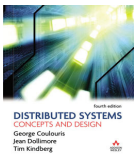
Slides for Chapter 8: Distributed File Systems



From **Coulouris, Dollimore and Kindberg**
Distributed Systems: Concepts and Design
Edition 4, © Pearson Education 2005

Learning Objectives

- Understand the requirements that affect the design of distributed storage services
- Case study on NFS: understand how a relatively simple, widely used service is designed

Distributed storage systems

- Earlier storage systems are file systems (e.g. NFS); units are files.
- More recently, distributed object systems (e.g. CORBA, Java); units are objects.

Storage systems and their properties

	Sharing	*Persis-tence*	*Distributed cache/replicas*	*Consistency maintenance*	*Example*
Main memory	×	×	×	1	RAM
File system	×	✓	×	1	UNIX file system
Distributed file system	✓	✓	✓	✓	Sun NFS
Web	✓	✓	✓	×	Web server
Distributed shared memory	✓	×	✓	✓	Ivy (DSM, Ch. 18)
Remote objects (RMI/ORB)	✓	×	×	1	CORBA
Persistent object store	✓	✓	×	1	CORBA Persistent Object Service
Peer-to-peer storage system	✓	✓	✓	2	OceanStore (Ch. 10)

Types of consistency:
1: strict one-copy. :✓ slightly weaker guarantees. 2: considerably weaker guarantees.

Characteristics of (non-distributed) file systems

- data and attributes (Fig 8.3)
- directory: mapping from text names to internal file identifiers
- layers of modules in file systems (Fig 8.2)
- file operation system calls in UNIX (Fig. 8.4)

File attributes

File length
Creation timestamp
Read timestamp
Write timestamp
Attribute timestamp
Reference count
Owner
File type
Access control list

File system modules

Directory module:	relates file names to file IDs
File module:	relates file IDs to particular files
Access control module:	checks permission for operation requested
File access module:	reads or writes file data or attributes
Block module:	accesses and allocates disk blocks
Device module:	disk I/O and buffering

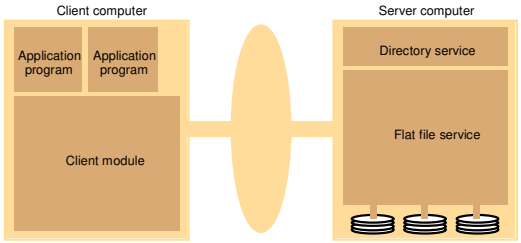
UNIX file system operations

filedes = open(name, mode) *filedes = creat(name, mode)*	Opens an existing file with the given *name*. Creates a new file with the given *name*. Both operations deliver a file descriptor referencing the open file. The *mode* is *read*, *write* or both.
status = close(filedes)	Closes the open file *filedes*.
count = read(filedes, buffer, n) *count = write(filedes, buffer, n)*	Transfers *n* bytes from the file referenced by *filedes* to *buffer*. Transfers *n* bytes to the file referenced by *filedes* from buffer. Both operations deliver the number of bytes actually transferred and advance the read-write pointer.
pos = lseek(filedes, offset, whence)	Moves the read-write pointer to offset (relative or absolute, depending on *whence*).
status = unlink(name)	Removes the file *name* from the directory structure. If the file has no other names, it is deleted.
status = link(name1, name2)	Adds a new name (*name2*) for a file (*name1*).
status = stat(name, buffer)	Gets the file attributes for file *name* into *buffer*.

Distributed file system requirements

- transparency:
  - access
  - location
  - mobility
  - performance
  - scaling
- concurrent file updates
- file replication
- consistency
- fault tolerance
- hardware and os heterogeneity
- security
- efficiency

File service architecture (Author's model)



File service architecture

- Flat file service
  - unique file identifiers (UFID)
- Directory service
  - map names to UFIDs
- Client module
  - integrate/extend flat file and directory services
  - provide a common application programming interface (can emulate different file interfaces)
  - stores location of flat file and directory services

Flat file service interface

- RPC used by client modules
  - not by user-level programs (which use client modules)
- More fault tolerant compared to UNIX
  - Repeatable (idempotent) operations
    - [except for Create: re-execution gets a new file]
    - at-least-once semantics
    - no open (hence close) so no state to remember
    - specify starting location and UFID (from directory service) in Read/Write
  - Stateless server
    - Can be restarted without the server or client restoring any state information

Flat file service operations

Read(FileId, i, n) -> Data — throws *BadPosition*	If $1 \leq i \leq Length(File)$: Reads a sequence of up to *n* items from a file starting at item *i* and returns it in *Data*.
Write(FileId, i, Data) — throws *BadPosition*	If $1 \leq i \leq Length(File)+1$: Writes a sequence of *Data* to a file, starting at item *i*, extending the file if necessary.
Create() -> FileId	Creates a new file of length 0 and delivers a UFID for it.
Delete(FileId)	Removes the file from the file store.
GetAttributes(FileId) -> Attr	Returns the file attributes for the file.
SetAttributes(FileId, Attr)	Sets the file attributes (only those attributes that are not shaded in Figure 8.3).

Access control

- UNIX checks access rights when a file is opened
  - subsequent checks during read/write are not necessary
- distributed environment
  - server has to check
  - stateless approaches
    1. access check once when UFID is issued
       - client gets an encoded "capability" (who can access and how)
       - capability is submitted with each subsequent request
    2. access check for each request.
  - second is more common

Directory service operations

Lookup(Dir, Name) -> FileId — throws *NotFound*	Locates the text name in the directory and returns the relevant UFID. If *Name* is not in the directory, throws an exception.
AddName(Dir, Name, FileId) — throws *NameDuplicate*	If *Name* is not in the directory, adds (*Name*, *File*) to the directory and updates the file's attribute record. If *Name* is already in the directory: throws an exception.
UnName(Dir, Name) — throws *NotFound*	If *Name* is in the directory: the entry containing *Name* is removed from the directory. If *Name* is not in the directory: throws an exception.
GetNames(Dir, Pattern) -> NameSeq	Returns all the text names in the directory that match the regular expression *Pattern*.

Hierarchical file system

- Directories containing other directories and files
- Each file can have more than one name (pathnames)
  - how in UNIX, Windows?

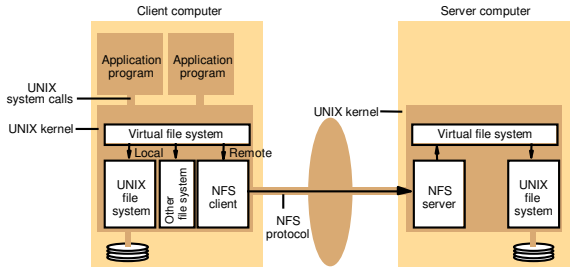
File groups

- a logical collection of files on one server
  - a server can have more than one group
  - a group can change server
  - filesystems in unix
  - different devices for non-distributed
  - different hosts for distributed

Case Study: Sun NFS

- Industry standard for local networks since the 1980's
- OS independent
- unix implementation
- rpc
- udp or tcp

NFS architecture: virtual file system



Virtual file system

- access transparency
- part of unix kernel
- NFS *file handle*, 3 components:
  - filesystem identifier
    - different groups of files
  - i-node (index node)
    - structure for finding the file
  - i-node generation number
    - i-nodes are reused
    - incremented when reused
- VFS
  - struct for each file system
  - v-node for each open file
    - *file handle* for remote file
    - i-node number for local file

Client integration

- nfs client emulates Unix file semantics
- in the kernel, not in a library, because:
  - access files via system calls
  - single client module for multiple user processes
  - encryption can be done in the kernel

Access control

- nfs server is stateless, doesn't keep open files for clients
- server checks identity each time (uid and gid)

NFS server operations (simplified)

What do you notice about read() and write()?

lookup(dirfh, name) -> fh, attr	Returns file handle and attributes for the file *name* in the directory *dirfh*.
create(dirfh, name, attr) -> newfh, attr	Creates a new file name in directory *dirfh* with attributes *attr* and returns the new file handle and attributes.
remove(dirfh, name) status	Removes file name from directory *dirfh*.
getattr(fh) -> attr	Returns file attributes of file *fh*. (Similar to the UNIX *stat* system call.)
setattr(fh, attr) -> attr	Sets the attributes (mode, user id, group id, size, access time and modify time of a file). Setting the size to 0 truncates the file.
read(fh, offset, count) -> attr, data	Returns up to *count* bytes of data from a file starting at *offset*. Also returns the latest attributes of the file.
write(fh, offset, count, data) -> attr	Writes *count* bytes of data to a file starting at *offset*. Returns the attributes of the file after the write has taken place.
rename(dirfh, name, todirfh, toname) -> status	Changes the name of file *name* in directory *dirfh* to *toname* in directory to *todirfh*
link(newdirfh, newname, dirfh, name) -> status	Creates an entry *newname* in the directory *newdirfh* which refers to file *name* in the directory *dirfh*.

Continues on next slide ...

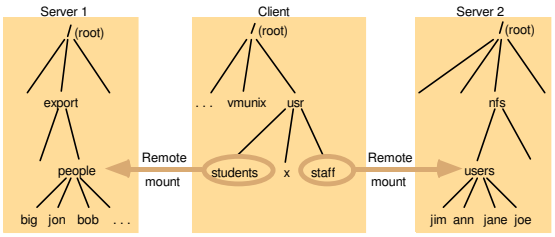
NFS server operations (simplified) – 2

symlink(newdirfh, newname, string) -> status	Creates an entry *newname* in the directory *newdirfh* of type symbolic link with the value *string*. The server does not interpret the *string* but makes a symbolic link file to hold it.
readlink(fh) -> string	Returns the string that is associated with the symbolic link file identified by *fh*.
mkdir(dirfh, name, attr) -> newfh, attr	Creates a new directory *name* with attributes *attr* and returns the new file handle and attributes.
rmdir(dirfh, name) -> status	Removes the empty directory *name* from the parent directory *dirfh*. Fails if the directory is not empty.
readdir(dirfh, cookie, count) -> entries	Returns up to *count* bytes of directory entries from the directory *dirfh*. Each entry contains a file name, a file handle, and an opaque pointer to the next directory entry, called a *cookie*. The *cookie* is used in subsequent *readdir* calls to start reading from the following entry. If the value of *cookie* is 0, reads from the first entry in the directory.
statfs(fh) -> fsstats	Returns file system information (such as block size, number of free blocks and so on) for the file system containing a file *fh*.

Mount service

- the process of including a new filesystem is called mounting
- /etc/exports has filesystems that can be mounted by others
- clients use a modified mount command for remote filesystems
- communicates with the mount process on the server in a mount protocol
- hard-mounted
  - user process is suspended until request is successful
  - when server is not responding
  - request is retried until it's satisfied
- soft-mounted
  - if server fails, client returns failure after a small # of retries
  - user process handles the failure

Local and remote file systems



Note: The file system mounted at */usr/students* in the client is actually the sub-tree located at */export/people* in Server 1; the file system mounted at */usr/staff* in the client is actually the sub-tree located at */nfs/users* in Server 2.

Pathname translation

- pathname: /users/students/dc/abc
- server doesn't receive the entire pathname for translation, why?
- client breaks down the pathnames into parts
- iteratively translate each part
- translation is cached

Automounter

- what if a user process reference a file on a remote filesystem that is not mounted
- table of mount points (pathname) and servers
- NFS client sends the reference to the automounter
- automounter check find the first server that is up
- mount it at some location and set a symbolic link (original impl)
- mount it at the mount point (later impl)
- could help fault tolerance, the same mount point with multiple replicated servers.

Server caching

- caching file pages, directory/file attributes
- read-ahead: prefetch pages following the most-recently read file pages
- delayed-write: write to disk when the page in memory is needed for other purposes
- "sync" flushes "dirty" pages to disk every 30 seconds
- two write option
  1. write-through: write to disk before replying to the client
  2. cache and commit:
     - stored in memory cache
     - write to disk before replying to a "commit" request from the client

Client caching

- caches results of read, write, getattr, lookup, readdir
- clients responsibility to poll the server for consistency

Client caching: reading (1)

- timestamp-based methods for consistency validation
  - *Tc*: time when the cache entry was last validated
  - *Tm*: time when the block was last modified at the server
- cache entry is valid if:
  1. $T - Tc < t$, where t is the freshness interval
     - t is adaptively adjusted:
       - files: 3 to 30 seconds depending on freq of updates
       - directories: 30 to 60 seconds
  2. $Tm_{client} = Tm_{server}$

Client caching: reading (2)

- need validation for all cache accesses
- condition #1 can be determined by the client alone--performed first
- Reducing getattr() to the server [for getting Tm_{server}]
  1. new value of Tm_{server} is received, apply to all cache entries from the same file
  2. piggyback getattr() on file operations
  3. adaptive alg for update t (condition #1)
- validation doesn't guarantee the same level of consistency as one-copy

Client caching: writing

- dirty: modified page in cache
- flush to disk: file is closed or sync from client
- bio-daemon (block input-output)
  - read-ahead: after each read request, request the next file block from the server as well
  - delayed write: after a block is filled, it's sent to the server
  - reduce the time to wait for read/write

Other optimization

- UDP, RPC
- extended to 9 kilobytes--entire block in a single packet

Security

- stateless nfs server
- user's identity in each request
- Kerberos authentication during the mount process, which includes uid and host address
- server maintain authentication info for the mount
- on each file request, nfs checks the uid and address
- one user per client

Performance

- overhead/penalty is low
- main problems
  - frequent getattr() for cache validation (piggybacking)
  - relatively poor performance if write-through is used on the server (delay-write/commit in current versions)
- write < 5%
- lookup is almost 50% (step by step pathname translation)

Summary for NFS

- access transparency: same system calls for local or remote files
- location transparency: could have a single name space for all files (depending on all the clients to agree the same name space)
- mobility transparency: mount table need to be updated on each client (not transparent)
- scalability: can usually support large loads, add processors, disks, servers...
- file replication: read-only replication, no support for replication of files with updates
- hardware and OS: many ports
- fault tolerance: stateless and idempotent
- consistency: not quite one-copy for efficiency
- security: added encryption--Kerberos
- efficiency: pretty efficient, wide-spread use

WebNFS (p. 360)

- HTTP and NFS both can read files, what NFS can do more than HTTP in terms of reading? [what can read() do in a file system that HTTP can't?]
- NFS is designed to be on "fast" LANs
- WebNFS is designed to be on "slower" WANs
- WebNFS clients talk to NFS servers
  - Small set up cost (thiner client)
    - public files, mostly read access, no authentication
    - no mounting
  - access portions of a file
- nfs://xyz.com/someFile