DiFFS: a Scalable Distributed File System

Christos Karamanolis, Mallik Mahalingam, Dan Muntz, Zheng Zhang
Computer System and Technology Laboratory
HP Laboratories Palo Alto

HPL-2001-19
January 24th , 2001*

E-mail: {christos, mmallik, dmuntz, zzhang}@hpl.hp.com

distributed file system, storage management, Storage Area Network (SAN)

Industry analysts see no limits to the world's expanding appetite for data-storage services. Emerging networking technologies allow incremental scaling of bandwidth and capacity of storage, which is attached to the network. A key challenge is to devise the software that provides transparent shared access to decentralized storage resources. Existing network file systems will not meet the scalability requirements of future storage services.

This paper introduces DiFFS, a distributed file system designed for storage area networks. DiFFS achieves high scalability by following a partitioning approach to sharing storage resources. The architecture is robust against failures and unfavorable access patterns. It is independent of the physical file system(s) used for the placement of data; multiple file systems can co-exist in a DiFFS system.

DiFFS: a Scalable Distributed File System

Christos Karamanolis, Mallik Mahalingam, Dan Muntz, Zheng Zhang

Hewlett-Packard Labs
1501 Page Mill Rd, Palo Alto, CA 94304, USA
{christos,mmallik,dmuntz,zzhang}@hpl.hp.com

Abstract

Industry analysts see no limits to the world's expanding appetite for data-storage services. Emerging networking technologies allow incremental scaling of bandwidth and capacity of storage, which is attached to the network. A key challenge is to devise the software that provides transparent shared access to decentralized storage resources. Existing network file systems will not meet the scalability requirements of future storage services.

This paper introduces DiFFS, a distributed file system designed for storage area networks. DiFFS achieves high scalability by following a partitioning approach to sharing storage resources. The architecture is robust against failures and unfavorable access patterns. It is independent of the physical file system(s) used for the placement of data; multiple file systems can co-exist in a DiFFS system.

1. Introduction

Demand for data-storage services is growing rapidly and is expected to grow even faster in the coming years [1]. A primary reason for this growth is the appetite for storage by Internet Data Centers (IDCs) that provide Internet and application services to corporate and private customers. Servicing data out of physical storage is the task of a file system.

Physical storage is cheap and will become cheaper. At the same time, Storage Area Networks (SANs) allow the incremental scaling of storage bandwidth and capacity by directly connecting pools of storage to clusters of servers. A number of commercial storage products and experimental file systems have been taking advantage of SANs, e.g. Storage Tank [2], Tivoli [3] and EMC HighRoad [4].

There are numerous research and commercial systems that provide various flavors of distributed file systems. They fall into two main classes: *cluster file systems* and *wide-area file systems*. Cluster file systems [5, 6] guarantee strong consistency for shared files. The file system is accessible via a cluster of strongly synchronized servers. Cluster file systems have an inherent scalability problem, because all cluster members contend for the system-wide resources. Wide-area file systems [7] provide access to large, geographically distributed storage, where the requirements for file sharing may not be as strong. They use aggressive caching to scale and consistency may be traded for performance.

DiFFS is a SAN-based distributed file system architecture with scalability and performance characteristics superior to the two approaches above. It avoids the inherent scalability problems of cluster systems by partitioning the storage; shared access is controlled on a per-partition basis. It facilitates the concurrent use of multiple physical file systems to accommodate diverse application requirements. Such flexibility is becoming increasingly important in IDC environments. DiFFS is designed to tolerate host and communication failures without sacrificing failure-free performance. Robustness against unfavorable access patterns is achieved by file-level migration.

2. Architecture Overview

DiFFS uses partitioning to address the problem of contention for storage resources. Each partition consists of a partition server and some storage (which may be part of a SAN) as shown in Figure 1. Examples of clients include web and application servers, in an IDC.

2.1. Storage Partitioning

Each partition is a self-contained physical file system, with its own resources (inodes, blocks, block maps, etc.). It is implemented over a "virtual" storage layer (e.g. LVM) that provides a very large range of logical blocks (for example, 2^{64} blocks). The virtual partition storage is sparsely and dynamically populated with physical storage. Physical storage may be striped across multiple devices for improved performance and high availability.

Each storage partition is assigned to a partition server (PS), which controls access to its local file system resources. Therefore, contention for system resources occurs on a per partition basis and is controlled by the partition server. Client operations can be classified into two categories according to whether or not they affect system resources. They are referred to as "writes" and "reads" respectively. For example, in the case of the Network File System (NFSv2) [8], client requests can be classified as follows:

"reads": getattr, lookup, readir, read, realink, statfs, mount.

"writes": mkdir, link symlink, rename, remove, rmdir, setattr, write.

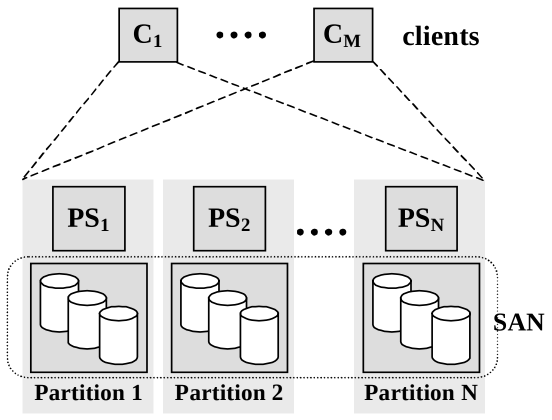

Figure 1. DiFFS architecture.

“Reads” are performed by the clients, possibly bypassing the partition servers, as described in detail in section 2.2. However, “writes” must be coordinated by the appropriate partition server. For example, when a file that resides on partition 1 is written, the client operation is forwarded to PS1, which allocates (from the partition resources) any new blocks that may be appended to the file. In this way, contention over system resources is restricted to a single partition and is resolved by a single point of control—the partition server—avoiding the use of a Distributed Lock Manager (DLM). DLMs are known for their complex design and inherent scalability problems [5].

The partition is the building block of a DiFFS system. The introduction of new partitions extends the capacity of the system without increasing the cost for contention control.

Partitioning policies. Initial assignment of files and directories to a specific partition is done at creation time, according to some “partitioning policy”. The policy can be implemented in a number of ways. For example, it can be implemented in a proxy that resides on the client, or in an application-level content switch on the network. The location of the object is then recorded in a DiFFS directory that references the object (see ”cross-partition references”).

The actual policy employed in the system is orthogonal to the proposed architecture. Examples of policies include:

- Files distributed amongst partitions according to some deterministic algorithm, which guarantees that all partitions accommodate a similar number of files, or that they have similar utilization of resources.
- Files distributed according to their type. Different partitions may use different local physical file systems that are fine tuned for different types of files, e.g. small, files, directories, video files, etc.

The only restriction for the partitioning policy is that a single file must be confined to only one partition. This is required to simplify the management of system resources and avoid consistency problems for “write” operations that might otherwise span more than one partition.

Cross-partition references. Files and directories of the global file system may reside on any partition. Thus, the namespace of DiFFS has to handle cross-partition references. Typically, in a file system, directories are special files that are used to maintain references to objects (e.g. files and directories). A directory file is a list of entries, each one containing at least the inode number (*inode#*) and *name* of the corresponding object.

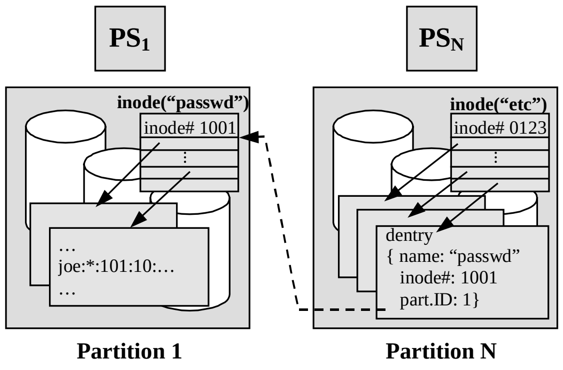

Figure 2. Example of cross-partition reference: an entry in directory */etc* references file */etc/passwd* which resides in a different partition.

In DiFFS, the inode# may refer to a file that does not reside in the same partition (physical file system) as the parent directory. To make the reference to an object unique in the system, the inode# in directory entries is augmented with the ID of the partition where the object resides (see Figure 2). The latter requires extensions to the directory entry structure. One possible implementation, which is independent of the actual physical file system, is to use normal “data files” in the file system to implement DiFFS directories. In the example of Figure 2, a DiFFS-space directory “/etc” is implemented as a regular file in the underlying file system of partition N. The contents of this file are DiFFS-specific entries referencing files that may reside in various partitions.

Implementing directories as regular files provides additional flexibility. Directories can be structured arbitrarily, allowing for example, a B-tree structure to be used to do efficient searches on large directories. Another possibility is to have a directory that is made up of several small files, where each file is leased independently, thus reducing directory file contention.

2.2. Access Protocols

In addition to traditional client-server protocols (e.g. NFS), DiFFS clients can use other protocols to access data on the partitions. The two examples described below take advantage of SAN.

Extended NFS (x-NFS). This protocol is employed when the clients are not aware of the details of the physical file system(s) on the partitions. The partition server acts as a metadata server. Client requests ("read" or "write") land on the server of the partition where the target file resides. The server interprets the file metadata (inode and indirect block contents) and returns, if necessary, a list of block IDs to the client. The client then directly accesses the target blocks on the storage (e.g. using iSCSI). The x-NFS functionality on the client is implemented either within the kernel (e.g. NFS client code) or as a specialized proxy.

Read-from-everywhere. When clients (kernel or proxy) are aware of the partitions' file system(s), they can interpret metadata without a metadata server. In these cases, "read" operations can be performed directly by the client, bypassing the partition server. This is because "reads" do not affect the file system resources and thus such operations need not be serialized via the partition server. All "write" operations must still be coordinated by the partition servers, using either a traditional client-server protocol or a protocol such as x-NFS.

The details of access protocols are outside the scope of this paper. The point to stress here is that access protocols are orthogonal to the DiFFS architecture and independent of the actual namespace management implementation. Moreover, multiple access protocols may be used concurrently in the system.

2.3. Caching and cache consistency

Distributed file systems traditionally use caching to improve performance. This raises the issue of cache consistency. Cache consistency in DiFFS is supported by means of "leases", as in NQNFS [9] and NFSv4 [10]. A lease is a promise from the server that the client can cache a specific object (for a limited time) without conflict. A lease must be renewed by the client, if it continues to cache the object. Leases provide a simple mechanism for the recovery of the server's state related to client caches—a recovering server waits for a time period that guarantees all outstanding leases have expired.

DiFFS supports two types of leases: Read and Write. The partition server is the lease server for files and directories residing in its partition. It is outside the scope of this paper to describe the details of the leasing protocol. It should be noted however that in DiFFS, leasing is used to guarantee cache consistency not only of cached data but also for cached directory contents and, in the case of x-NFS, for cached block lists.

2.4. File handles

A "lookup" operation is performed as depicted in Figure 3. The execution is described in a way that is independent of the access protocol used. For example, a lookup for file "/etc/passwd" is performed in three stages: (1) retrieve the file handle for the root inode—this information is typically stored on every partition, in this case it is retrieved from partition 1; (2) read the contents of directory "/" and construct the file handle for directory "etc"; (3) read the contents of "etc" and construct the file handle for the file "passwd".

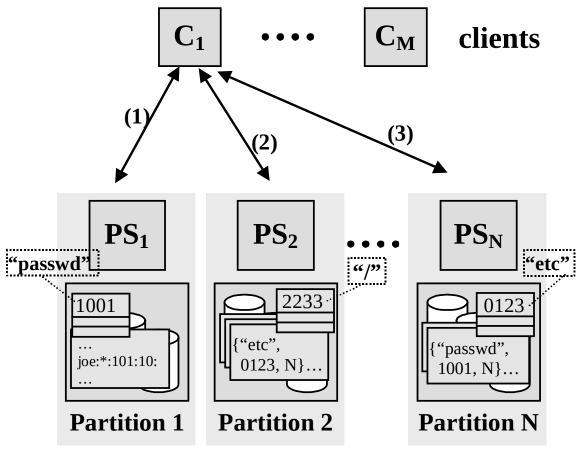

Figure 3. Execution of *lookup(/etc/passwd)* in DiFFS.

A file handle consists of the inode# and the partition ID of the file or directory as well as a generation number that makes the file handle unique in the system (even if the inode is reused in the future for another object). As described in the example, the file handle for a target file is obtained by first constructing the file handle for every directory in the path to the file. This is typical of existing systems, such as NFS v2 and v3. File handles are cached on the client (the DNLC in the case of NFS). Such caches have extremely high hit ratios (more than 90% [8]), improving considerably the performance of lookups. The only difference in the case of DiFFS is that the partition ID of the file must be included in addition to any other information in the file handle.

3. Advanced Issues

3.1. Fault-tolerance

The design of DiFFS is based on the assumption that servers may crash but they do not exhibit any malicious (Byzantine) failures. Messages can be lost due to host crashes or network partitioning. However, the communication protocol guarantees in-order delivery of messages between non-faulty hosts (e.g. TCP). The design goal for DiFFS is to provide fast recovery, while guaranteeing the integrity of the distributed file system.

The integrity of local file systems in the partitions is a traditional file systems issue. There exist well-known solutions such as recovery procedures (*fsck*), *soft updates* and *journaling*. The main challenge in DiFFS is the robustness of operations that span more than one partition. These are operations that affect the DiFFS namespace, namely (in NFS terms): *create*, *mkdir*, *link*, *remove*, *rmdir* and *rename*.

Existing systems that follow a partition-based approach suggest the use of transactional semantics (for example 2-phase-commit protocol) for the execution of such operations. These protocols are expensive and affect the performance of operations in the failure-free case. A more light-weight approach is used in DiFFS. By imposing a strict order on the execution of such operations, we can guarantee that all possible inconsistencies are reduced to an instance of “orphan” files. An orphan is a file that physically exists in some partition but is not referenced from any point in the DiFFS namespace. Consider the removal of file “/etc/passwd” (Figure 3). The client request lands on the partition where the parent directory resides (PS_N). The entry for “passwd” is removed from the directory and a message is sent to PS1 to remove the actual file. This message may be lost due to host failures or network partitioning. The only possible inconsistency is for the file to exist but not to be referenced from any directory.

The required execution order for cross-partition operations can be abstracted to the following three steps:

1. Remove reference from the namespace, if necessary.
2. Perform changes of the target object, if any.
3. Insert reference in the namespace, if necessary.

So, the problem of namespace integrity is reduced to garbage collection of orphan files. Global garbage collection algorithms may be both complex and impractical for the scale of DiFFS. Instead, we propose the use of algorithms that perform garbage collection based on local information. This is achieved by using *intention logs* [11] on the partition where each operation is initiated. The log is used, in the case of failure, to reconfirm or undo (as appropriate) the results of operations with unknown outcome. The algorithms take into consideration potential conflicts due to concurrent operations that affect the same objects or namespace entries.

There are two important points to be noted here. First, DiFFS provides the same failure semantics to clients as traditional client-server systems, (e.g. NFS). Second, distributed recovery is performed off-line; it does not affect the performance of failure-free operation and does not block normal operation in the presence of failures.

3.2. File-level Migration

File migration is essential for the scalability of DiFFS. The goal is to perform load balancing by moving objects from “hot spots” to less loaded partitions. Deciding which objects to migrate, when and where is a policy issue performed by a management tool. The entity that performs the migration obtains a Write-lease for the target object and copies it over to its new location. Using protocols such as x-NFS minimizes the copying overhead on the saturated partitions. The integrity of the DiFFS namespace must be maintained when migration is performed. There are two issues to consider:

- Updating affected file handles on the clients.
- Updating references in the DiFFS namespace.

Both problems are addressed by keeping a forward pointer in the original location of the object, which indicates the new partition ID and inode# of the object. This information can be kept, for instance, in the original inode of the object.

When a client request lands on the original location of the object, two things happen. First, the parent directory of the object that was used to obtain that file handle is updated (if it is in another partition, a message is sent); the corresponding entry is changed to reference the new location of the object. Second, an updated file handle with the new location of the object is sent back to the client.

One issue is when to garbage-collect the forward pointer (reuse the inode). This can be done as soon as two conditions are satisfied: 1) there are no references in the DiFFS namespace to the old position of the object (link count of the original inode is 0), *and* 2) there are no out-of-date file handles at clients.

To satisfy (2), clients are required to periodically revalidate any cached file handles, even if they do not use them. However, the entry in the parent directory of a migrated object is updated only when that directory is used to lookup a file handle for the object. There is no guarantee that all the references (hard links) to an object are eventually used to access it. Thus, there is no upper bound on the time required to keep the forward pointer. We are currently looking into solutions for this problem.

4. Related work

The distributed storage market is currently dominated by Network Attached Storage (NAS) systems, where access to storage is provided via NFS front-ends. Due to the inherent scalability limits of NAS systems, several industrial projects are currently investigating ways of aggregating multiple NAS systems under a single namespace [12]. These solutions do not improve performance (data is still copied through server memory) and they lack robustness and flexibility (sub-trees of the global file system are assigned statically to specific servers).

Ideally, SAN storage should be accessible directly by all clients in the system. The requirement is to provide coordinated access to the storage pool. For this purpose, a number of research systems have extended the idea of cluster file systems [13], which provide access to global storage resources through a cluster of servers. Processing power is added by introducing servers to the cluster. All of these systems use some type of DLM among the cluster servers. GFS [14] is a system designed for Linux that uses a proprietary physical file system. It is based on nonstandardized low-level locks implemented by the SCSI devices to achieve efficient distributed locking. Frangipani [5] introduced one of the most scalable DLM solutions in the literature. System resources are partitioned into logical volumes [15] and there is one DLM server dedicated to each volume. Both "reads" and "writes" are performed from every server and require coordination in the cluster. Frangipani's DLM is a complex service, which took three iterations to reach the final two-level design. All cluster file systems depend on their own proprietary physical file system.

DiFFS resembles Frangipani in its partitioning of the storage resources for improving contention control. However, it has certain advantages, in terms of robustness and performance. DiFFS facilitates the use of any physical file system in the partitions. So, files can be stored in partitions fine-tuned for their type. "Reads" can be performed from any node in the system, while "writes" are always coordinated by the server of the partition. Thus, contention control is significantly simplified, without sacrificing performance. Recovery does not require expensive distributed protocols (failure detectors and distributed agreement) and can be performed in the background during normal operation.

The system closest to DiFFS, in terms of design principles, is Slice [16], an ongoing research project at Duke University. Slice also suggests the distribution of a global file system across multiple partitions. Slice's partitioning mechanism (small vs. large files and a deterministic distribution within each class of files) is implemented in the so-called µproxies—modules that forward client operations to the right partition (at IP level). Due to the state (distribution tables) that is kept in the µproxies, Slice's reconfiguration is coarse grained. Furthermore, Slice assumes object-oriented storage to simplify resource allocation, whereas DiFFS is built on top of block-oriented SAN technology.

5. Conclusions

This paper introduces DiFFS, a scalable, flexible and robust distributed file system that leverages the latest storage technology. By taking a partitioning approach, DiFFS avoids the inherent scalability limits and complexity of traditional cluster file systems imposed by DLM. By layering over different physical file systems, DiFFS affords itself immense flexibility in serving diverse contents hosted by an IDC. Furthermore, it leverages SAN technology by accommodating protocols that allow clients to directly access SAN storage devices, and thereby ensuring high performance.

DiFFS addresses robustness on two fronts. First, cross-partition protocols are carefully designed to guarantee that the global namespace is never corrupted. This reduces the problem to off-line, local garbage collection and physical file system recovery, which is attained by existing technologies such as journaling file systems and soft updates. DiFFS does not require any complex global recovery algorithms. Second, the architecture encodes cross-partition references directly in the DiFFS directory structure. It is therefore possible to construct proactive file migration protocols to tackle load imbalance.

It is an open research issue to investigate how the architecture scales in wide-area systems (for example, cross-IDC configurations). File migration and replication are of particular interest in that context. Security is a major issue that has not been considered in depth yet. The current prototyping is based on NFS protocol implementations. It is a future work item to investigate how the architecture can accommodate other file access protocols. The design of DiFFS is work in progress. We are currently in the prototyping phase and expect to have publishable performance results within the next few months.

Acknowledgements

Extensive discussions with Jeff Chase helped formulate the ideas presented in this paper. Lisa Liu has participated in the architecture design. Dejan Milojicic, John Wilkes and other colleagues in HP Labs have provided valuable input.

References

1. Alster, N. and L. Hawkins, *Trouble in Store for Data-Storage King?,* in *Business Week*. 2000.
2. Burns, R., R. Rees, and D. Long. *Safe Caching in a Distributed File System for Network Attached Storage*. In *International Parallel and Distributed Processing Symposium*, Cancun, Mexico, May, 2000.
3. *Tivoli Storage Manager*, .2001, IBM Tivoli.
4. *EMC Celerra HighRoad File System Software*, 2001.
5. Thekkath, C., T. Mann, and E. Lee. *Frangipani: A Scalable Distributed File System*. In *16th ACM Symposium on Operating Systems Principles (SOSP)*, Saint-Malo, France, 1997.
6. Anderson, T., *et al.*, *Serverless Network File Systems.* ACM Transactions on Computer Systems, 1996, 14(1): pp. 41-79.
7. Howard, J., *et al.*, *Scale and Performance in a Distributed File System.* ACM Transactions on Computer Systems, 1988, 6(1): pp. 51-81.
8. Callaghan, B., *NFS Illustrated*. Addison-Wesley Professional Computing Series, Adison-Wesley, 2000.
9. Macklem, R. *Not Quite NFS, Soft Cache Consistency for NFS*. In *Winter 1994 Usenix Conference*, San Francisco, CA, USA, January 1994.
10. Shepler, S., *et al.*, *NFS version 4 Protocol*, 2000.
11. Seltzer, M., *et al. Journalling versus Soft Updates: Asynchronous meta-data protection in file systems*. In *2000 USENIX Annual Technical Conference*, San Diego, California, USA, June 18-23, 2000.
12. Zhang, Z. and A. Bhide, *NAS Aggregation*, . 2001, Hewlett-Packard Labs, Palo Alto.

13. Kronenberg, N., H. Levy, and W. Stecker, *VAXClusters: A closely-coupled distributed system.* ACM Tansactions on Computer Systems, 1986, 4(2): pp. 130-146.

14. Preslan, K., *et al. A 64-bit, Shared Disk File System for Linux*. In *16th IEEE Mass Storage Systems Symposium*, San Diego, CA, USA, 1999.

15. Lee, E. and C. Thekkath. *Petal: Distributed Virtual Disks*. In *ASPLOS VII*, MA, USA, 1996.

16. Anderson, D., J. Chase, and A. Vadhat. *Interposed Request Routing for Scalable Network Storage*. In *Usenix OSDI*, San Diego, CA, USA, 2000.