**DISTRIBUTED FILE SYSTEMS**

**Introduction**

* A file system is responsible for the organization, storage, retrieval, naming, sharing, and protection of files.
* File systems provide directory services, which convert a file name (possibly a hierarchical one) into an internal identifier (e.g. inode, FAT index). They contain a representation of the file data itself and methods for accessing it (read/write).
* The file system is responsible for controlling access to the data and for performing low-level operations such as buffering frequently used data and issuing disk I/O requests.

A distributed file system is to present certain degrees of **transparency** to the user and the system:

* **Access transparency:** Clients are unaware that files are distributed and can access them in the same way as local files are accessed.
* **Location transparency:** A consistent name space exists encompassing local as well as remote files. The name of a file does not give it location.
* **Concurrency transparency:** All clients have the same view of the state of the file system. This means that if one process is modifying a file, any other processes on the same system or remote systems that are accessing the files will see the modifications in a coherent manner.
* **Failure transparency:** The client and client programs should operate correctly after a server failure.
* **Heterogeneity:** File service should be provided across different hardware and operating system platforms.
* **Scalability:** The file system should work well in small environments (1 machine, a dozen
* machines) and also scale gracefully to huge ones (hundreds through tens of thousands of systems).
* **Replication transparency:** To support scalability, we may wish to replicate files across multiple servers. Clients should be unaware of this.
* **Migration transparency:** Files should be able to move around without the client's knowledge. Support fine-grained distribution of data: To optimize performance, we may wish to locate individual objects near the processes that use them.
* **Tolerance for network partitioning:** The entire network or certain segments of it may be unavailable to a client during certain periods (e.g. disconnected operation of a laptop). The file system should be tolerant of this.

**File service types**

To provide a remote system with file service, we will have to select one of two models of operation.

Models of operation

**The upload/download model**

In this model, there are two fundamental operations:

* *read file* transfers an entire file from the server to the requesting client, and
* *write file* copies the file back to the server.

It is a simple model and efficient in that it provides local access to the file when it is being used.

Three problems are evident. It can be wasteful if the client needs access to only a small amount of the file data. It can be problematic if the client doesn't have enough space to cache the entire file. Finally, what happens if others need to modify the same file?

The second model is a **remote access model**

The file service provides remote operations such as *open*, *close*, *read bytes*, *write bytes*, *get attributes*, etc. The file system itself runs on servers. The drawback in this approach is the servers are accessed for the duration of file access rather than once to download the file and again to upload it.

Another important distinction in providing file service is that of understanding the difference between *directory service* and *file service*. A directory service, in the context of file systems, maps human-friendly textual names for files to their internal locations, which can be used by the file service. The file service itself provides the file interface (this is mentioned above). Another component of file distributed file systems is the client module. This is the client-side interface for file and directory service. It provides a local file system interface to client software (for example, the vnode file system layer of a UNIX kernel).

**Introduction**

* File system were originally developed for centralized computer systems and desktop computers
* File system was as an operating system facility providing a convenient programming interface to disk storage.
* Distributed file systems support the sharing of information in the form of files and hardware resources.
  + With the advent of distributed object systems (CORBA, Java) and the web, the picture has become more complex.

Figure 1 provides an overview of types of storage system.

**Figure 1. Storage systems and their properties**

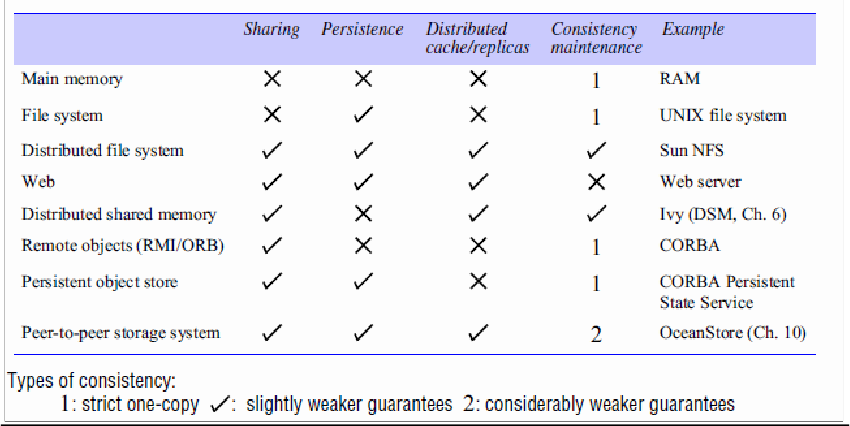
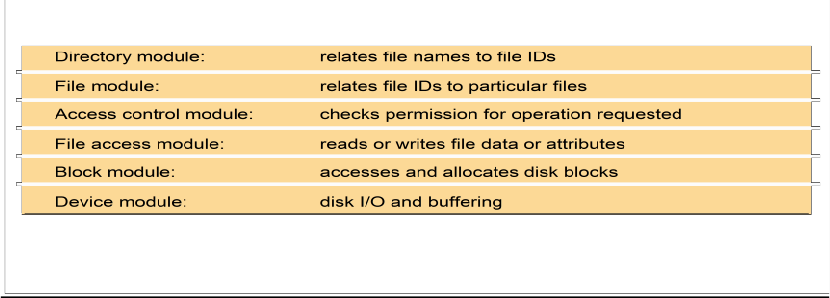


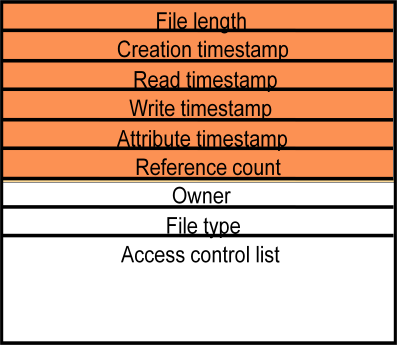
Figure 2 shows a typical layered module structure for the implementation of a non- distributed file system in a conventional operating system.

**Figure 2. File system modules**



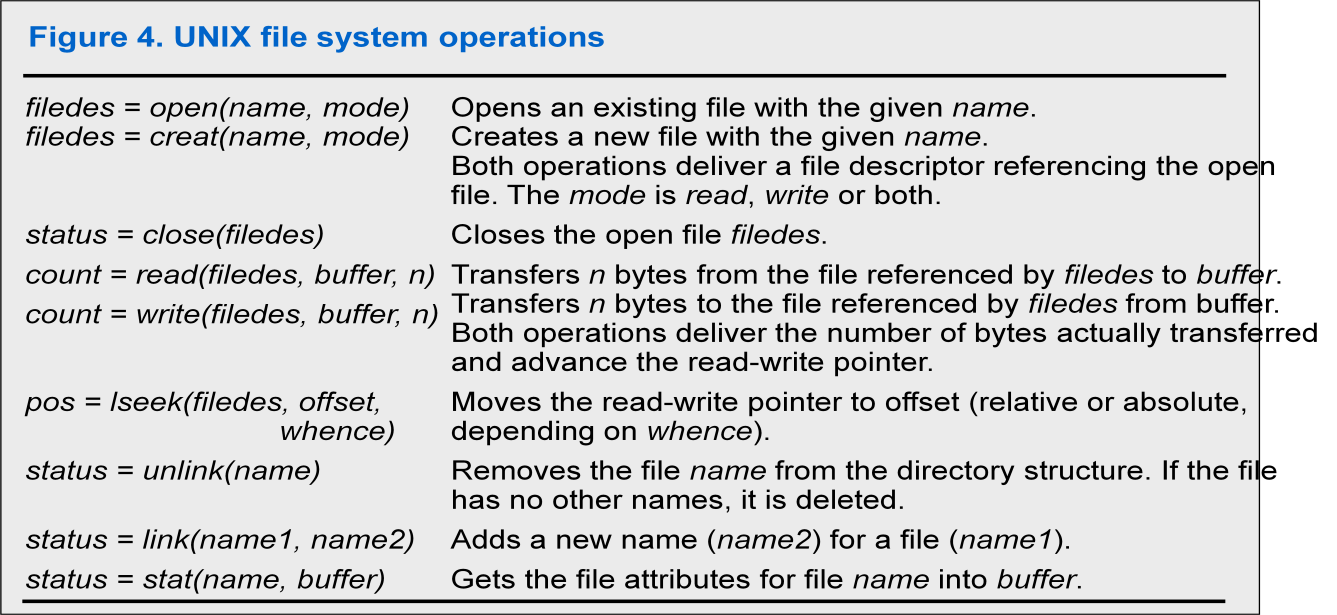
* File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
* Files contain both data and attributes.
* A typical attribute record structure is illustrated in Figure 3.

**Figure 3. File attribute record structure**



 Figure 4 summarizes the main operations on files that are available to applications in

UNIX systems.



Distributed File system requirements

 Related requirements in distributed file systems are:

* Transparency
  + - * Concurrency
      * Replication
      * Heterogeneity
      * Fault tolerance
      * Consistency
      * Security
      * Efficiency

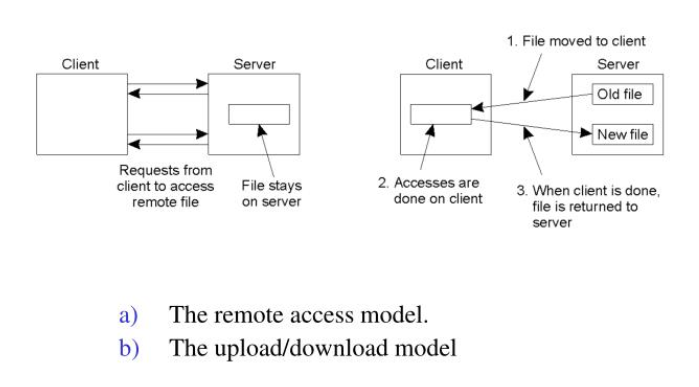
**Case studies**

**File service architecture**

**•** This is an abstract architectural model that underpins both NFS and AFS. It is based upon a division of responsibilities between three modules – a client module that emulates a conventional file system interface for application programs, and server modules, that perform operations for clients on directories and on files. The architecture is designed to enable a *stateless* implementation of the server module.

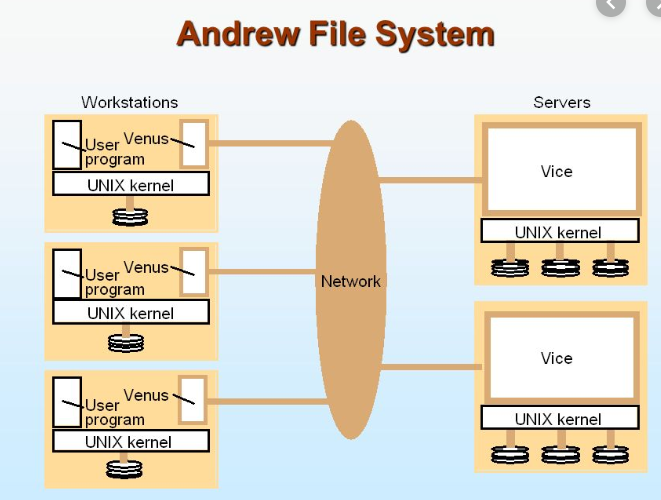
**SUN NFS**

**•** Sun Microsystems’s *Network File System* (NFS) has been widely adopted in industry and in academic environments since its introduction in 1985. The design and development of NFS were undertaken by staff at Sun Microsystems in 1984. Although several distributed file services had already been developed and used in universities and research laboratories, NFS was the first file service that was designed as a product. The design and implementation of NFS have achieved success both technically and commercially.



**Andrew File System**

**•** Andrew is a distributed computing environment developed at Carnegie Mellon University (CMU) for use as a campus computing and information system. The design of the Andrew File System (henceforth abbreviated AFS) reflects an intention to support information sharing on a large scale by minimizing client-server communication. This is achieved by transferring whole files between server and client computers and caching them at clients until the server receives a more up-to-date version.



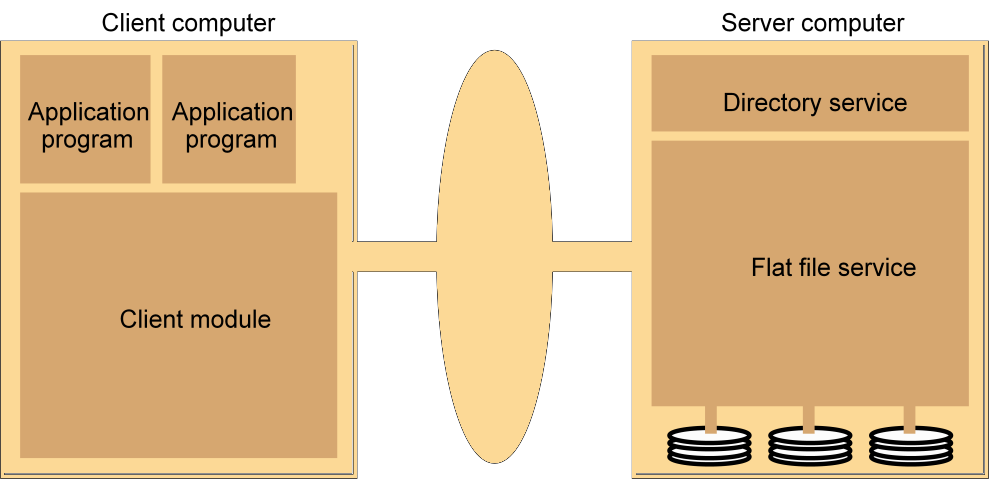
**File Service Architecture**

An architecture that offers a clear separation of the main concerns in providing access to files is obtained by structuring the file service as three components:

* A flat file service
* A directory service
* A client module.

The relevant modules and their relationship is shown in Figure 5.

Figure 5. File service architecture



The Client module implements exported interfaces by flat file and directory services on server side.

Responsibilities of various modules can be defined as follows:

 Flat file service:

 Concerned with the implementation of operations on the contents of file.

Unique File Identifiers (UFIDs) are used to refer to files in all requests

For flat file service operations. UFIDs are long sequences of bits chosen so that each file has a unique among all of the files in a distributed system.

 Directory service:

 Provides mapping between text names for the files and their UFIDs.

Clients may obtain the UFID of a file by quoting its text name to directory service. Directory service supports functions needed generate directories, to add new files to directories.

 Client module:

 It runs on each computer and provides integrated service (flat file and directory) as a single API to application programs. For example, in

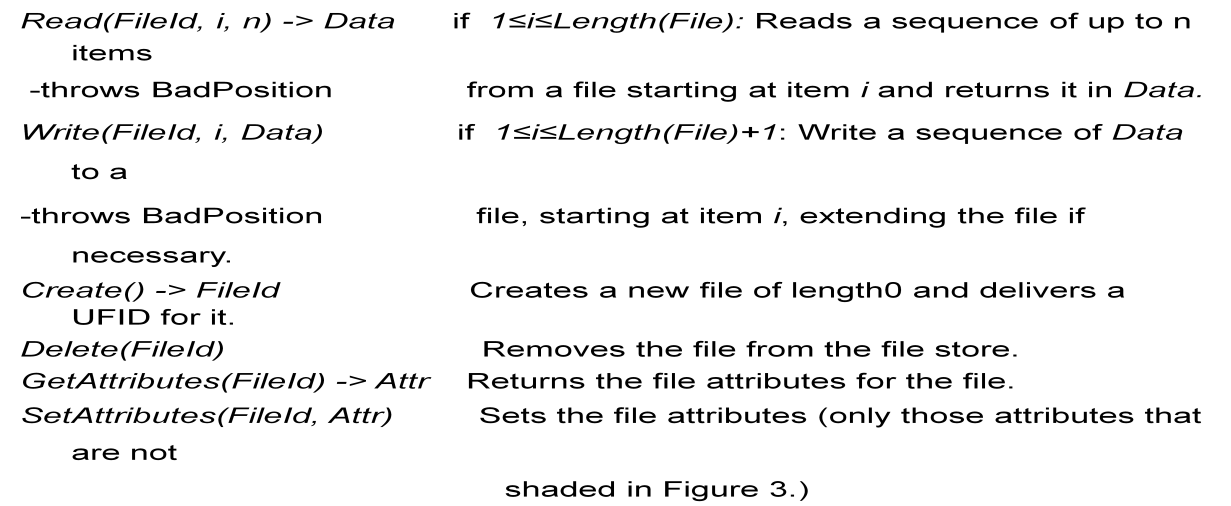
UNIX hosts, a client module emulates the full set of Unix file operations.

It holds information about the network locations of flat-file and directory server processes; and achieve better performance through implementation of a cache of recently used file blocks at the client.

Flat file service interface:

Figure 6 contains a definition of the interface to a flat file service.

**Figure 6. Flat file service operations**



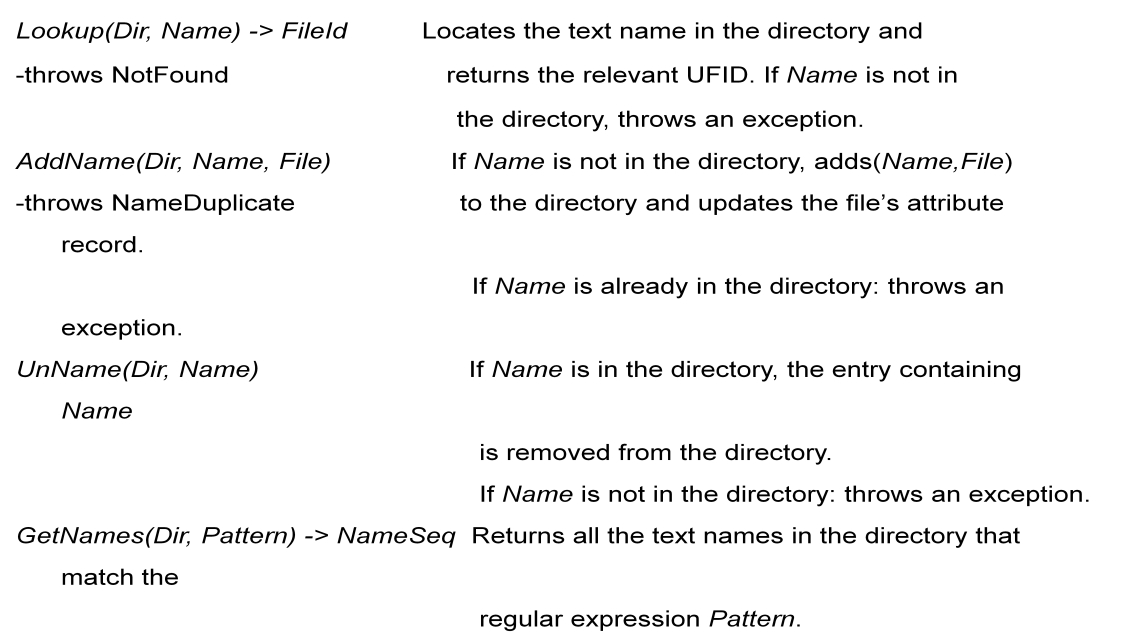
 Access control

 In distributed implementations, access rights checks have to be

performed at the server because the server RPC interface is an otherwise unprotected point of access to files.

 Directory service interface

 Figure 7 contains a definition of the RPC interface to a directory service.

**Figure 7. Directory service operations**

 Hierarchic file system

A hierarchic file system such as the one that UNIX provides consists of a

number of directories arranged in a tree structure.

 File Group

A file group is a collection of files that can be located on any server or moved between servers while maintaining the same names.

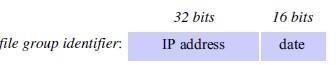
– A similar construct is used in a UNIX file system.

– It helps with distributing the load of file serving between several servers.

– File groups have identifiers which are unique throughout the system (and hence for an open system, they must be globally

unique).

To construct globally unique ID we use some unique attribute of the machine on which it is created. E.g: IP number, even though the file group may move subsequently.



**DFS: Case Studies**

 NFS (Network File System)

**** Developed by Sun Microsystems (in 1985)

**** Most popular, open, and widely used.

**** NFS protocol standardized through IETF (RFC 1813)

 AFS (Andrew File System)

**** Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)

**** A research project to create campus wide file system.

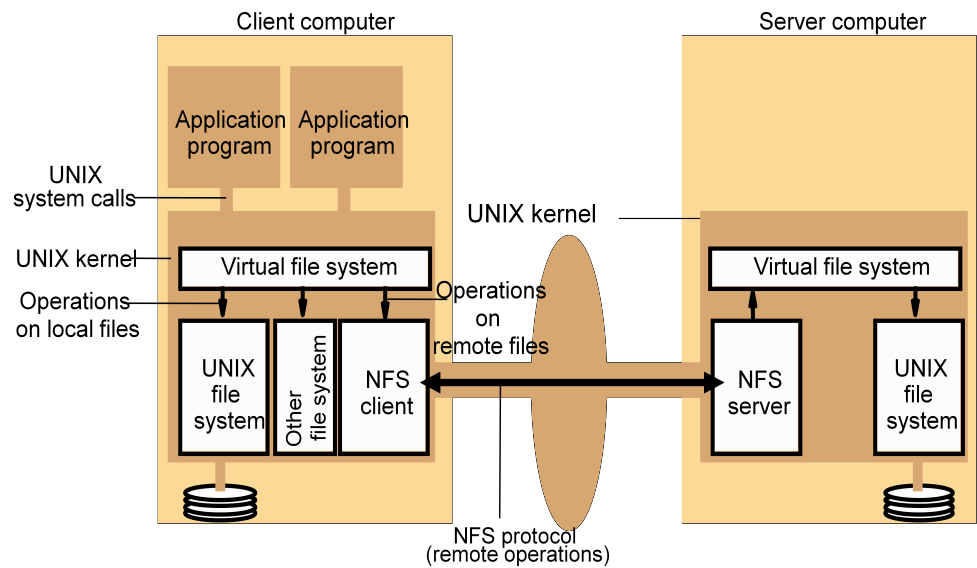
**** Public domain implementation is available on Linux (LinuxAFS)

**** It was adopted as a basis for the DCE/DFS file system in the Open Software

Foundation (OSF, [www.opengroup.org)](http://www.opengroup.org) DEC (Distributed Computing

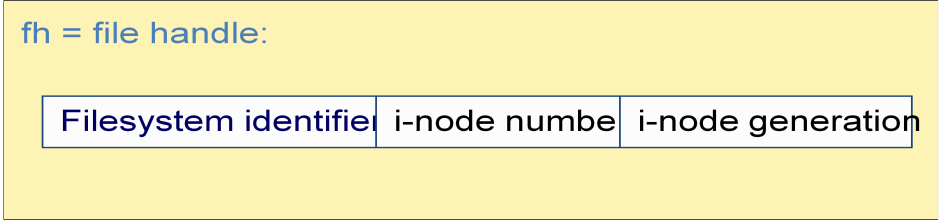
Environment

**NFS architecture**

Figure 8 shows the architecture of Sun NFS

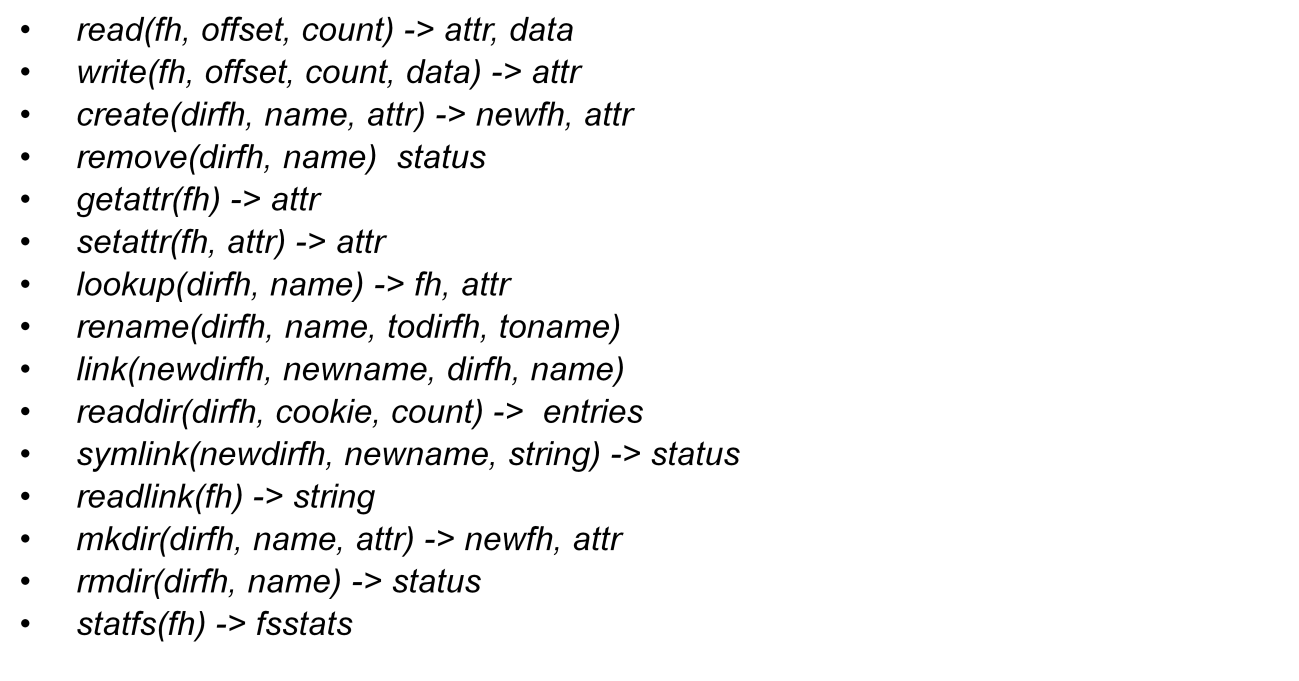
Page | 135

 The file identifiers used in NFS are called file handles.



 A simplified representation of the RPC interface provided by NFS version 3 servers is shown in Figure 9.

**Figure 9. NFS server operations (NFS Version 3 protocol, simplified)**



 NFS access control and authentication

 The NFS server is stateless server, so the user's identity and access rights must be checked by the server on each request.

 In the local file system they are checked only on the file’s access

permission attribute.

 Every client request is accompanied by the userID and groupID

 It is not shown in the Figure 8.9 because they are inserted by the RPC

system.

 Kerberos has been integrated with NFS to provide a stronger and more comprehensive security solution.

 Mount service

 Mount operation:

mount(remotehost, remotedirectory, localdirectory)

 Server maintains a table of clients who have mounted filesystems at that server.

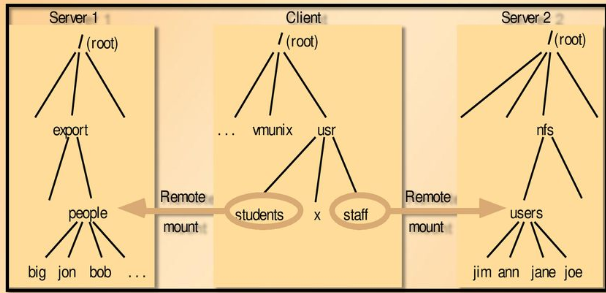
 Each client maintains a table of mounted file systems holding:

< IP address, port number, file handle>

 Remote file systems may be hard-mounted or soft-mounted in a client computer.

 Figure 10 illustrates a Client with two remotely mounted file stores.

**Figure 10. Local and remote file systems accessible on an NFS client**



 Automounter

 The automounter was added to the UNIX implementation of NFS in order to

mount a remote directory dynamically whenever an ‘empty’ mount point is

referenced by a client.

 Automounter has a table of mount points with a reference to one or more

NFS servers listed against each.

 it sends a probe message to each candidate server and then uses the mount service to mount the file system at the first server to respond.

 Automounter keeps the mount table small.

 Automounter Provides a simple form of replication for read-only file systems.

 E.g. if there are several servers with identical copies of /usr/lib then each server will have a chance of being mounted at some clients.

 Server caching

 Similar to UNIX file caching for local files:

 pages (blocks) from disk are held in a main memory buffer cache until the space is required for newer pages. Read-ahead and delayed-write optimizations.

 For local files, writes are deferred to next sync event (30 second intervals).

 Works well in local context, where files are always accessed through the

local cache, but in the remote case it doesn't offer necessary synchronization guarantees to clients.

 NFS v3 servers offers two strategies for updating the disk:

 Write-through - altered pages are written to disk as soon as they are

received at the server. When a write() RPC returns, the NFS client knows that the page is on the disk.

 Delayed commit - pages are held only in the cache until a commit() call is received for the relevant file. This is the default mode used by NFS v3

clients. A commit() is issued by the client whenever a file is closed.

 Client caching

 Server caching does nothing to reduce RPC traffic between client and server

 further optimization is essential to reduce server load in large networks.

 NFS client module caches the results of read, write, getattr, lookup and readdir operations

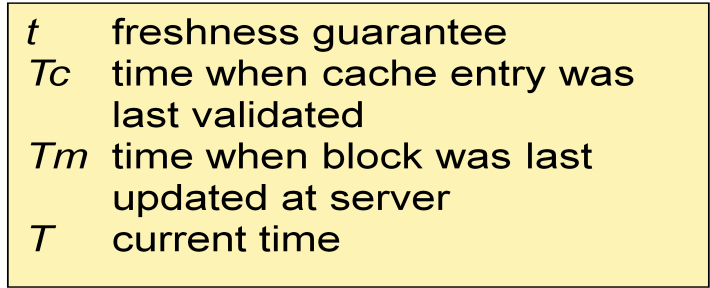
 synchronization of file contents (one-copy semantics) is not guaranteed when two or more clients are sharing the same file.

 Timestamp-based validity check

 It reduces inconsistency, but doesn't eliminate it.

 It is used for validity condition for cache entries at the client:

*(T - Tc < t) v (Tmclient = Tmserver)*



 it is configurable (per file) but is typically set to 3 seconds for files and 30 secs. for directories.

 it remains difficult to write distributed applications that share files with NFS.

 Other NFS optimizations

 Sun RPC runs over UDP by default (can use TCP if required).

 Uses UNIX BSD Fast File System with 8-kbyte blocks.

 reads() and writes() can be of any size (negotiated between client and server).

 The guaranteed freshness interval t is set adaptively for individual files to reduce

getattr() calls needed to update Tm.

 File attribute information (including Tm) is piggybacked in replies to all file

requests.

 NFS performance

 Early measurements (1987) established that:

 Write() operations are responsible for only 5% of server calls in typical

UNIX environments.

 hence write-through at server is acceptable.

 Lookup() accounts for 50% of operations -due to step-by-step pathname resolution necessitated by the naming and mounting semantics.

 More recent measurements (1993) show high performance.

 see [www.spec.org f](http://www.spec.org/)or more recent measurements.

 NFS summary

 NFS is an excellent example of a simple, robust, high-performance distributed

service.

 Achievement of transparencies are other goals of NFS:

 Access transparency:

 The API is the UNIX system call interface for both local and

remote files.

 Location transparency:

 Naming of file systems is controlled by client mount operations, but transparency can be ensured by an appropriate system

configuration.

 Mobility transparency:

 Hardly achieved; relocation of files is not possible, relocation of file systems is possible, but requires updates to client configurations.

 Scalability transparency:

 File systems (file groups) may be subdivided and allocated to separate servers.

 Replication transparency:

– Limited to read-only file systems; for writable files, the SUN Network Information Service (NIS) runs over NFS and is used to replicate essential system files.

- Hardware and software operating system heterogeneity:

– NFS has been implemented for almost every known operating system and hardware platform and is supported by a variety of

filling systems.

- Fault tolerance:

– Limited but effective; service is suspended if a server fails.

Recovery from failures is aided by the simple stateless design.

- Consistency:

– It provides a close approximation to one-copy semantics and meets the needs of the vast majority of applications.

– But the use of file sharing via NFS for communication or close coordination between processes on different computers cannot be recommended.

- Security:

– Recent developments include the option to use a secure RPC

implementation for authentication and the privacy and security of

the data transmitted with read and write operations.

– Efficiency:

 NFS protocols can be implemented for use in situations that

generate very heavy loads.

**Case Study: The Andrew File System (AFS)**

AFS differs markedly from NFS in its design and implementation. The differences are primarily attributable to the identification of scalability as the most important design goal. AFS is designed to perform well with larger numbers of active users than other distributed file systems. The key strategy for achieving scalability is the caching of whole files in client nodes. AFS has two unusual design characteristics:

*Whole-file serving*: The entire contents of directories and files are transmitted to client computers by AFS servers (in AFS-3, files larger than 64 kbytes are transferred in 64-kbyte chunks).

*Whole file caching*: Once a copy of a file or a chunk has been transferred to a client computer it is stored in a cache on the local disk. The cache contains several hundred of the files most recently used on that computer. The cache is permanent, surviving reboots of the client computer. Local copies of files are used to satisfy clients’ *open* requests in preference to remote copies whenever possible.

 Like NFS, AFS provides transparent access to remote shared files for UNIX programs running on workstations.

 AFS is implemented as two software components that exist at UNIX processes called

Vice and Venus**.**

**Scenario •** Here is a simple scenario illustrating the operation of AFS:

1. When a user process in a client computer issues an *open* system call for a file in the shared

*-*file space and there is not a current copy of the file in the local cache, the server holding the

file is located and is sent a request for a copy of the file.

2. The copy is stored in the local UNIX file system in the client computer. The copy is then

*open*ed and the resulting UNIX file descriptor is returned to the client.

3. Subsequent *read*, *write* and other operations on the file by processes in the client computer are applied to the local copy.

4. When the process in the client issues a *close* system call, if the local copy has been updated its contents are sent back to the server. The server updates the file contents and the timestamps on the file. The copy on the client’s local disk is retained in case it is needed again by a user-level process on the same workstation.

**Figure 11. Distribution of processes in the Andrew File System**

Workstations Servers

User Venus program

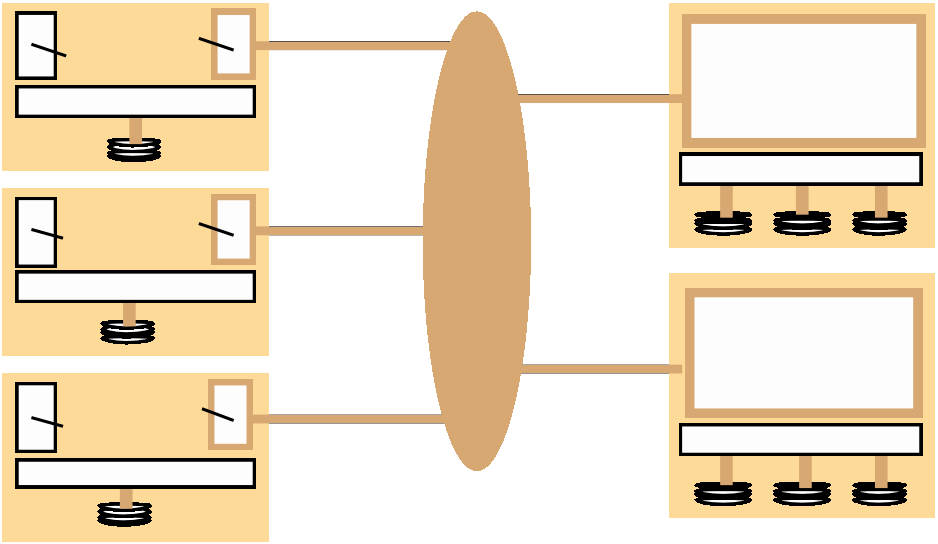
UNIX kernel

Vice

User Venus program

UNIX kernel

Network



UNIX kernel

Vice

User Venus program

UNIX kernel

UNIX kernel

 The files available to user processes running on workstations are either local or shared.

 Local files are handled as normal UNIX files.

 They are stored on the workstation’s disk and are available only to local user processes.

 Shared files are stored on servers, and copies of them are cached on the local disks of workstations.

 The name space seen by user processes is illustrated in Figure 12.

**Figure 12. File name space seen by clients of AFS**

/ (root)



tmp bin . . . vmunix cmu

bin

Symbolic links

Local Shared

 The UNIX kernel in each workstation and server is a modified version of BSD UNIX.

 The modifications are designed to intercept open, close and some other file system calls when

they refer to files in the shared name space and pass them to the Venus process in the client computer. (Figure 13)

**Figure 13. System call interception in AFS**

Workstation

User program

UNIX file systemcalls

Non-local file operations

Venus

UNIX kernel

Venus



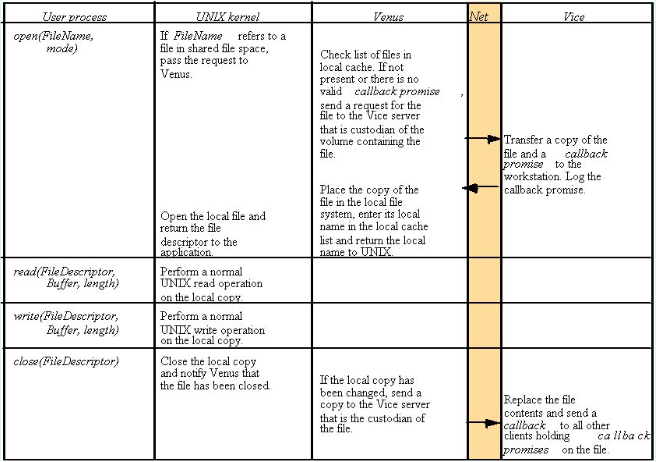
UNIXfilesystem

Local disk

Page | 142

 Figure 14 describes the actions taken by Vice, Venus and the UNIX kernel when a user process issues system calls.

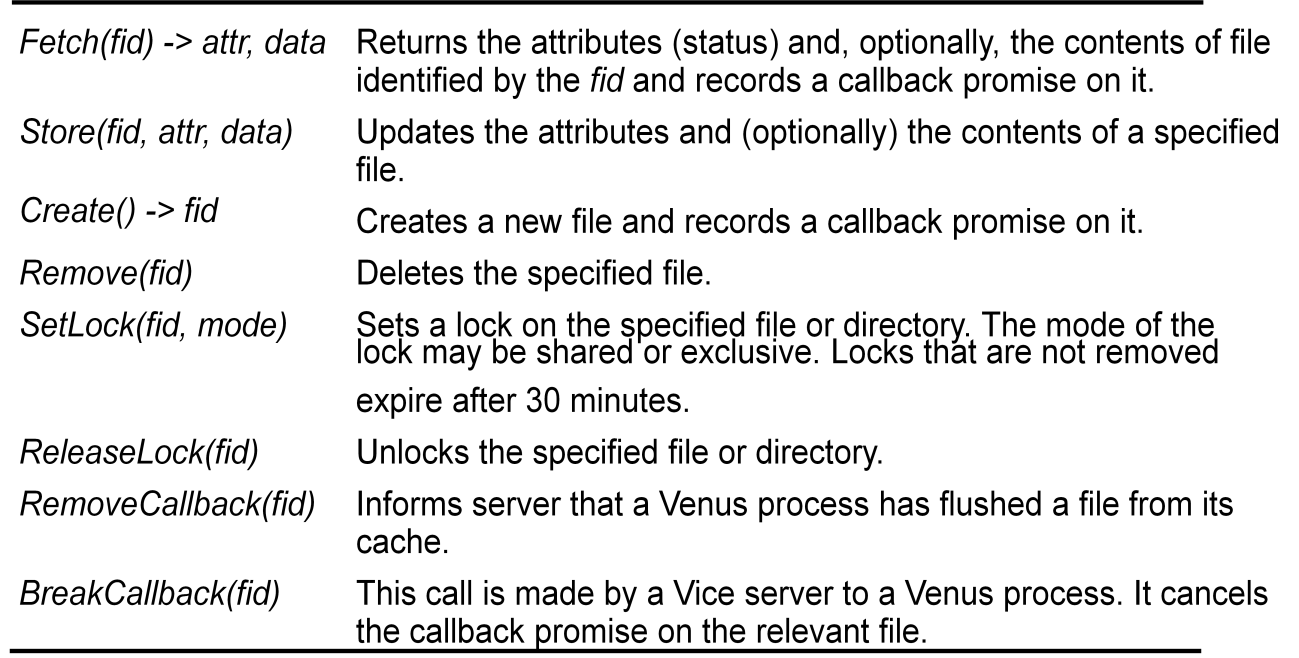
**Figure 14. implementation of file system calls in AFS**





Figure 15 shows the RPC calls provided by AFS servers for operations on files.

**sFigure 15. The main components of the Vice service interface**



Other aspects

AFS introduces several other interesting design developments and refinements that we outline here, together with a summary of performance evaluation results:

1. UNIX kernel modifications

2. Location database

3. Threads

4. Read-only replicas

5. Bulk transfers

6. Partial file caching

7. Performance

8. Wide area support