Active Disk File System : A Distributed, Scalable File System

Hyeran Lim[1], Vikram Kapoor[2], Chirag Wighe[3] and David H.-C. Du[1]

[1]Dept. of Computer Science and Engineering
University of Minnesota
Minneapolis, MN 55455
hrlim, du@cs.umn.edu
tel +1 612 626-7526
fax +1 612 625-0572

[2]Valicert Inc.
Trust Services Engineering
Mountain View, CA 94043
vikramk@valicert.com
tel + 1 650 567-6575
fax + 1 650 254-2148

[3]Server Products Group
Wind Rivers Systems
Alameda, CA
chiragw@windriver.com
tel +1 510 749-2026
fax +1 510 749+2914

Abstract
Consistent improvements in processor and memory technology have led to disks having greater processing power and cache memory in a compact size. This increased processing power and memory allows disks to execute more than just the basic disk operations, sometimes, even run user defined codes. Offloading part of the application processing to the disks can help reducing the latency of data manipulation as well as the amount of data transferred across the network. Such disks are called *active disks*.
In this paper, a file system for the active-disk-based data server (**ADFS**) is proposed. All data files stored on active disks are provided with operations, forming objects. For some applications such as database, application-specific operations can be run by disk processors so that only the results are returned to clients, rather than whole data file is read by the clients. Therefore, ADFS is able to reduce the application-processing overhead of the system. In ADFS, part of file system functionality of the file manager can be delegated to active disks.
Our implementation is based on CORBA specification. A set of file system interfaces is implemented for each module of each file system component. Part of the file system functionality, such as lookup, is embedded in the modules for active disks. We tested the performance of ADFS with 64 files in three-layer directory system. The performance results are obtained in a limited manner and presented in this paper.
Our ADFS is basically stateful, but the delegation of file system functionality to disks allows our file system to be more scalable and helps to overcome some of the limitations of current prevailing file systems, like NFS, by greatly reducing the work of the central file manager. In addition, it is conceivable that the central file manager could be eliminated altogether.

1 Introduction
Traditional disk drives are passive devices that start I/O operations only if there is a data request. They are closely attached to a host computer and function merely as repositories of data. Hence, their I/O operations are basically READ and WRITE. As advanced processor technology allows the intelligent storage control modules to be embedded in a small chip at a cheap price and deal with direct access from clients, the location of disk drives need not be close to hosts any longer. This type of intelligent disk drives is

categorized into *network attached secure disks* (*NASDs*)[1-4]. In NASDs, upon receiving a data request from a client, the central file manager locates the data and provides the client with necessary information to access the data such as the address of a NASD, keys for access verification, etc. Then the client provides a NASD with proof of permission granted by the file manager and the encrypted data is transferred directly from disks to the client. The main advantage of NASD based server system is that direct data transfer between disks and clients offloads the I/O operations previous issued by the central file manager to NASDs, reducing the amount of data the file manager has to manipulate and, thus, relieving I/O bus bottleneck on the file manager. However, a way of verifying legal data transfer should be carefully designed and data should be distributed across storage devices to be accessed evenly in order to take advantage of direct data transfer. This means that the storage components can be a bottleneck of system performance. With traditional disks and NASDs, clients need to retrieve the required data from disks and run the code on the retrieved data to get the desired result at client sites, which requires the network bandwidth for the transfer of a whole file. If the storage of the result is required, it should be sent back to the disks.

Recent improvements in processor and memory technology introduce a new view of functionalities for disk drives. A disk drive with a powerful processor and substantial memory can perform more than the basic functionality of disks such as storing and loading data. The increased computation power and large memory can be used to execute user-defined functions or part of the file-system functionality. These disk drives are called *active disks* [5-8]. Unlike NASDs, active disks can reduce the data transferred across the network, by running some user-defined code and filtering the unnecessary data for clients. In addition, active disks can fulfill a part of system functionality to offload file server's burden. Therefore, active disks can be a good choice to design a scalable server system. The question is how to utilize active disks effectively to achieve these benefits. It is important to find out what type of requests an active disk can process to make use of its processing power. The requests could generate small amount of data out of huge amount of data as the result of the operations running at the disk. Then active disks can reduce the amount of data transferred across the network tremendously, saving network resources and money. Database applications and on-site scientific computing are the applications using active disks in this way. Different use of active disks is to take a part of system functionality off from the file server, eliminating the performance bottleneck or a single point of failure at the file server. For example, active disks can easily be the file servers by sharing the distributed state information, making the file system scalable. Many of the present file systems are stateless for the purpose of avoiding performance degradation due to central file server and complexity of maintaining states. By letting disks understand the file system, maintain and process the distributed state information, the file system can be stateful as well as scalable.

In this paper, we propose a design of a distributed file system with the active disks attached as well as the conventional disks. Our purpose is to provide file system with active disks connected to a file manager and clients in a global environment. We view data as objects, which are the access units of storage to serve a request. The term "object" is used by Seagate Technology [9] to broaden the concept of data accessible to different file systems. In our term, an object is characterized by the data component and the operations on it. Groups of objects sharing the same data representation and operations

form a class. In other words, operations are defined on a class and objects belonging to a class share the operations. The central file manager creates and maintains the class-related state information and deals with creating/deleting of an object. Object-related state information is managed by the active disk that stores the object. When a client makes a request for an object, it locates an active disk with the help from a file manager. The request of an object identifies a series of operations required and the active disk(s) that receive(s) the request perform(s) the operations to generate the desired result. We implement our file system using CORBA specification. CORBA allows a group of communicating entities to interact with each other based on the client-server paradigm and it provides a means for the various entities in a network to access each other. Specifically, the implementation with CORBA provides a means for a diverse group of clients, file managers, and storage components to interact with each other. Another advantage is the ability of current implementations to integrate readily into object-oriented languages like C++ and JAVA. Thus it is possible for a client to access objects on a remote storage component as if they were present on the client through the use of an object-oriented implementation. Using CORBA, we implemented (simulated) the file system modules and user-defined operations. The interfaces and implementation codes of each system component (client, file manager, and disk) are grouped. Since we ran the codes at different machines within our LAN actually running NFS, we needed to filter the overhead by our NFS-based computer system. Since some processing jobs to perform a request, such as data retrieval within disks, are common in every file system, we obtained only the time to measure the file system-specific performance.
This paper is organized as follows. In section 2, four distributed file systems are briefly reviewed: Network File System (NFS), Andrew File System (AFS), file systems for Network Attached Secure Disks (NASD), and Global File System (GFS). In section 3, active disk based file system is introduced. Its system components and the software modules in each component are explained. The interaction between client/server modules for file system functionalities is presented in section 4. Finally, our implementation using CORBA and the observation from the tests are presented in section 5. Since our implementation is limited to the verification of primary functionality of ADFS, we discuss the limitation and further research regarding active disk based file system in section 6.

2 Overview of Distributed File System

In this section, a brief overview of some distributed file systems is presented. In a typical system, the file system resides in the file server. This central file server receives a request from a client, reads data from disks connected through its local bus, and transfers the retrieved data to the client across the data network. The limited memory capacity in central file server and the bandwidth of I/O bus between the file server and disks are saturated quickly and become the performance bottleneck. In a network-attached secure disks based system, read and write operation previously done by the file system has been offloaded to the network attached disks. However, the central file server still serves open/close requests and has to maintain all the state information required.

2.1 Distributed File System

The purpose of a file system is to provide clients with long-term storage of data. The access unit of data is a file which is created by a client and exists until the client deletes it. A distributed file system (DFS)[13] is an extension of the single-user file system where a number of different clients share data physically dispersed across various sites of a distributed system. DFS is a solution when the system is wider and the range of the location of storage devices is broad. It provides better availability of data and the transfer is more reliable by having duplicates closer to clients. If there is a central file server, it may be a problem if scalability is an issue. Depending on whether or not a file system maintains state regarding sessions (open/close files), a file system is either stateful or stateless. Stateful file systems are able to provide enhanced features such as cache consistency and locking, while stateless file systems are easy to implement and more efficient. Some semantics and services should be studied in order to present the complete interfaces for requests in a file system, as follows.

- Naming service and transparency
- Sharing semantics
- Caching
- Fault tolerance and replication
- Scalability and dynamic behavior

2.2 Some Distributed File Systems

2.2.1 Network File System (NFS)

Network File System (NFS)♠[14] is probably the most prevalent distributed file system in use. It operates on a client-server basis using remote procedure call (RPC) mechanism to transfer requests and data between clients and servers. NFS uses the UDP protocol. Data values are encoded in the external data representation (XDR) method. Several file systems can be integrated into UNIX virtual file system through mounting. By mounting a remote directory, the same interface can be used to access remote files as local files. When the client accesses the local file, its virtual file system under UNIX kernel invokes the UNIX file system and accesses the file stored in local disks. When the requested file is not local, the virtual file system runs the NFS client module to translate the requests to NFS protocol operations and then pass to the NFS server module at the file server through a remote procedure call. Handling each component of a path name separately avoids complexity with different path name syntaxes on different hosts and makes it easier to deal with "mount-on-mount" situations where the server is also a client of another server.

Due to the nature of stateless file service and simple design, NFS provides simple but highly efficient file service. However, an NFS request can be duplicated when transmitted or re-sent if an acknowledgement is lost. NFS does not guarantee cache consistency. File locking should be handled by separate service if it's desired. Various platforms of NFS under many operating systems make NFS almost independent of the operating system running on both clients and the server.

2.2.2 Andrew File System (AFS)

Andrew file system is much closer to a fully distributed file system than NFS. Its primary distinguishing characteristic is its scalability. The file names from users might be local or remote. The remote file service is implemented by a collection of dedicated file servers known as *vices*.

Andrew is characterized by *whole file caching*, which means that a file is brought in its entirety to the local cache when an application requests the file. As the file service is involved only for file open and close and not for others, the load on the file server is reduced. However, as the overall network system grows, even these read and write operations on the entire file could heavily load the file server and its I/O bus.

2.2.3 Network Attached Secure Disks (NASDs)

The file system for NASDs views variable size of files as *objects*. Each NASD drives can provide storage for multiple file managers and the storage capacity for each file manager in a NASD drive is called a *partition*. Each partition is independent and disjoint from each other. The actual data transfer takes place between NASDs and the clients without store-and-forwarding at the file manager machine. Instead the file manager is in charge of access control. In order to provide secure data transfer over an insecure network, the file system should support the use of cryptographic techniques in access to the disks.

The efficiency of this system heavily depends the performance of the underlying network of switching data without store and forward and low latency.

2.2.4 Global File System

One of the file systems currently being implemented is Global File System (GFS)[10-12]. GFS is a distributed file system for clustered servers (GFS nodes) share data stored in storage devices connected via a storage network. Clients make requests over the network through which a GFS node (server) is connected and receives the requests. Several GFS nodes (servers) serve the data requests from clients by retrieving data from the network storage pool via a storage network. There is a centralized lock server, which maintains the control over device locks for the concurrent accesses to the data in a storage pool. Locking mechanism is carefully designed for resource groups and devices.

Each GFS node caches data in its main memory only during I/O request processing. After each request is served, the data is either released or written back to the storage devices. It might have local storage devices to cache data so as to exploit the locality of references. GFS is a good example of a file system designed for storage devices connected via a storage area network.

3 Active Disk based File system (ADFS)

In our Active Disk-based File System (ADFS), each file is seen as a named object with operation codes combined. When some objects represent the same characteristics and share the same operations but their data components are different, the objects form a class of the type. Objects are accessed through well-defined file system interfaces. Some interfaces are used to run user-defined operation codes for some objects.

Figure 1 shows our proposed data server system with active disks attached. The central file manager, clients, and a group of active disks are attached to a high-speed network. The file manager will be involved only in system-wide decisions such as creation or deletion of classes and creation/deletion/replication of an object (an object without codes is a file in a traditional point of view). Clients access disks directly to make a request of data. In this system, active disks take over the most of the functionality of the file manager and present the file system based interfaces to clients, and clients cooperate with the active disks by embedding the part of file system interfaces.
In the following subsections, we describe the key capabilities required by active disks, and define the interfaces to access the services. We also propose a set of file system client/server modules as shown in Figure 2.

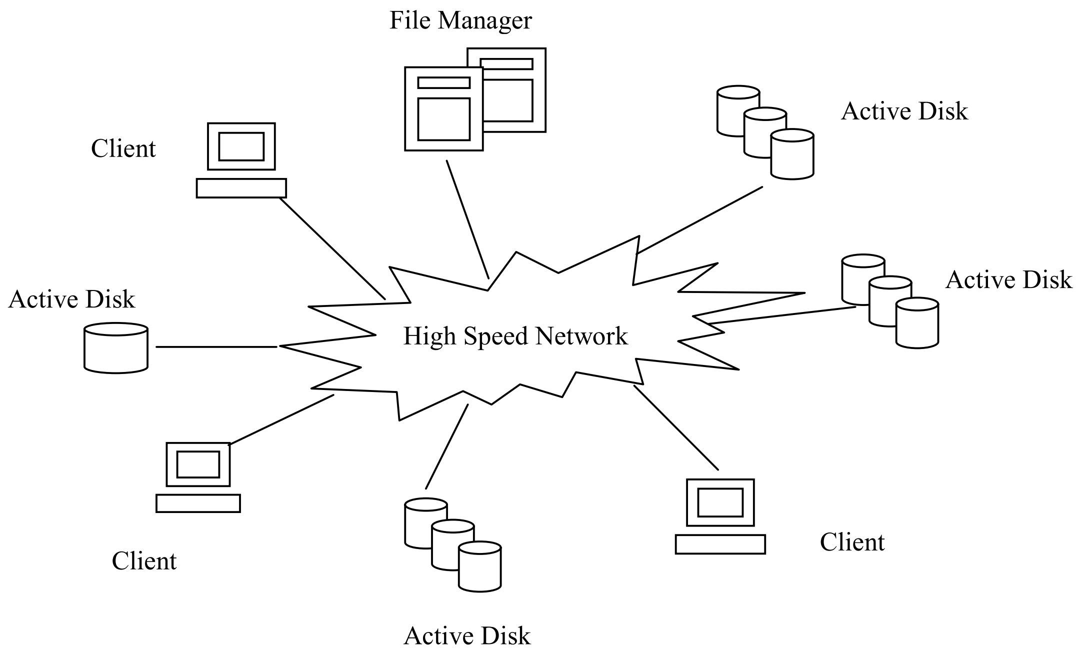

Figure 1. Architecture of an active disk based system

3.1 Data as an object

In our Active Disk-based File System (ADFS), data will be combined with its own operations as an object. The objects sharing the same data representation and operations form a class. The concept of data class allows a single copy of operations to be shared by objects belonging to the class. General file system interfaces are provided to handle all types of objects as well as files provided by traditional disk drives attached to ADFS. Both clients and disks have modules and well-defined interfaces to handle the representation of data objects.

3.2 Active Disk

The data stored in an active disk is seen as objects to clients and the central file manager, unlike the data block in the traditional view of disks. Each data block is not specified in a request. In order to provide the object-oriented view of data, a request from a client should contain object ID, an offset within the object and type of operation applied to the object. Active disk itself manages the mapping from the offset to a particular block. The request is processed through several layers with libraries available to each layer. The layers are **active disk**, **object-oriented disk** (`OOD`), and **block oriented disk** (`BOD`), where BOD is the layer dealing with the underlying disk hardware.

3.2.1 Block Oriented Disk (BOD)

The Block Oriented Disk is the abstract representation of the traditional disk that exports a block-oriented view to the other components. Disks commonly found today are viewed as block oriented disks. BOD library provides the interfaces used by BOD to access the hardware disk blocks.

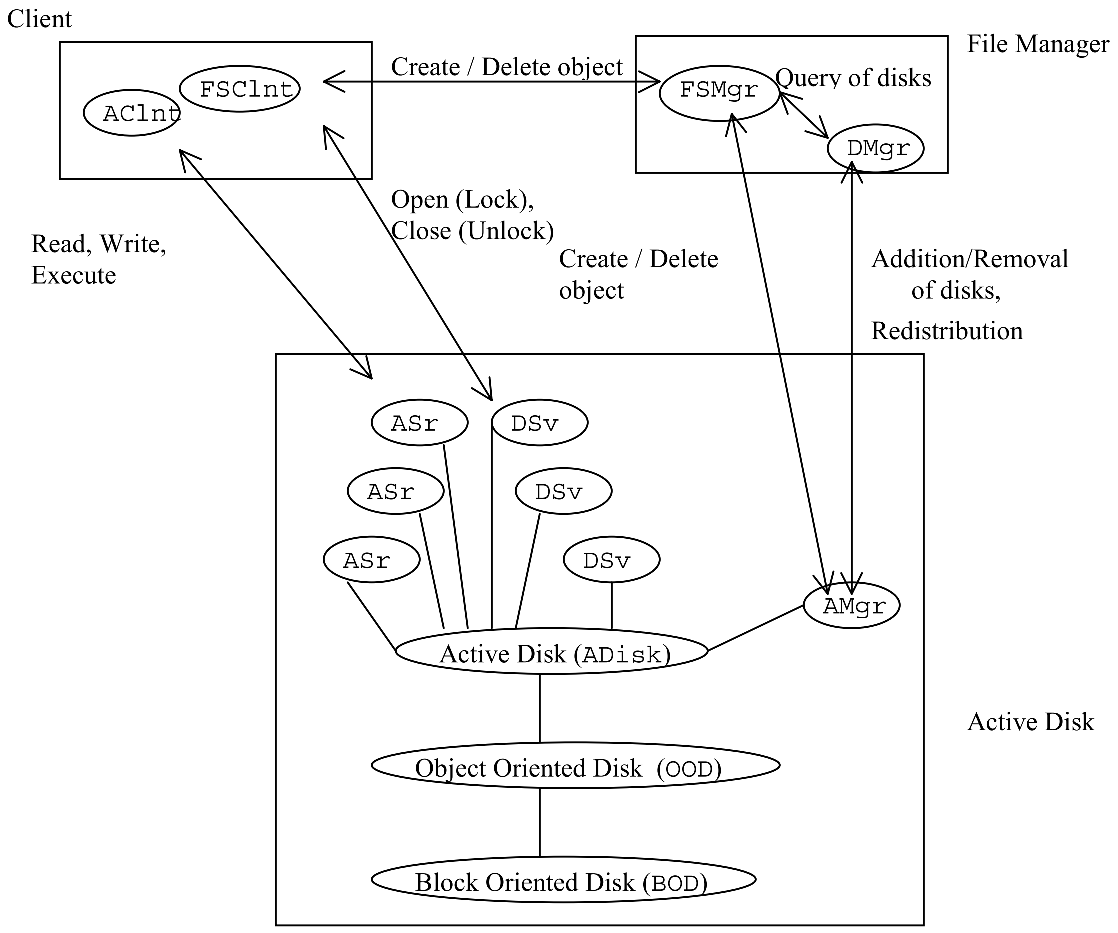

Figure 2. Serving modules of each architectural component

3.2.2 Object Oriented Disk (OOD)

The Object Oriented Disk is implemented on top of `BOD`, providing an object-oriented view of data stored on `BOD`. Clients cannot access each individual block directly and are allowed to access data located by offsets with an object ID. Then `OOD` takes the responsibility of mapping offsets to blocks and managing disk storage. In order to map the offsets to data blocks, `OOD` is given an Object ID (OID) identifying the object. This is similar to I-node used in UNIX. Since every `OOD` manages only its own space of objects, an OID is assumed to be unique only within the `OOD`. `OOD` library provides the functions and interfaces used by `OOD`.

3.2.3 Active Disk (ADisk)

The Active module is implemented on top of `OOD`. It provides common functions needed to access `OOD`, such as authentication. All servers running on an active disk share `ADisk` to communicate with `OOD` and serve their duties to clients or the central file manager.

3.2.4 Active Disk Server Modules Providing File Services

There are several servers running on top of ADisk, interacting with clients to provide file services: Active Disk Server (`ASrv`) and File System Server (`DSrv`). **`ASrv`** is responsible for serving data requests from clients, such as read and write. Several `ASrv` processes can be running on an active disk simultaneously, each serving a request from a client by calling functions provided by ADisk library.

`DSrv` is a file system server also running on top of `ADisk`. As we mentioned earlier, partial file system is launched on each active disk to serve individual file system service to clients. `DSrv` uses the object table in the active disk (maintained by active disk manager which is described in the next subsection) to provide pathname lookup. Since each `DSrv` has limited knowledge of the name space due to its residency on an active disk, it cannot parse the whole path. Therefore, `DSrv` parses a path as much as it can and sends the rest of the path back to client who's responsible for parsing the rest of the path. The other responsibility of `DSrv` is to maintain the state information of open files and locked files. The state information is used for cache consistency.

3.2.5 Active Disk Server Modules Providing Management : Active Disk Manager (`AMgr`)

The Active Disk Manager (`AMgr`) is responsible for maintaining storage space in an active disk by creating and deleting objects on the active disk. There is only one `AMgr` per each active disk to provide the infrastructure of disk storage for clients' requests. It coordinates dynamic removal and addition of an active disk, and acts as a client for Disk Manager (`DMgr`) module in the central file manager.

3.3 File Manager

Unlike the traditional file system server, our file manager delegates many of its responsibility to active disks and clients. Its functions remaining after delegation are mainly those that require knowledge of the entire file system. There are two main functions: Disk Management and Overall File System Operations.

3.3.1 Disk Manager (DMgr)

The Disk Manager (`DMgr`) keeps track of the active disks currently in the system. It is the responsibility of `DMgr` to support dynamic disk management. By dynamic disk management, we mean that it should be possible to remove an active disk temporarily and then attach it back to the system without bringing the system down. DMgr maintains the information on the available disks in the Super Block of the Root Disk. DMgr also checks periodically if active disks are alive or down.

3.3.2 Overall File System Manager (FSMgr)

This is the remnant of the central file server. Most of its functionality such as read/write, open/close, lookup, and locking, has been distributed. However, the file manager still needs to make system-wide decisions such as file creation. During file creation, `FSMgr` interacts with `DMgr` in order to get the information regarding available active disks for the new file and creates the file on one of the available active disks.

When a client tries to access data stored in a disk which is down currently, the request is not responded by the disk and the client will notice the failure eventually. Conceptually we considered the replication of the data for fault-tolerance. If the primary disk is down, the backup disk will take over the request by the intervention of the central file manager. However, our implementation has not included the fault-tolerant feature, so there is no backup disk of the desired disk and the access trial would be failed.

3.4 Client

A client can be any machine that uses ADFS to access data. We assume that the clients be in the same network as active disks and the file manager so that data can be transferred directly between clients and active disks. The operating system running on clients is modified so that it can call ADFS functions. Two ADFS components running on the kernel of a client are **active disk client** (`AClnt`) and **File System Client** (`FSClnt`).

`AClnt` is responsible for manipulating data stored on active disks. It makes a request of reading and writing data and running codes on data embedded in an object. In our implementation, it acts as a CORBA client to `ASrv` in active disks.

`FSClnt` provides the file system functionality to client programs. It interacts with `DSrv` on the active disks where the accessed objects lie. It contacts the file manager when it creates a new file.

4 Operation Details

In this section, we discuss the implementation of some aspects of ADFS. The discussion presented in this section is limited on the components providing file system interfaces to active disks. The implementation of the most aspects, except executing user codes at an active disk, has been done and tested as shown in the next section♥.

4.1 Dynamic Disk Management

`AMgr` of each active disk is responsible for managing storage space in the active disk. It maintains a special object, called Super Block, which contains important information such as the name of the disk, user objects defined, and the codes for user objects. On the other hand, the central file manager has a disk called “Root Disk.” This disk acts as the permanent storage for management of files and disks. The Root Disk might be replicated for reliability. A standby central file manager can be added for reliable file manager. Root disk contains a Super Block, which keeps all the data structures used by the file manager.

When a new active disk is booted and needs to be added, **`AMgr`** of the active disk advertises that it's ready to serve requests. Then **`FSMgr`** of the central file manager contacts the **`AMgr`** in order to add the active disk in its table. *If an active disk goes down*, **`FSMgr`** pings **`AMgr`** of the disk and deletes the entry of the disk from its table.

4.2 File System Service

4.2.1 Creating an Object

When a client (**`FSClnt`**) requests to create a new object, **`FSMgr`** at the file manager contacts **`DMgr`** to find out an available active disk. Then **`FSMgr`** sends a request to create a new object to the active disk (**`AMgr`**), which, in turn, modifies its directory table to include this new object. The file manager (**`FSMgr`**) uses transactions or 3-phase protocol to ensure atomicity of this request.
In order to decide which available active disk to create the new object, **`FSMgr`** can choose one of several policies. One is to put the new object in the active disk as close as to the active disk containing the direct parent directory of the new object. It could be the same active disk. In extreme cases, all objects of a client can be in the same active disk. Another policy is to find an active disk with most free disk space (**`AMgr`**) and lets **`DMgr`** update the state information of disks.

4.2.2 Deleting an Object

In order to delete an object, **`FSMgr`** first updates the directory by removing the entry of the object in the directory table. Then it contacts **`AMgr`** of the active disks where the object resides and asks the **`AMgr`** to delete the object. The corresponding **`AMgr`** deletes the object and frees its disk space. **`FSMgr`** makes sure atomic operation of the object deletion.

4.2.3 Lookup

In order to access its data, a client (**`FSClnt`**) contacts the file manager (**`FSMgr`**) to find out the active disk that contains the root directory of the client. This information is cached so that the client can starts with the disk without contacting the file manager in the future unless the disk is down.
Whenever a client needs to access a data object, its **`FSClnt`** looks up the pathname as far as it can in the cache and then sends a request with the remaining pathname to **`DSrv`** holding that directory. For example, assume a client is looking up /A1/B11/C111. Let's say that / is cached and A1 is the Object X on disk Y. Then the client sends a request to look up B11/C111 to **`DSrv`** on Disk Y with X as an argument (which is the parent directory of the given remaining pathname).
Each component in the pathname is assigned an object ID, since we treat directory and file names as a special type of objects. Starting from the object ID indicating the given directory, whenever each component in a pathname is parsed, an object ID identifying the pathname up to the component is returned. When the pathname is completely parsed, **`DSrv`** returns the object ID of the data object to **`FSClnt`** in the client so that the client can invoke **`AClnt`** to apply operations on the data object. It is possible for **`DSrv`** to find out that the data accessed is located in another active disk. Then the **`DSrv`** returns the remaining pathname and the disk name with which **`FSClnt`** can start lookup. It is

repeated until the pathname is completely parsed. Since **DSrv** parses a pathname as much as possible without returning to clients in the middle, this type of lookup eliminates pathname recursion at client machines.

4.1.4 Stateful File System Operations

The file system server modules in active disks export stateful interfaces to access data. The **open object table** is maintained by **DSrv**s recording the object ID of the open file, the client who opened the file and the mode in which it was opened. This table is shared among **DSrv**s serving clients' open/close requests. The **lock table** is also maintained by **DSrv**s.

When **DSrv** receives an **open** request, the lock table is searched for the status of the object (file). If the object is not locked, it may be opened. The mode of the object being opened and the client opening the object are recorded in the open table. The lock entry of the object is added to the lock table.

In case of a **close** request, it is required to verify the open object table entry. And then the entry is deleted from the open object table. If it is the only client using the object or if the object was locked exclusively, serving the close request also releases the lock to the object by deleting the lock entry of the object from the lock table.

Reading and writing are basically two operations mostly done on data. We add **execute** operations to retrieve code object and run the code on data object. Although read and write can be considered as codes to be run by **execute**, we separate them since they are the most frequent operations. Read, Write, and Execute commands are sent by **AClnt** in a client to the active disk containing data object. And one **ASrv** is assigned to serve the request by the active disk. ASrv uses active disk library to transfer object-oriented disk operations to block-oriented disk operations.

5 Simulation

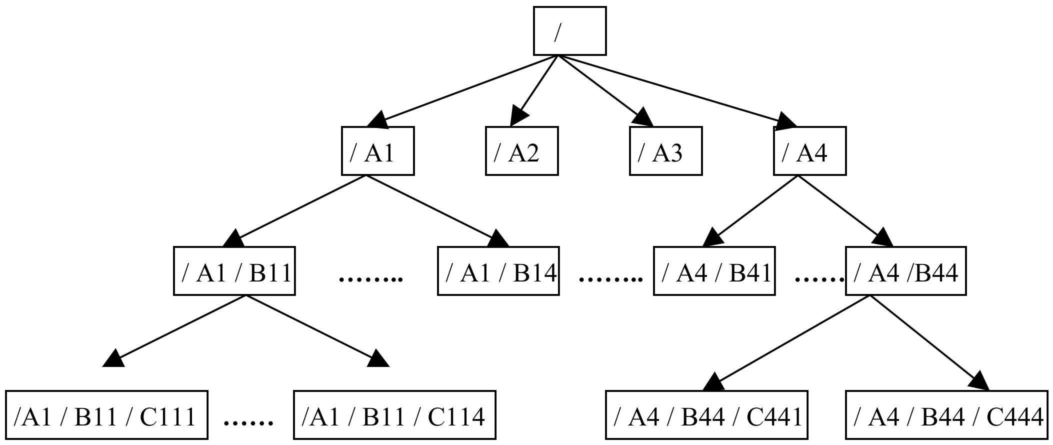

Figure 3. Directory structure used for simulation.

We used CORBA to simulate the active disk based file system. The CORBA specification provides a client / server based communication between entities. Initially, the basic functionality of the file system has been tested and verified. Then we tested the efficacy of ADFS under the following configuration. First, four active disks were running in the file system. The file structure consisted of a root directory containing four directories (A1, A2, A3, A4), each of which contained four directories (Bi1,…, Bi4 for i=1,…4) , and again each Bij had subdirectories Cij1,Cij2, Cij3, and Cij4, as shown in Figure 3. There were 64 files in the file system. These files were created by the procedure in which a client contacts the file manager to consult where(which disk) to create an object, and then sends a request for object creation to **`AMgr`** on that disk with the object attributes. These files were used to obtain the following results.

5.1 Eliminating Pathname Recursion

Each of 64 files was accessed by a read or write operation, and the number of lookup requests at the file manager and active disks were recorded. Since the root directory was assumed to be cached, its lookup requests were not counted. All 64 calls were made by one client in one second. The average size of a directory is 1 Kbytes. The request and reply messages are small and ignored in consideration of network bandwidth. It was our intention to identify the overhead associated with the name resolution – resolving the pathname to DiskID and ObjectID. The main variables under consideration are the load on the I/O bus of either the server or active disks, the network bandwidth used and latency in terms of RPC calls. The other kinds of latency are constant and ignored.

Type of file system	Max. Load on I/O bus (Kbytes/Sec)	Net. BW used (Kbytes/Sec)	Avg. latency/ lookup (No. RPC calls)
NFS (Centralized file server)	128	Negligible	1
NASD-based file system	32	128	3
ADFS – Optimal case	32	Negligible	1
ADFS – Worst case (overloaded disk)	128	Negligible	1
ADFS –Worst case (Cross referenced directories)	32	128	3

Table 1. Comparison of lookup performance in different file system

- In the centralized file server such as NFS, each of the lookup calls was directed to the centralized file server. Therefore, the file server received 64 lookup requests per second. For each individual request(Ax/Byy/Czzz), two levels of reads were required to get the two directories(Byy, Czzz). Thus the load on the I/O bus of the server is 64*2*1Kbytes/sec = 128 Kbytes/sec. It was assumed that the NFS server could parse the pathnames and hence, in our implementation of NFS server, the latency is one RPC call for the lookup to the server.

- In NASD-based file system, the latency is three RPC calls, one to server and two to NASD disks, because the directory information lies in the NASD disks. The file server also has to access two directory entries. Two directory entries are transmitted across network for each lookup, resulting in the load on the network of 128 Kbytes/sec.
- The performance of ADFS can be different in various situations. The **optimal** case of ADFS is when each of direct subdirectories of root resides completely in one active disk. In other words, all files under A1 resides in one disk, all files under A2 are on another disk, and so on. Therefore, each active disk receives 16 requests, and the load on the active disk I/O bus is thus 32 Kbytes/sec. In addition, each request is completely parsed at each particular active disk, producing the latency of one RPC call. The **worst** situation can happen either when all files were created on a single active disk, or when each entry in the pathname was on a separate disk. Under the case of overloaded active disk, the I/O busload would be the same as the centralized file server, 128 Kbytes/sec. However, the performance can be poorer than the centralized file server, since the disk would perform slower than a file server. In the latter worst case, the file system would perform similar to a NASD file system. Thus it would suffer from high latency and heavy use of network bandwidth. ADFS performs similar to centralized file system or NASD based file system in the worst case, and generally better than both types of file system by eliminating pathname recursion and reducing overhead.

5.2 Distribution of State Information

The same simulation was repeated except that all the files were opened. At the end of the test, the size of the open object tables on the file manager and active disks were compared. We assume that each entry in the open object table occupies 100 bytes.

- Under NFS or NASD based file system, all 64 open requests come to the file server which maintains the open object table. Therefore, the table size is 6400 bytes.
- In ADFS, we assume that the open files are distributed evenly across the active disk. Thus, each active disk receives 16 requests for a table size of 1600 bytes.

The information is the open object table is helpful for implementing file operations like locking or cache coherency. The above simulation indicates that as the number of objects –or files- in the system increases, the amount of state information maintained by the central file server in traditional file systems become excessive. This is one of the main reasons why most current distributed file systems do not provide stateful file system primitives. For example in NFS, locking has to be implemented on top of the existing stateless file system. The distribution of state information allows for a richer set of primitives at the file system level, which can be useful for applications.

6 Conclusions

As we have discussed, active disks have potential of reducing the burden of central file server by taking part of file system functionality over from the central file manager. In addition, active disks reduce the network traffic by running some system or user-define codes and returning only the desired results to clients. From our design of file system, active disks are even able to reduce both of the latency and bandwidth by eliminating pathname recursion in lookup calls, offload a large amount of work from the file manager. Distributing state information across active disks helps creating a more scalable file system, increasing availability and reliability of the file system.
However, in order to achieve these advantages, the files should be evenly distributed. When only one copy of the user codes for the same type of objects resides in a disk and is shared among the objects in the disk, the distribution of files may not be easy. For the sake of availability of data and code objects, replication might be introduced, which makes the consistency of data harder to implement. These are not included in our design and should be considered in further research.

References

[1] G A Gibson *et al.,* “File systems for Network-Attached Secure Disks,” *CMU technical report CMU-CS-97-118*, 1997.

[2] H. Gobioff, G Gibson and D Tygar. “Security for Network Attached Client Devices,” *CMU Technical Report CMU-CS-97-185,* 1997

[3] G A Gibson *et al*., “File Server Scaling with Network-Attached Secure Disks,” *Sigmetrics*, pages 272-284, 1997

[4] G A Gibson *et al*., “A Case for Network-Attached Secure Disks,” *CMU Technical Report CMU-CS-96-142*, 1996

[5] E Riedel, G A Gibson and C Faloutsos, “Active Storage for Large-scale Data Mining and Multimedia,” *Proceedings of the 1998 Very Large Data Bases conference* (VLDB), pages 62-73, August 1998

[6] E Riedel and G A. Gibson. “Active Disks - Remote Execution for Network-Attached Client,” *Technical Report CMU-CS-97-198,* 1998.

[7] M. Uysal, A. Acharya and J. Saltz. “Evaluation of Active Disks for Decision Support Databases,” *Proceedings of the 6th International symposium on High-Performance Computer Architecture*, Toulouse, France, January 10-12, 2000

[8] A. Acharya, M. Uysal and J. Saltz. “Active Disks: Programming Model, Algorithms and Evaluation,” *Eighth International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS98), pages 81-91, San Jose, California, October 4-7, 1998.

[9] Seagate Technology, Inc. “Object Oriented Devices: Description of requirements*,” White Paper*.

[10] M. T. O'Keefe, “Shared File Systems and Fibre Channel,” *Proceedings of the Sixth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, pages 1-16, March 23-26, 1998, College Park, MD.

[11] S. R. Soltis, T. M. Ruwart, M. T. O'Keefe, “The Global File System,” *Proceedings of the Fifth NASA Goddard Space Flight Center Conference on Mass Storage Systems and Technologies*, September 17-19, 1996, College Park, MD.

[12] S. R. Soltis. The Design and Implementation of a Distributed File System based on Shared Network Storage, PhD Thesis. University of Minnesota, 1997

[13] E. Levy and A. Silberschatz, "Distributed File Systems: Concepts and Examples," *ACM Computing Survey,* 22, 4, Pages 321 – 374, December 1990

[14] Brent Callaghan, *NFS Illustrated* (Addison-Wesley Professional Computing Series, 1999)

♠ NFS quoted in this paper is NFS version 1.

♥ The detailed implementation information can be provided upon request.