pages 95--108, Philadelphia, May 1996
9. Isaila, F.Tichy, W.F. "Clusterfile: a flexible physical layout parallel file system", Cluster Computing, 2001. Proceedings. 2001 IEEE International Conference, 2001
10. Miller, E.L.Brandt, S.A.Long, D.D.E. "HeRMES: high-performance reliable MRAM-enabled storage" Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop, May 2001