Distributed file system

Definition :

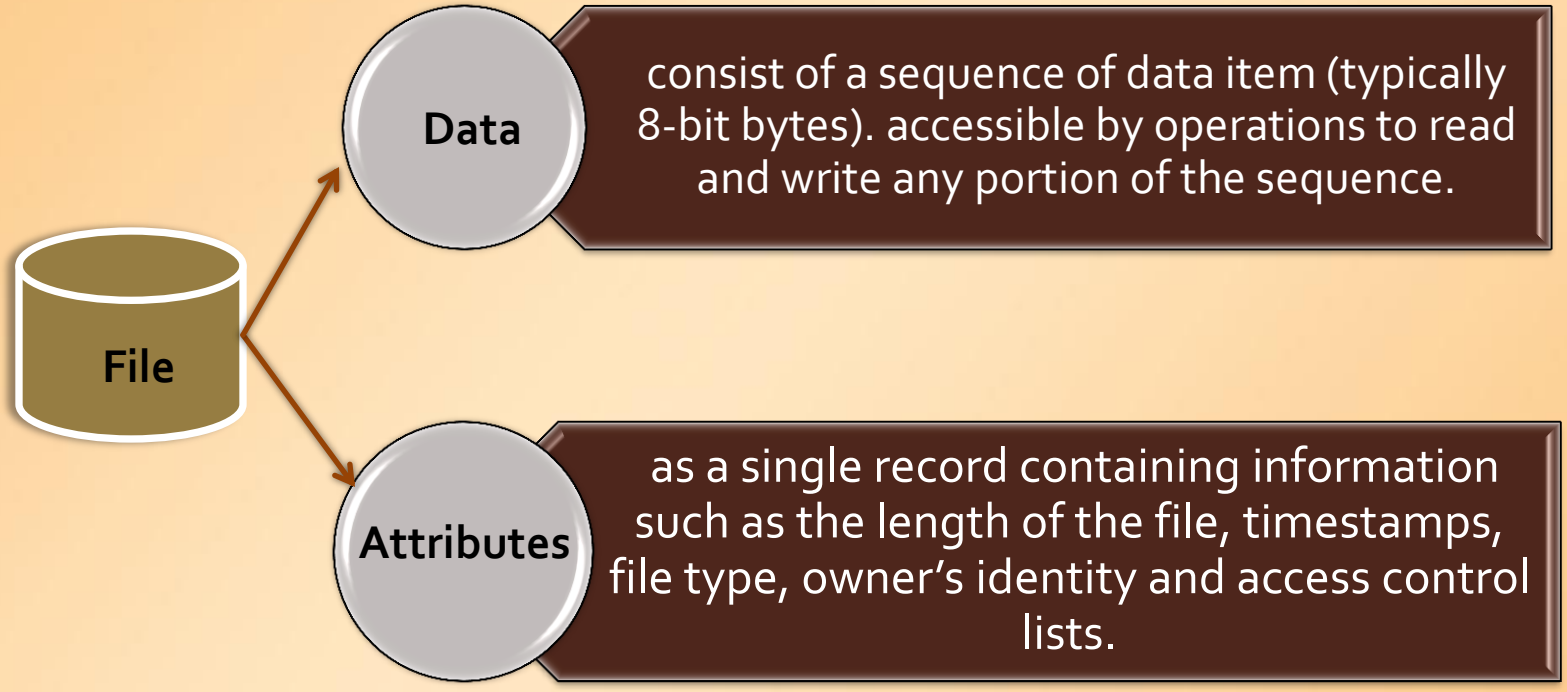
- distributed file system enables programs to store and access remote files as they do local ones.
- allowing users to access files from any computer on a network.

- The concentration of persistent storage at a few servers result :
  - reduces the need for local disk storage
  - enables economies to be made in the management and archiving of the persistent data owned by an organization. (more importantly)
  - Other services, such as the name service, the user authentication service and the print service, can be more easily implemented when they can call upon the file service to meet their needs for persistent storage

Characteristics of file system

- File systems are responsible for the organization, storage, retrieval, naming, sharing and protection of files.
- They provide a programming interface that freeing programmers from concern with the details of storage allocation
- Files are stored on disks or other non-volatile storage media.
- File systems also take responsibility for the control of access to files, restricting access to files according to users' authorizations.

Characteristics of file system



Distributed file system requirements

- Transparency
- Concurrency
- Replication
- Heterogeneity
- Fault tolerance
- Consistency
- Security
- Efficiency..

Goal for distributed file systems is usually performance comparable to local file system.

•The techniques used for the implementation of file services are an important part of the design of distributed systems.

part2

Distributed File service architecture

An architecture that offers a clear separation 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 flat file service and the directory service each export an interface for use by client programs, and their RPC interfaces, providing a set of operations for access to files.

❑Client module that perform operations for clients on directories and on files

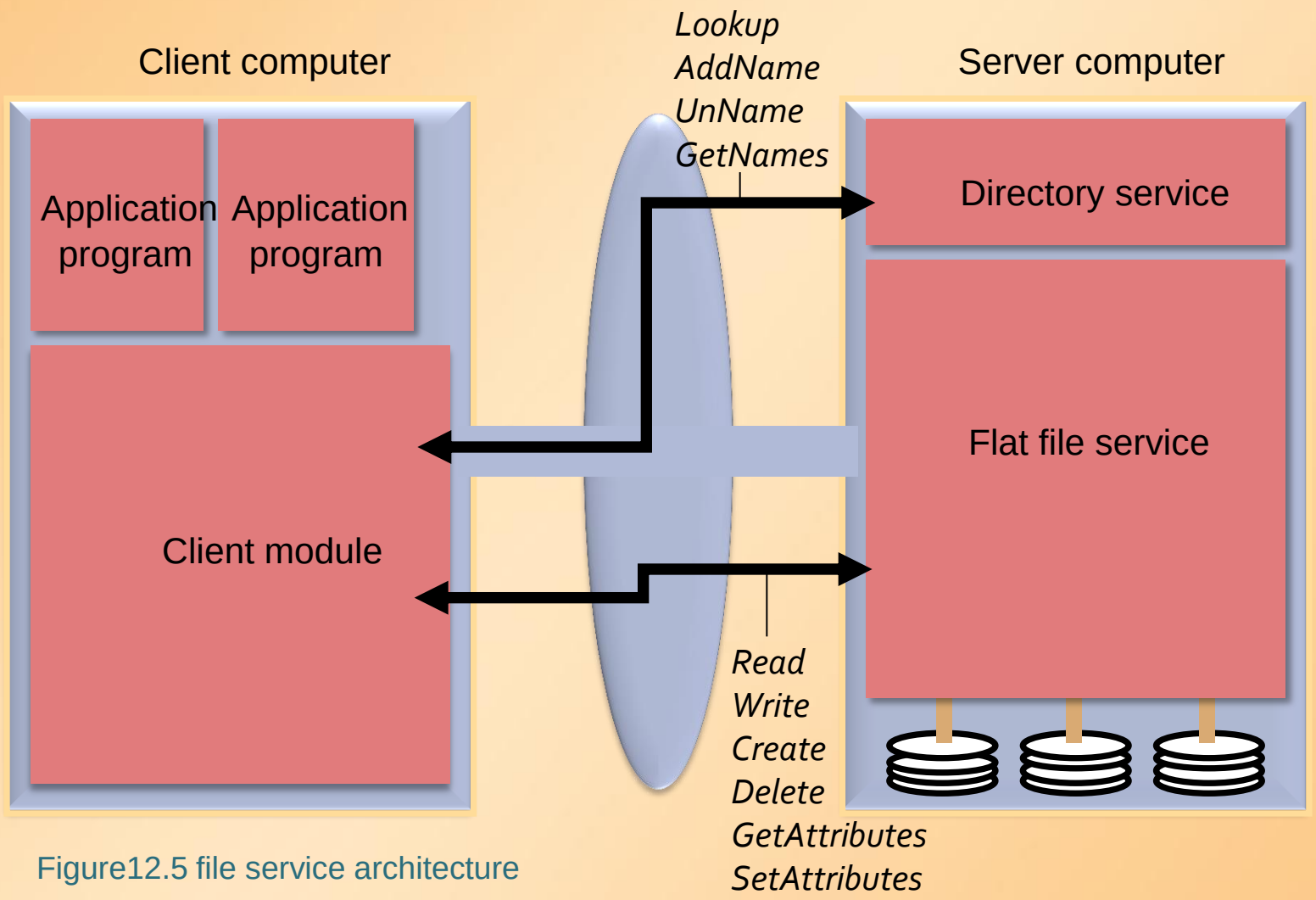


Figure12.5 file service architecture

Responsibilities of various modules

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 theirUFIDs.
- Clients may obtain the UFID of a file by quoting its text name to directory service.

Responsibilities of various modules

- Directory service supports functions needed generate directories, to add new files to directories.

Client Module:

- It runs on each computer and provides extended service (flat file and directory) as a single API to application programs
- It holds information about the network locations of flat-file and directory server processes.
- achieve better performance through implementation of a cache of recently used file blocks at the client.

Server operations/interfaces for the model file service

Flat file service

Read(FileId, i, n) -> Data

position of first byte

Write(FileId, i, , len,Data)

position of first byte

Create() -> FileId

Delete(FileId)

GetAttributes(FileId) -> Attr

SetAttributes(FileId, Attr)

Directory service

Lookup(Dir, Name) -> FileId

AddName(Dir, Name, FileId)

UnName(Dir, Name)

GetNames(Dir, Pattern) -> NameSeq

FileId

Contain an valid UFID with user' sufficient access rights

DFS: Case Studies

- **NFS (Network File System)**
  - Developed by Sun Microsystems (in 1985)
  - NFS was the first file service that was designed as a product.
  - Their design is operating system–independent
- **AFS (Andrew File System)**
  - Developed by Carnegie Mellon University as part of Andrew distributed computing environments (in 1986)
  - intention to support information sharing on a large scale by minimizing client-server communication
  - Public domain implementation is available on Linux (Linux AFS)

part3

Sun Network File System(NFS)

Sun NFS

- The NFS client and server modules communicate using remote procedure calls
- Supports many of the design requirements already mentioned:
  - transparency
  - heterogeneity
  - efficiency
  - fault tolerance

Limited achievement of:

  - concurrency
  - replication
  - consistency
  - security

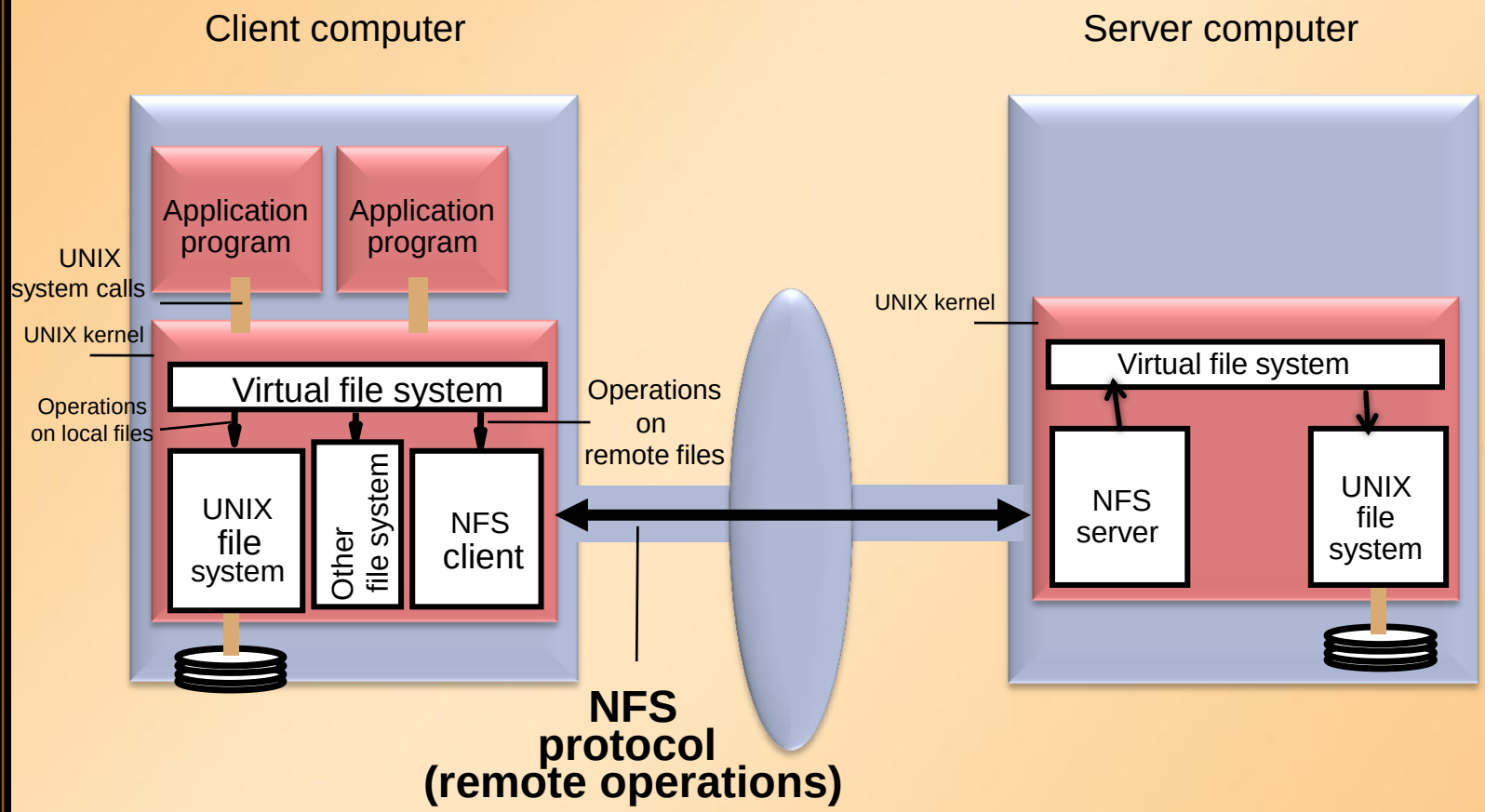


Figure 12.8.NFS architecture

Virtual file system

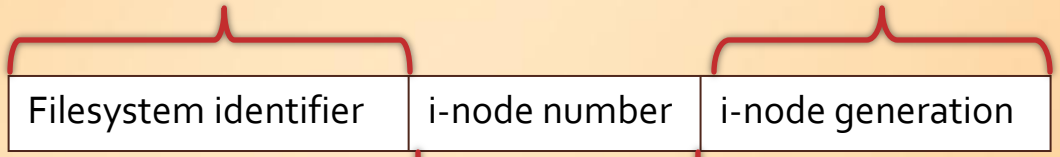
- The integration is achieved by a VFS module, which has been added to the UNIX kernel to distinguish between local and remote files.
- it passes each request to the appropriate local system module (the UNIX file system, the NFS client module or the service module for another file system).
- Translate between file identifiers used by NFS and the internal file identifiers normally used in UNIX and other file systems.

File handle

- The file identifiers used in NFS.
- A file handle is unclear to clients and contains whatever information the server needs .

a unique number that is allocated to each file system when it is created

is incremented each time the i-node number is reused



a number that serves to identify and locate the file in which the file is stored and are reused after a file is removed

NFS server operations

- read(fh, offset, count) -> attr, data
- write(fh, offset, count, data) -> attr
- create(dirfh, name, attr) -> newfh, attr
- remove(dirfh, name) status
- getattr(fh) -> attr
- setattr(fh, attr) -> attr
- lookup(dirfh, name) -> fh, attr
- rename(dirfh, name, todirfh, toname)
- mkdir(dirfh, name, attr) -> newfh, attr
- rmdir(dirfh, name) -> status
- statfs(fh) -> fsstats

Model flat file service

Read(FileId, i, n) -> Data
Write(FileId, i, Data)
Create() -> FileId
Delete(FileId)
GetAttributes(FileId) -> Attr
SetAttributes(FileId, Attr)

Model directory service

Lookup(Dir, Name) -> FileId
AddName(Dir, Name, Fileid)
UnName(Dir, Name)
GetNames(Dir, Pattern) ->NameSeq

NFS access control and authentication

- Stateless server, so the user's identity and authentication information (user ID and group ID) must be checked by the server on each request.
  - In the local file system they are checked only on open()
- The client can modify the RPC calls to include the user ID of any user(impersonating the user), unless the userID and groupID are protected by encryption
- Kerberos has been integrated with NFS to provide a stronger security solution.

Mount service

- The mounting of subtrees of remote filesystems by clients is supported by a separate mount service that runs at each NFS server.
- the well-known name (/etc/exports) containing the names of local filesystems that are available for remote mounting in server.
- Request mounting in operation:

  mount(remotehost, remotedirectory, localdirectory)
- Each client maintains a table of mounted file systems in NFS client and VFS layer,holding

  < IP address, port number, file handle>

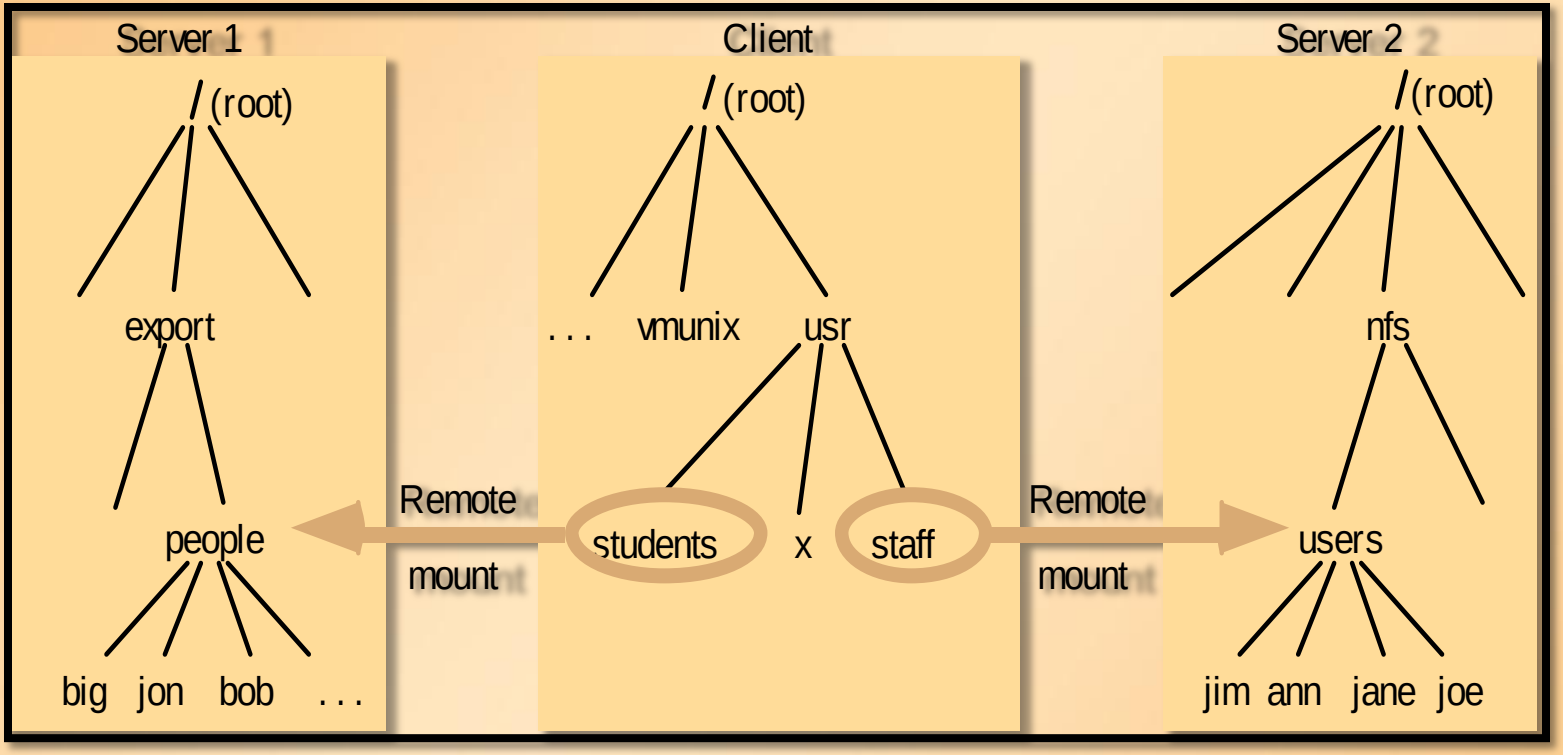


Figure 12.10 Local and remote filesystems accessible on an NFS client

the meaning of this is that programs running at Client can access files at Server 1 and Server 2 by using pathnames such as /usr/students/jon and /usr/staff/ann.

Securing NFS with Kerberos

- Kerberos protocol is too costly to apply on each file access request
- Kerberos is used in the mount service:
  - to authenticate the user's identity
  - User's UserID and GroupID are stored at the server with the client's IP address
- For each file request:
  - The UserID and GroupID sent must match those stored at the server
  - IP addresses must also match
- This approach has some problems
  - all remote filestores must be mounted each time a user logs in

NFS optimization - server caching

- 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
- To guard against loss of data in a system crash, the UNIX *sync* operation flushes altered pages to disk every 30 seconds.

Works well in local context, but in the remote case extra measures are needed to ensure that clients can be confident that the results of the write operations are persistent, even when server crashes occur.

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 reply is sent, 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. A commit() is issued by the client whenever a file is closed.

NFS optimization - client caching

- Server caching does nothing to reduce RPC traffic between client and server.
  - 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.
  - Instead, clients are responsible for polling the server to check the currency of the cached data that they hold.

Timestamp-based to validate cached blocks before use:

A cache entry is valid at time T if this statement is true

$$(T - Tc < t) \text{ v } (Tmclient = Tmserver)$$

- **t** is configurable (per file) but is typically set to 3 seconds for files and 30 secs for directories.
- There is one value of Tmserver for all the data blocks in a file and another for the file attributes.
- if the first part is false, the current value of Tmserver is obtained (by a getattr call to the server)

t	freshness interval
Tc	time when cache entry was last validated
Tm	time when block was last updated
T	current time

NFS summary

- 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.
- Single-CPU implementations based on PC hardware achieve throughputs in excess of 12,000 server ops/sec
- large multi-processor configurations with many disks achieved throughputs of up to 300,000 server ops/sec.

NFS summary

- An excellent example of a simple, high-performance distributed service.
- Achievement of transparencies:

Access: *Excellent;* the UNIX system call interface for both local and remote files. No modifications to existing programs are required to enable them to operate correctly with remote files.

Location: *Not guaranteed* but normally achieved; naming of filesystems is controlled by client mount operations, have different pathnames on different clients;but transparency can be ensured by an appropriate system configuration.

Mobility: *Hardly achieved;* Filesystems may be moved between servers, but the remote mount tables in each client must then be updated separately to enable the clients to access the filesystems in their new locations

Replication: *Limited* to read-only file systems; for writable files on several server, the SUN Network Information Service (NIS) separately runs over NFS and is used to replicate essential system files.

Scaling: *Good;* NFS servers can be built to handle very large real-world loads in an efficient manner. The performance of a single server can be increased by the addition of processors, disks

When the limits of that process are reached, additional servers must be installed and the filesystems must be reallocated between them that need to support replication

Concurrency: *Limited* when read-write files are shared concurrently between clients, consistency is not perfect.

Fault tolerance: *Limited but effective*; service is suspended if a server fails. but once it has been restarted user-level client processes proceed from the point at which the service was interrupted, unaware of the failure.(except in soft-mounted

Security: The integration of Kerberos with NFS was a major step forward. Recent developments include the option to use a secure RPC implementation for authentication of the data transmitted with read and write operations.

Efficiency: *Good;* The measured performance of several implementations show that NFS protocols can be implemented for use in situations that generate very heavy loads.

Summery

- Distributed File systems provide illusion of a local file system and hide complexity from end users.
- Sun NFS is an excellent example of a distributed service designed to meet many important design requirements
- Effective client caching can produce file service performance equal to or better than local file systems
- Superior scalability can be achieved with whole-file serving (Andrew FS)

Future requirements:

- support for mobile users.
- Full Replication
- support for data streaming and video file server