An Architecture for Scalable and Manageable File Services

C. Karamanolis, L. Liu, M. Mahalingam, D. Muntz, Z. Zhang
Internet Systems and Storage Laboratory
HP Laboratories Palo Alto
HPL-2001-173
July 12th , 2001*

E-mail: {christos, lisaliu1, mmallik, dmuntz, zzhang} @hpl.hp.com

distributed file service, namespace, resource aggregation, manageability

Monolithic file servers are limited by the power of an individual system. Cluster file servers are limited by resource sharing and recovery issues as the number of cluster nodes increases. D*i*FFS is a file service architecture that allows system resources to be added (or removed) dynamically, e.g., storage and processors. Resources are partitioned in such a way that contention is avoided, while maintaining a single namespace. Resources may be heterogeneous, and geographically dispersed.

This architecture has several advantages. A file's *physical location* is decoupled from its *location in the namespace*. This decoupling enables a powerful and flexible mechanism for the placement of file system objects. For example, different types of files, e.g., text or video, may reside anywhere in the namespace while being hosted by servers best suited to handling their content type. D*i*FFS also provides lightweight protocols for online dynamic reconfiguration (*volume reassignment and object migration*) to address fluctuating demand and potentially mobile file system entities. A D*i*FFS prototype has been implemented in Linux. Performance results indicate that the architecture achieves its flexibility and scalability goals without sacrificing performance.

An Architecture for Scalable and Manageable File Services

C. Karamanolis, L. Liu, M. Mahalingam, D. Muntz, Z. Zhang [†]

Hewlett-Packard Laboratories
1501 Page Mill Rd, Palo Alto, CA 94304, USA
{christos,lisaliu1,mmallik,dmuntz,zzhang}@hpl.hp.com

Abstract

Monolithic file servers are limited by the power of an individual system. Cluster file servers are limited by resource sharing and recovery issues as the number of cluster nodes increases. *DiFFS* is a file service architecture that allows system resources to be added (or removed) dynamically, e.g., storage and processors. Resources are partitioned in such a way that contention is avoided, while maintaining a single namespace. Resources may be heterogeneous, and geographically dispersed.

This architecture has several advantages. A file's *physical location* is decoupled from its *location in the namespace*. This decoupling enables a powerful and flexible mechanism for the placement of file system objects. For example, different types of files, e.g., text or video, may reside anywhere in the namespace while being hosted by servers best suited to handling their content type. *DiFFS* also provides lightweight protocols for online dynamic reconfiguration (*volume reassignment* and *object migration*) to address fluctuating demand and potentially mobile file system entities. A *DiFFS* prototype has been implemented in Linux. Performance results indicate that the architecture achieves its flexibility and scalability goals without sacrificing performance.

1. Introduction

A recent study by researchers at the University of California at Berkeley [23] reports that there has been approximately 10 exabytes (10^{18} bytes) of information produced during all of human history, in various forms: e.g., books and videos—from cave drawings to computer files. In addition, up to 2 exabytes of new data is produced each year. The majority of this new data is stored in digital form. "*Not only is digital information production the largest in total, it is also the most rapidly growing*." The study provides two main reasons for this explosive growth of digital data. First, "democratization of data"—a vast amount of unique information is created and stored by individuals. Second, the dropping cost of magnetic storage; as of early 2001 a gigabyte of storage costs less than $10 and it is predicted this cost will drop to $1 by 2005. The study concludes, "…*better tools are desperately needed if we are to take full advantage of the ever-increasing supply of information*."

A big part of the world's data is currently held in file systems. The "file" abstraction will likely continue to be a prevalent way for perceiving, storing and managing data. To meet increasing demand for file storage, file services must be both highly scalable *and* highly manageable. Many systems have attempted to address scalability in different contexts [8, 14, 12, 7, 20]. Only recently has the issue of manageability become a key factor (perhaps the primary factor) in large system design. Various studies show the expense of providing large computing services lies mostly in the cost of people to manage these services, rather than, for example, hardware and software [6, 21].

Research on cluster file systems has attempted to address scalability while keeping the management of large, distributed file systems simple [19, 33, 30, 4]. A cluster of servers provides

[†] Names in alphabetical order.

access to a (large) pool of resources. Processing power and/or storage resources can be added or removed with minimal human administration, in a way that is completely transparent to clients. However, cluster file systems do not scale sufficiently for pools of resources that span large SANs or multiple Data Centers. Their scale is limited by resource sharing issues, requiring distributed lock management.

DiFFS is a file service architecture designed to serve Petabytes of data, distributed across heterogeneous, possibly geographically dispersed resources, accessed by thousands of clients (e.g., web and application servers), while minimizing the need for system management. DiFFS allows resources to be added incrementally, e.g., additional storage, processing power, etc. Resources are partitioned in such a way that resource contention is avoided, while maintaining a unified namespace for the overall file service [28]. The consistency of the namespace is guaranteed by lightweight, fault-tolerant protocols that allow metadata operations to scale well .

The DiFFS architecture extends the state-of-the-art in the following ways:

- Provides a simple mechanism for aggregating diverse, distributed system resources and presenting a single uniform file space (file virtualization) [27].
- Introduces a novel design of distributed namespaces, where the physical location of files is independent of their position in the namespace hierarchy. This facilitates policy-driven allocation of files to resources, and transparent support for various file types arbitrarily dispersed throughout the namespace.
- Proposes a loose coupling of storage to computation resources, which facilitates dynamic and efficient system reconfiguration to meet changing demands. The proposed mechanisms facilitate automated load balancing and resource utilization.

In this paper, a first prototype of the architecture is evaluated using a number of synthetic benchmarks. File virtualization and the goals of scalability and flexibility are achieved without sacrificing performance.

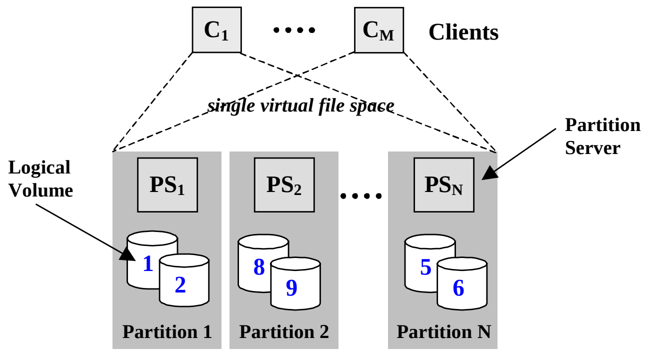


Figure 1. The partitioning approach of DiFFS.

2. Architecture overview

Ideally, a distributed file service should scale arbitrarily, with new resources adding to system capabilities without diminishing marginal returns. Also, the service should have simple mechanisms for aggregation and (re)configuration of system resources. To meet these goals, DiFFS takes a partitioning approach to resource sharing, by dividing resources into *storage partitions* (Figure 1). Resources in a partition are controlled exclusively by a single *partition server*. Storage within a partition is divided into volumes. Each volume contains a single physical file system.

Another basic design principle of the architecture is to decouple the names of file and directory objects (location in some namespace) from the objects' actual physical location (volume, partition). The aim is to allow for maximum flexibility of placing objects across the resources available in the system, while presenting a virtual file space to the clients. Files and directories can be placed arbitrarily on any volume and in any partition in the system. Namespaces in DiFFS are therefore distributed. The consistency of distributed metadata is guaranteed by fault-tolerant protocols, as discussed in section 3. The integrity of the objects in the volumes is guaranteed by the native file system of the volumes (journaling, fsck, etc).

Decoupling the physical location of objects from their position in the namespace offers several advantages:

- Objects can be placed according to "placement policies" addressing issues such as load balancing among servers, storage resource utilization, access patterns and content type, to mention just a few.
- Both data and metadata operations are completely distributed and scalable.
- System resources can be reconfigured dynamically, without requiring changes in the namespace. Servers can be added or removed and volumes can be moved dynamically. Objects can be aggregated into volumes and volumes into partitions. Volumes in DiFFS are simply "containers of objects" and are not linked to a location in the system's namespace, as is the case for AFS [15]. Instead, they are used as object aggregators and form a unit of storage resource reconfiguration.

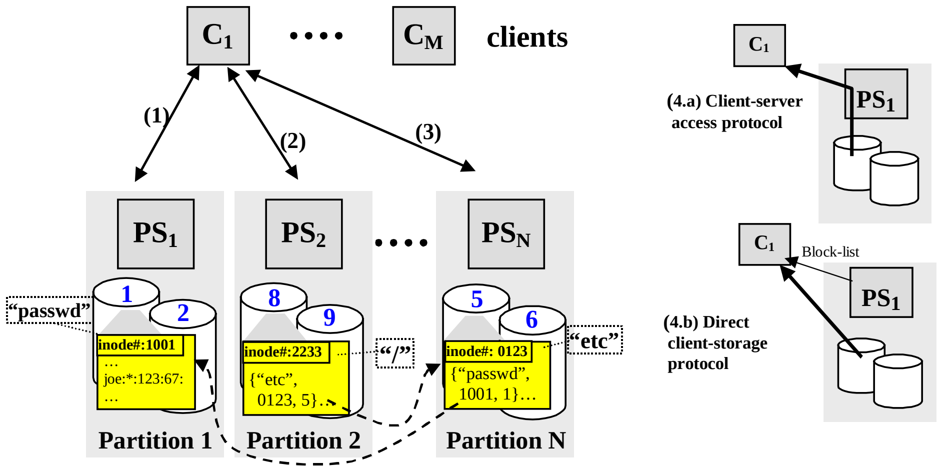


Figure 2. Example of lookup and file access.

The basic principles of the architecture are illustrated with the scenario of Figure 2—a client accessing the file "/etc/passwd." The two directories, "/" and "/etc," are hosted in logical volumes 8 and 5 respectively, and file "passwd" in logical volume 5. They all reside in separate partitions. In this case, the lookup operation, required to retrieve the file handle for "passwd," is performed in three stages:

(1) retrieve the file handle for the root directory–this information is typically replicated in every partition; in this case it is retrieved from volume 1, partition 1;
(2) using the latter file handle, read the contents of "/" (in volume 8, partition 2) and construct the file handle for directory "etc";
(3) read the contents of "etc" to construct the file handle for "passwd".

Having resolved the name “/etc/passwd,” the resulting file handle is used by the client to access the file “passwd,” on volume 1, for read or write. The DiFFS architecture is orthogonal to data access protocols, and can use either traditional client-server style (4.a) or direct client-storage protocol (4.b) where the client is informed about the location of the data.

A file handle consists of the inode number (and generation number) of the object, the volume where the object resides and the communication endpoint of the corresponding partition server (IP address and port). Pathname resolution is performed in stages, one for each element of the path. This resembles NFS v2. One difference is that, in the case of DiFFS, every element in the path may have to access a different server in the system.

Cache consistency in DiFFS is supported by means of leases [24, 10, 32]. The client must renew a lease, if it continues to cache a leased object. The partition server is the lease server for files and directories residing in its partition. 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, before issuing new leases. DiFFS supports two types of leases: Read and Write. The protocols resemble those described in [24]. 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.

3. Distributed namespaces

DiFFS aggregates multiple storage resources and presents a single, uniform file space to clients. Objects can reside on any partition, irrespectively of their position in the namespace (unlike NFS and AFS, where the objects of an entire namespace sub-tree are located behind one server). In existing systems, directories are special files used to maintain references to objects. A *directory file* is a list of entries, each one containing at least the *inode number* and *name* of the corresponding object. In the case of DiFFS, the entry must also contain the object’s physical location. An entry therefore consists of three fields:

1. *Volume#*: the id of the volume where the object resides; this id is unique across the system and is mapped to a particular partition server, as described in section 5.1.
2. *Inode#*: the inode number of the object in its resident volume.
3. *Generation#*: the inode generation number.

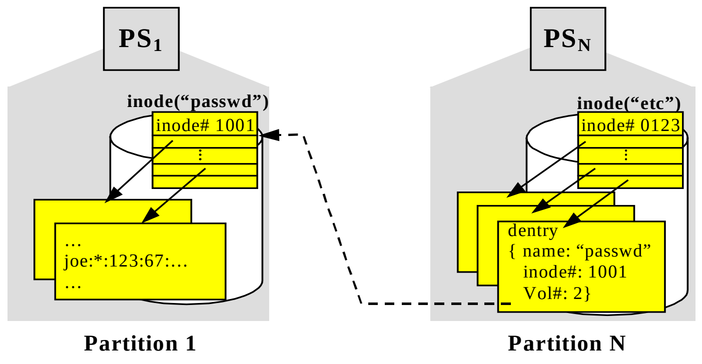


Figure 3. Example of object references in DiFFS: an entry in directory /etc references the file /etc/passwd, which resides in a different location.

Extending references to include the volume number of the object requires modifications to the structure of directory entries. “DiFFS-space” directories are implemented as regular data files in the underlying file system. This provides independence from the native file system directory structure *and* the ability to create *customized directories*. For example, one can use B-trees for efficient searches on large flat namespaces, or have directories that are split into several small files, or use encryption to achieve privacy of directory contents. In the example of Figure 3, a DiFFS-space directory "/etc" is implemented as a regular file in the native file system of a volume in partition N. The contents of this file are DiFFS-specific directory entries referencing files that may reside at any location, e.g., the file "passwd" in a volume of partition 1. The drawback of DiFFS-space directories is that they do not take advantage of optimized directory implementations of the underlying physical file systems.

In DiFFS, namespace metadata operations, such as *create*, *remove*, *link*/*unlink*, and *rename*, are, in the general case, distributed. This introduces the problem of maintaining a consistent (robust) namespace over a collection of distributed objects during normal operation, and in the presence of communication or host failures. DiFFS is not the first system to introduce distributed namespaces. Slice [7] and Archipelago [16] are two systems that also distribute metadata across multiple sites. Both of these systems resort to 2-phase commit implementations of metadata operations to guarantee fault-tolerance. However, 2-phase commit is expensive. It imposes high overhead to failure-free execution due to synchronous logging; it locks system resources across all involved sites for the duration of the multi-phase protocol execution; and it follows a conservative approach for recovery from failure—operations are aborted. Given that metadata operations comprise the majority of operations in typical workloads [31], the efficiency of the corresponding protocols is important to overall performance.

For this reason, a set of lightweight protocols for metadata operations has been proposed, in the context of DiFFS. Details of the protocols are published in [34]. Overhead to failure-free operations is minimized by reducing the number of synchronous I/O's in the critical path of operations. Resource locking is avoided; intention logging and serialization of operations on the participating sites suffices. Aggressive recovery techniques are used to re-play incomplete operations (transparently to the application), in most failure scenarios. In [34], the protocols are compared with typical 2-phase commit implementations and are shown to be superior in all critical performance factors: communication round-trips, synchronous I/O, and operation concurrency. The price for these benefits is additional data structures. Each object is annotated with "back pointers" to all its parent directories in the namespace. Distributed metadata operations do *not* affect the scalability of DiFFS, as more servers are added to the system; they add a constant performance factor. All operations involve, in the worst case, two servers (with the exception of *rename* that involves three)—site of the parent directory and the site of the object.

4. Data placement

Initial assignment of files and directories to a specific volume and partition is done at creation time, according to a "placement policy." The *create* request is sent to the server that hosts the parent directory of the new object. The server applies some placement policy to decide where (partition, volume) the object should reside. The request is then forwarded to the (potentially remote) site of the new object and an entry is created in the parent directory, as shown by the message diagram of Figure 4.

The actual placement policy employed by the servers is orthogonal to the architecture. Examples of policies include the following.

- Files are distributed amongst partitions in a way that guarantees "even" distribution of load. For example, all partitions accommodate a similar number of files (or volume of data), or all servers have comparable processing load.

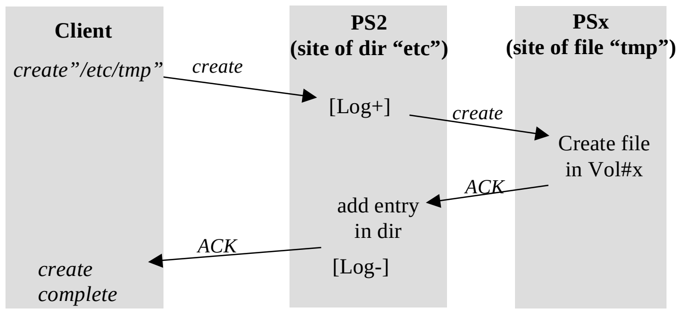


Figure 4. Object placement at creation time.

Such policies can potentially lead to some performance overhead due to distributed metadata operations (create, lookup, remove, etc). A "mkdir affinity" policy can provide better locality for metadata operations:

- Files are always created in the same volume as the parent directory. New directories are created (with probability *p*) on a volume and partition different than that of the parent directory [7].

DiFFS is implemented as a kernel module independent of file system implementations; it assumes that the file system supports the *v-node* interface [18]. A partition server may host multiple volumes, each one managed by a different native file system. Therefore, content type is another important factor for placement decisions.

- Objects are placed to volumes with the most appropriate file systems for their content type, e.g., small files, directories, video files, etc.

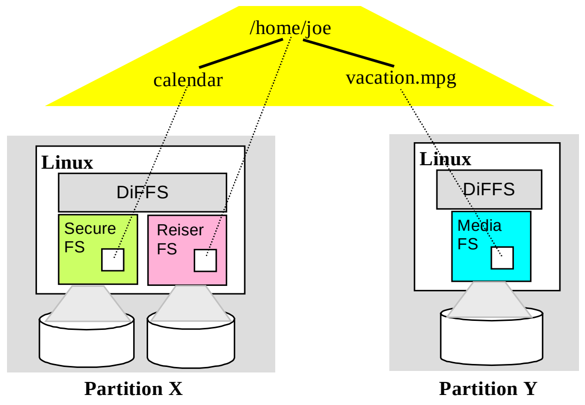


Figure 5. File placement according to content type.

Figure 5 illustrates an example of a user's home directory and how objects in that namespace are placed to different volumes according to their content type. No changes to the application are required. Placement decisions are made locally on the servers, based on a global view of the system's configuration (partitions and volumes) they maintain—see section 5.1.

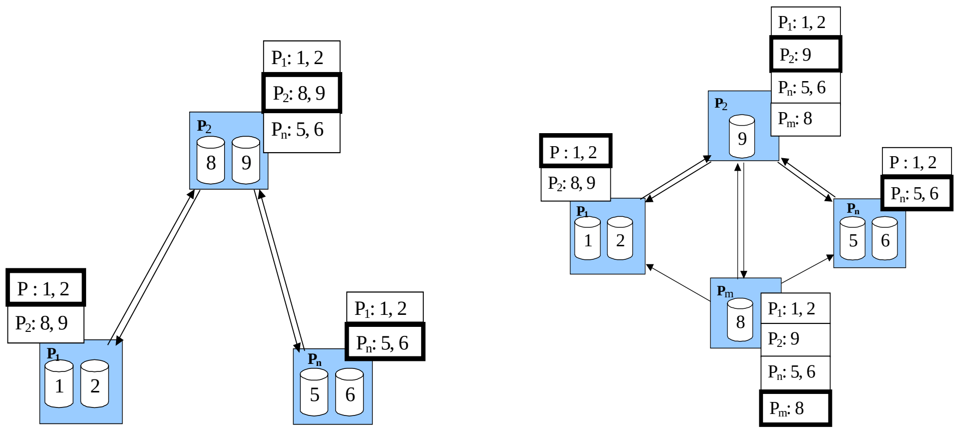
5. Resource reconfiguration

System resources, both servers and storage, may be added or removed dynamically in a running DiFFS system, to address changing workloads and fluctuating demand. Resource (re)allocation is performed at the granularity of volumes. This section describes two mechanisms necessary for resource reconfiguration. These are protocols for (re)assigning volumes to partition servers and protocols for reconfiguring volumes by moving individual objects across volumes. These mechanisms have been designed to facilitate the development of automated tools to perform on-line management of the system.

5.1 Volume reassignment

In order to access a file, the client retrieves the hosting volume and partition from the file handle (returned by a lookup operation). A distributed algorithm is executed amongst the servers to maintain a global, consistent view of the volume-to-partition mapping in the system.

Each server maintains a *volume table* with the id's of the volumes it hosts. The table is assigned a *version number,* which is incremented every time its contents change. Volume tables of other servers are also locally cached forming the server's *neighborhood* view. Arrows in Figure 6 depict "knowledge of another server's mapping"; thick-bordered labels depict local tables. The cached tables are kept loosely synchronized with the master copies on their host servers. When a server receives a lookup request, it returns a file handle that contains the object's volume id and the corresponding server according to its view. In the example of Figures 2 and 6(a), the lookup for file "passwd" in directory "etc" lands on partition server PS_n, which responds with a file handle containing the volume id (#1) and the address of the partition server (PS_1). The identity of the server is just a hint to the client, because volumes may move between servers.



(a) Snapshot of mapping state (b) Mapping state after reconfiguration

Figure 6. Volume-to-partition mapping.

Volume re-assignment between servers offers the means to perform reconfiguration in the system. In the example of Figure 6, a new server PS_m is added to the system and volume #8 is reassigned from PS_2 to PS_m. As a result of the peer-peer protocol between PS_2 and PS_m, their mutually cached volume tables are up-to-date. However, PS_1 and PS_n have stale tables for PS_2 and no tables for PS_m. Stale tables are updated in two ways. First, cross-partition communications (e.g., distributed metadata operations) are piggybacked with the version number of the sender's local volume table. Thus, other servers find out about their stale copies and request the up-to-date version from the owner.

Second, a server may receive requests for objects in volumes it *used* to host but that now have moved. The server forwards those requests to the new location; note that the server "knows" where its old volumes have moved[1]. The file handle that parameterizes the request contains

[1] In theory, multiple forwarding hops may occur, if the volume has moved many times in the meanwhile. Eventually the current host server will be located. In practice, volume reassignments do not happen often.

information about the parent directory of the object. In DiFFS, this information is extended to fully identify the parent directory, including partition and volume id. This information is used to communicate an updated table of the object's volume location to the parent directory host. Even this message may have to be forwarded (possibly several hops) until the actual host of the parent directory is located and its table updated.

Volume reassignment can be implemented in two different ways depending on the physical proximity of the involved servers:

- When the two servers have access to the same pool of physical storage resources, e.g., a SAN, reassignment requires just re-mounting the volume between the servers. The original server puts all new requests to the volume on hold, unmounts the volume and sends a request to the new server to mount the volume. Upon receiving an ack from the new server, it updates its volume tables and forwards all pending requests for that volume to the new location.
- When the two servers do not share physical storage resources, physical volume migration is required, in addition to the above protocol. There are many (including commercial) systems that implement volume migration even across widely distributed sites [5, 3].

The current prototype of DiFFS implements the former protocol, but does not incorporate any mechanisms for copying volume contents between remote sites.

5.2 Object migration

In some situations, it is desirable to move individual objects (files or directories) between volumes. For example, the files that constitute the personal profile of a nomadic user may be migrated close to the physical location of access. Or a file that exceeds some size threshold may be moved from a "small files" volume to a "large files" volume. The protocol for object migration consists of six stages:

1. Put on hold any new requests for the object.
2. Create a replica of the object in the new volume.
3. Create forwarding information at the original volume (the original inode is re-used for this purpose).
4. Forward any pending requests to the new location.
5. Update the namespace references to reflect the new volume.
6. Garbage-collect the forwarding information when all namespace references have been updated and all the cached file handles of the object are revalidated.

Details of the protocol can be found in [25]. Any requests arriving at the original location between stages 4 and 6 are forwarded to the new location. An interesting part of the protocol is the way garbage collection of forwarding pointers is performed. The namespace references are updated by means of metadata kept together with the object; these are "back-pointers" to all the directories that contain references (hard links) to the object. Updating the namespace is not sufficient to garbage collect the forward pointer. We must also ensure that there are no cached file handles at the clients that reflect the old volume of the object[2]. For this reason, clients are required to revalidate any cached file handles periodically (every t_r sec). Thus, a partition server that maintains a forward pointer for a migrated object can safely remove the pointer af-

[2] Recall that the identity of the partition server in the file handle is just a hint to the client, but the volume# is necessary to locate the object in the system.

ter time t_r has elapsed (given that the namespace has also been successfully updated in the meantime). Alternative mechanisms are also under consideration [29].

6. Implementation

The DiFFS prototype is a proof-of-concept for the architecture, based on the Linux NFSv2 implementation. The prototype was started on Linux 2.2.14, with an NFSv2 base (Linux NFSv3 was not stable at that time). Subsequently, it was moved to the 2.4.2 release but remained on the NFSv2 base. Improvements in NFSv3 are orthogonal to DiFFS and some will eventually be integrated into the prototype.

The majority of the DiFFS implementation is in the NFS server code, with the following major components:

- DiFFS directory files: each directory entry contains not only conventional fields such as the name, length, inode number and generation number, but also the volume id, which uniquely identifies the physical storage where the object resides. Currently, the volume id contains the file handle of the native root directory of the volume. The DiFFS directory file can be based on a variety of data structures, such as a sequential list or a hash table. To improve performance of frequent operations such as lookup, the prototype uses a hash structure in the directory files. The hash is performed on the name of the file and the value directly taken from the Linux dentry, and the number of hash buckets is specified at the header of the directory file. Experiments reported in this paper use 128 hash buckets, each bucket is 4K bytes in size. Overflow buckets chained and are appended at the end of the file. Larger number of buckets will help to locate an entry faster, at the expense of a sparser directory file and incur more indirect block I/O access.
- DiFFS cross-partition operations: *lookup*, *create*, *remove* and *rename*. The protocols follow closely those specified in [34], but recovery is not currently implemented.
- A file placement policy module, which decides where new objects are to be located. The module implements a number of policies, including round-robin (RR) and affinity-round-robin (AR). RR spreads objects evenly across volumes in the system. AR always colocates a file with its parent directory, whereas directories are created using RR.

The unused byte-range in the NFSv2 file-handle is used to encode the object's volume id and server IP address. At the client side, the NFS protocol has been extended to extract this information from the file handle and direct subsequent requests for the object to the correct server. Lastly, we developed utility programs for analyzing the directory structure and changing allocation policies online.

7. Performance evaluation

In the test results below, ext2 is the underlying physical file system. Unless otherwise specified, the allocation policy is round-robin. The test configuration is based on HP Kayak PCs running Linux 2.4.2, connected by a 100Mb switch. All machines are Pentium-II/III, ranging in speed from 400-500MHz, with 256MB RAM, and fast-wide SCSI disks.

7.1 Base-line evaluation

The first set of experiments compare DiFFS to NFSv2, using two synthetic workloads with one client and one server. The goal of these experiments is to evaluate overhead introduced by the DiFFS architecture.

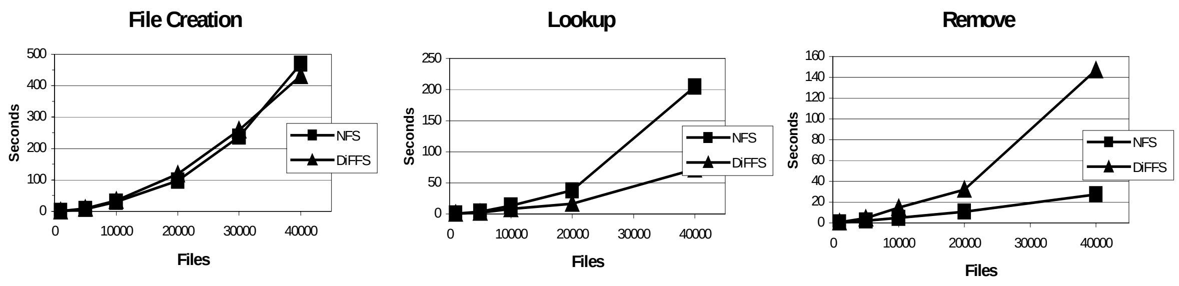


Figure 7. Comparing DiFFS and NFSv2.

The first workload is a version of the *lat_fs* benchmark [26]. This test performs *creates*, *lookups* and *deletes* for a given number of files. For these experiments, the number of files varies from 1K to 40K in a flat directory; results for *creation* and *lookup* are shown in Figure 7. The DiFFS *create* results are very similar to those of NFS. This is because the majority of the time is spent allocating space for the object (an equal cost for both systems). DiFFS introduces a penalty for doing directory I/O, but recovers most of this time due to the hash-structured directory files. Moreover, DiFFS *lookups* using hashed directories scale better than NFS lookups on top of ext2. The DiFFS *remove* results are quite poor relative to NFS because ext2's unlink routines simply mark the inode dirty and let the flush daemon take care of the delete asynchronously[3]; instead DiFFS has to do an actual high-level I/O to a directory file for each remove.

The second set of experiments performed compares the data path of DiFFS to NFS using *Iozone* [1]. Results from these experiments are virtually identical for NFS and DiFFS, because there is extremely little overhead in the DiFFS data path, relative to the original NFS code on which the prototype is based.

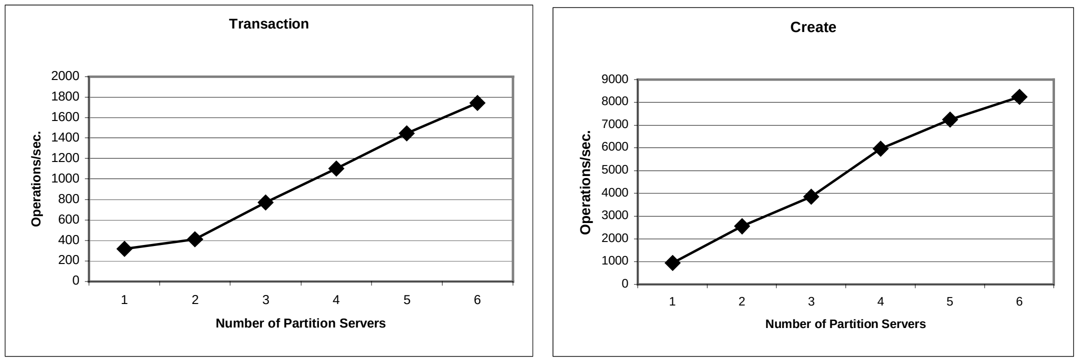


Figure 8. Postmark results: aggregate operations per second (total 12 client processes).

7.2 *Scalability of the architecture*

The scalability of DiFFS is evaluated using the industry-standard benchmark *Postmark* [17] on configurations with varying numbers of partition servers. Postmark simulates ISP workloads (mail/news servers), testing both data and metadata operations. Postmark operations include creation and deletion of files, and the transactions performed on them. The reported results in Figure 8 are based on two client machines, each running six Postmark client processes. This configuration delivers the approximate load needed to stress four partition servers before the client machines become overloaded (i.e., when the clients become the bottleneck). The parame-

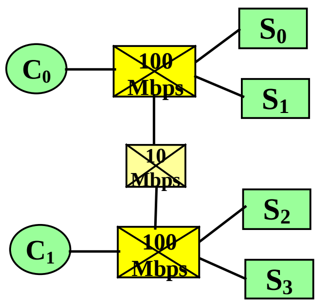
[3] Ext2 offers no consistency guarantee and performs deletion operations in memory.

ters used for this experiment were: 1000 files, 5 sub-directories, and 10,000 transactions. Each point in the graphs is the average value of one Postmark process across 12 runs. The allocation policy used for the experiments is “round-robin” (RR). Due to the relatively small size of the files (500-10K bytes), the metadata component of the transaction test (namely, opening/closing the files) may be significant.

The results of Figure 8 demonstrate that transactions (I/O operations) scale approximately linearly with the number of servers. Moreover, they confirm our initial hypothesis that distributed metadata operations in DiFFS do not impose a burden to scalability, even with an unfavorable allocation policy (round-robin). For example, creates scale also linearly with the number of servers. In the case of transactions, the marginal difference between the two first points of the graph is due to the distributed metadata component of the transactions introduced in the two-server case.

7.3 Flexibility of the architecture

One of the interesting aspects of DiFFS is the complete relaxation of object placement. In addition to placement policies outlined in section 4, performance characteristics of the infrastructure can also be used as a placement guideline. To demonstrate that, we performed an experiment simulating a wide-area deployment of DiFFS. We configured two “clusters” interconnected by a slow 10Mbps hub; in each cluster, there are one client and two partition servers connected by a 100Mbps switch. The ideal placement policy in this configuration would be “vicinity placement” (VP), where files accessed by a client are located within the same cluster.



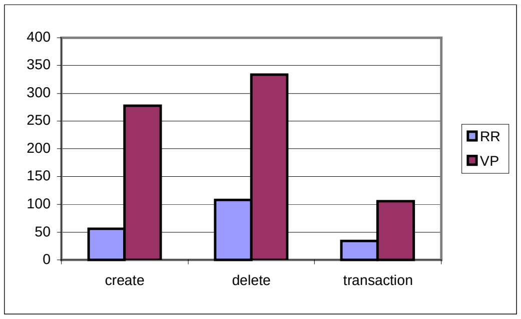


Figure 9: Vicinity placement and its performance (operations per sec per client).

The difference between the vicinity placement and the round-robin policy is demonstrated in Figure 9. The postmark transaction rate is more than 3 times better with the vicinity policy. While the vicinity information was hard-wired in the clients and servers for this experiment, we plan in the future to investigate ways to automate the decision-making.

8. Related work

A number of research efforts investigate techniques for building scalable file systems over resources interconnected by a network infrastructure. We classify these systems in two broad categories. The first is based on the idea of separating file managers (metadata management) from the block storage service. The first system to follow this model was Cambridge’s Universal File Server [9]. Subsequent systems extended this model by either allowing bulk I/O to bypass the file managers [11, 13] or completely separating the meta-data from the data access path [2, 10]. The principle behind these systems was to improve scalability by reducing the load of the server—data are no longer copied through the server. Even in these architectures, metadata servers are eventually saturated in a large system. The DiFFS architecture is complementary to these techniques. When storage resources reside on the SAN, clients can use a direct access protocol to retrieve metadata from the server but access the data directly from the storage device [2]. Moreover, DiFFS can be combined with object storage technologies, such us CMU's NASD [12], to provide a file virtualization service on top of object repositories.

The other group of scalable storage systems is based on the idea of clustering a number of servers to provide access to a pool of networked storage resources. Most of these systems follow the cluster file system approach, first introduced in VAXclusters [19]. Their principle is that the higher the load of the system the more servers (processing power) you add to the cluster [33, 30, 4]. All of these systems require some kind of distributed lock management (DLM) among the servers to coordinate access to the shared resources. DLM introduces new scalability issues. Frangipani proposed one of the most scalable DLM solutions in the literature [33]. System resources are partitioned into logical volumes [22] and there is one DLM server dedicated to each volume. This requires using two levels of virtualization: virtual disk and file system. DiFFS resembles Frangipani in its partitioning of the storage resources for improving contention control. However, DiFFS uses one level of virtualization allowing decisions for resource utilization and file placement to be made at the file service level. Also, cluster file systems, including Frangipani, depend on their own, proprietary physical file system. DiFFS is a protocol-level service and can leverage diverse file systems for optimal content placement and delivery. Nevertheless, DiFFS is complementary to cluster file systems—a partition can be implemented as a cluster file system and can be integrated into a broader file space.

Slice is a system with design goals close to those of DiFFS [7]. It also aggregates disparate storage and computation resources presenting one virtual file system to clients. Slice's file placement policies (small vs. large files and a deterministic distribution within each class of files) are implemented in *μproxies*—modules that forward client operations to the right partition, operating at the IP layer. In order to make placement decisions, *μproxies* have to maintain a view of the server membership in the system. In case of reconfiguration, the new membership information is diffused in the (possibly thousands) *μproxies* in a lazy fashion. As a result, resource reconfiguration in Slice is coarse-grained; also, file allocation is static for the duration of an object's life. DiFFS extends the traditional file system namespace metadata to achieve highly flexible and dynamic file placement and resource reconfiguration. However, this requires extensions (even if minor) to the client access protocol. Slice's μproxy idea could be used to transparently intercept client-service communication and redirect it to the appropriate partition server. In that case, μproxies will not need to maintain distribution tables; instead, they will interpret the contents of the (opaque to the client) file handles to retrieve the location of the server for each client request.

9. Conclusions

This paper introduces DiFFS, a file service architecture that aggregates diverse, distributed system resources (servers, storage). DiFFS servers cooperate to present a single virtual file space to clients. The service is designed as a kernel module independent of native file systems. Hierarchical namespaces are implemented by DiFFS-space directories; directory entries reference objects that may be located anywhere in the system. This facilitates policy-driven allocation of files to resources. For example, objects are stored to locations and native file systems that are best suited to the type of their content. Fault-tolerant protocols for distributed metadata operations have been designed to guarantee the consistency of DiFFS metadata even in the presence of server and/or communication failures. The protocols introduce minimal overhead to failure-free execution. A loose mapping of storage resources to servers facilitates cheap dynamic resource reconfiguration to address system evolution and fluctuating demand.

A prototype of the architecture has been developed, as an extension of the NFS protocol. Initial results indicate that the DiFFS layer does not introduce a considerable performance overhead for operation execution, compared to a typical NFS implementation. Measurements from an industry-standard benchmark (Postmark) demonstrate the scalability of DiFFS in two dimensions: distributed metadata operations and I/O intensive transactions. Even limited experimentation with different placement policies demonstrates large performance impact of policies for different types of workloads.

We are currently working on protocols for object replication for high availability and improved performance. We plan to first apply the resulting mechanisms to guarantee high availability of the DiFFS namespace. We would also like to gain further experience with the prototype under load and validate its scalability by testing it in larger configurations. In the future, we would like to: investigate further the idea of placement policies and device protocols for data placement in very large scale; study new ways, not necessarily deterministic, of naming and locating data in very large scale; investigate a security model for file services that span administrative domains. Last but not least, we would like to see ideas from DiFFS making their way into commercial storage products and promote the idea of the new “resources economy” model.

Acknowledgements

This work has benefited from discussions with many people including Jeff Chase, Erik Riedel and John Wilkes.

References

[1] “IOzone Filesystem Benchmark”, iozone.org.

[2] *Transoft FS,* , TransoftNetworks/HP: Santa Barbara, CA, USA.

[3] “Migrating Data Between NetApp Filers”, NetworkAppliance Corp., Sunnyvale, California, USA, TR-3018, 2000.

[4] “Veritas Cluster File System (CFS)”, Veritas Corp., Mountain View, California, USA, 2000.

[5] “Veritas Volume Replicator”, Veritas Corp., Mountain View, California, USA, 2000.

[6] Alvarez, G., *et al.* "Storage Systems Management (Tutorial)", in *Proc. of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS 2000)*. Santa Clara, CA, USA, June, 2000.

[7] Anderson, D., Chase, J., and Vadhat, A. "Interposed Request Routing for Scalable Network Storage", in *Proc. of the Usenix OSDI*. San Diego, CA, USA. USENIX, 2000.

[8] Anderson, T., *et al.*, *Serverless Network File Systems.* ACM Transactions on Computer Systems, Vol. **14**(1): pp. 41-79, 1996.

[9] Birrel, A. and Needham, R., *A Universal File Server.* IEEE Transactions on Software Engineering, Vol. **6**(5), 1980.

[10] Burns, R., Rees, R., and Long, D. "Safe Caching in a Distributed File System for Network Attached Storage", in *Proc. of the International Parallel and Distributed Processing Symposium*. Cancun, Mexico, May, 2000.

[11] Cabrera, L.-F. and Long, D., *Swift: Using distributed disk striping to provide high I/O data rates.* Computing Systems, Vol. **4**(4), 1991.

[12] Gibson, G., *et al.* "NASD Scalable Storage Systems", in *Proc. of the USENIX 1999 - Extreme Linux Workshop*. Monterey, CA, USA, 1999.

[13] Hartman, J. and Ousterhout, J., *The Zebra striped network file system.* ACM Transactions on Computer Systems, Vol. **13**(3), 1995.

[14] Haskin, R., *Tiger Shark - a scalable file system for multimedia.* IBM Journal of Research and Development, Vol. **42**(2): pp. 185-197, 1998.

[15] Howard, J., *et al.*, *Scale and Performance in a Distributed File System.* ACM Transactions on Computer Systems, Vol. **6**(1): pp. 51-81, 1988.

[16] Ji, M., Felten, E.W., Wang, R., and Singh, J.P. "Archipelago: An Island-Based File System for Highly Available and Scalable Internet Services", in *Proc. of the 4th USENIX Windows Systems Symposium*, August, 2000.

[17] Katcher, J., “Postmark: A new file system benchmark”, Network Appliance Corp., Sunnyvale, Technical Report, TR-3022.

[18] Kleiman, S.R. "Vnodes: An Architecture for Multiple File System Types in Sun UNIX", in *Proc. of the Summer 1986 USENIX Conference*. Atlanta, GA, USA, 1986.

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

[20] Kubiatowicz, J., *et al.* "OceanStore: An Architecture for Global-Scale Persistent Storage", in *Proc. of the ASPLOS 2000*. MA, USA. ACM, Novemeber, 2000.

[21] Lamb, E., *Graphiti: Hardware Spending Spatters*, in *Red Herring*. pp. 32-33, June 1, 2001.

[22] Lee, E. and Thekkath, C. "Petal: Distributed Virtual Disks", in *Proc. of the ASPLOS VII*. MA, USA. ACM, 1996.

[23] Lyman, P., *et al.*, , School of Information Management & Systems (SIMS), University of California at Berkeley, 2000.

[24] Macklem, R. "Not Quite NFS, Soft Cache Consistency for NFS", in *Proc. of the Winter 1994 Usenix Conference*. San Francisco, CA, USA. Usenix, January, 1994.

[25] Mahalingam, M., *et al.*, “Data Migration in a Distributed File Service”, Hewlett-Packard Labs, Palo Alto, Technical Report, HPL-2001-128, May, 2001.

[26] McVoy, L. and Staelin, C. "lmbench: Portable tools for performance analysis", in *Proc. of the Winter 1996 USENIX*. San Diego, CA, USA, January, 1996.

[27] Muntz, D., “Building a Single Distributed File System from Many NFS Servers”, Hewlett-Packard Labs, Palo Alto, CA, Technical Report, HPL-2001-176, 2001.

[28] Muntz, D., “Separating Directory Structures from Physical File Systems”, Hewlett-Packard Labs, Palo Alto, CA, Technical Report, HPL-2001-174, 2001.

[29] Muntz, D. and Liu, L., “My Permanent Address: Finding A File After It Has Moved”, Hewlett-Packard Labs, Palo Alto, Technical Report in preparation, HPL-2001-175, 2001.

[30] Preslan, K., *et al.* "A 64-bit, Shared Disk File System for Linux", in *Proc. of the 16th IEEE Mass Storage Systems Symposium*. San Diego, CA, USA, 1999.

[31] Roselli, D., Lorch, J.R., and Anderson, T.E. "A Comparison of File System Workloads", in *Proc. of the USENIX Annual Technical Conference*. San Diego, California, June 18-23, 2000.

[32] Shepler, S., *et al.*, “NFS version 4 Protocol”, RFC 3010, 2000.

[33] Thekkath, C., Mann, T., and Lee, E. "Frangipani: A Scalable Distributed File System", in *Proc. of the 16th ACM Symposium on Operating Systems Principles (SOSP)*. Saint-Malo, France. ACM, 1997.

[34] Zhang, Z. and Karamanolis, C. "Designing a Robust Namespace for Distributed File Services", in *Proc. of the 20th Symposium on Reliable Distributed Systems*. New Orleans, USA. IEEE Computer Society, October 28-31, 2001.