Distributed-File Systems

- Background
- Naming and Transparency
- Stateful versus Stateless Service
- NFS
- AFS

1

Background

- Distributed file system – a distributed implementation of a file system, where multiple users share files and storage resources.
- A DFS manages set of dispersed storage devices
- Overall storage space managed by a DFS is composed of different, remotely located, smaller storage spaces.
- There is usually a correspondence between constituent storage spaces and sets of files.

2

DFS Structure

- *Service* – software entity running on one or more machines and providing a particular type of function to a priori unknown clients.
- *Server* – service software running on a single machine.
- *Client* – process that can invoke a service using a set of operations that forms its *client interface.*
- A client interface for a file service is formed by a set of primitive *file operations* (create, delete, read, write).

3

Naming and Transparency

- *Naming* – mapping between logical and physical objects
- Ideally, client interface should be transparent, i.e., not distinguish between local and remote files
  - In practice, this is not always possible
  - More complicated failure modes, different design goals sometimes motivate a different interface
- A *transparent* DFS hides the location where in the network the file is stored.
  - There is a binding from file f to server s
  - Either static or dynamic
  - S is possibly a set, for replicated files

4

Naming Structures

- **Location transparency** – file name does not reveal the file's physical storage location.
  - File name still denotes a specific, although hidden, set of physical disk blocks.
  - Convenient way to share data.
  - Can expose correspondence between component units and machines.
- **Location independence** – file name does not need to be changed when the file's physical storage location changes.
  - Better file abstraction.
  - Promotes sharing the storage space itself.
  - Separates the naming hierarchy form the storage-devices hierarchy.

Naming Schemes — Three Main Approaches

- Files named by combination of their host name and local name; guarantees a unique systemwide name.
- Attach remote directories to local directories, giving the appearance of a coherent directory tree; only previously mounted remote directories can be accessed transparently
- Total integration of the component file systems.
  - A single global name structure spans all the files in the system.
  - If a server is unavailable, some arbitrary set of directories on different machines also becomes unavailable. .

Caching

- Reduce network traffic by retaining recently accessed disk blocks in a cache, so that repeated accesses to the same information can be handled locally.
  - If needed data not already cached, a copy of data is brought from the server to the user.
  - Accesses are performed on the cached copy.
  - Files identified with one master copy residing at the server machine, but copies of (parts of) the file ar scattered in different caches.
  - *Cache-consistency* problem – keeping the cached copies consistent with the master file.

Location – Disk Caches vs. Main Memory Cache

- Advantages of disk caches
  - More reliable.
  - Cached data kept on disk are still there during recovery and don't need to be fetched again.
- Advantages of main-memory caches:
  - Permit workstations to be diskless.
  - Data can be accessed more quickly.
  - Performance speedup in bigger memories.
  - Server caches (used to speed up disk I/O) are in main memory regardless of where user caches are located; using main-memory caches on the user machine permits a single caching mechanism for servers and users.

Cache Placement

- Two locations for a cache
  - In the client
  - In the server
- Client caches can reduce network traffic:
  - Read-only operations on unchanged files do not need to go over the network
- Server caches can reduce server load:
  - Cache is amortized across all clients (but needs to be bigger to be effective)
- In practice, need both kinds of caches

Cache Update Policy

- *Write-through* – write data through to disk as soon as they are placed on any cache. Reliable, but poor performance.
- *Delayed-write* – modifications written to the cache and then written through to the server later. Write accesses complete quickly; some data may be overwritten before they are written back, and so need never be written at all.
  - Poor reliability; unwritten data will be lost whenever a user machine crashes.
  - Variation – scan cache at regular intervals and flush blocks that have been modified since the last scan.
  - Variation – *write-on-close*, writes data back to the server when the file is closed. Best for files that are open for long periods and frequently modified.

Consistency

- Is locally cached copy of the data consistent with the master copy?
- Client-initiated approach
  - Client initiates a validity check.
  - Server checks whether the local data are consistent with the master copy.
- Server-initiated approach
  - Server records, for each client, the (parts of) files it caches.
  - When server detects a potential inconsistency, it must react.

Stateful File Service

- Mechanism.
  - Client opens a file.
  - Server fetches information about the file from its disk, stores it in its memory, and gives the client a connection identifier unique to the client and the open file.
  - Identifier is used for subsequent accesses until the session ends.
- Increased performance.
  - Fewer disk accesses
  - Stateful server knows if a file was opened for sequential access and can thus read ahead the next blocks
  - RPCs are small, contain only an identifier
  - File may be cached entirely on the client, invalidated by the server if there is a conflicting write

Stateless File Server

- Avoids state information by making each request self-contained.
- Each request identifies the file and position in the file.
- No need to establish and terminate a connection by open and close operations
- Advantage:
  - A fileserver crash does not affect any clients
- Disadvantages:
  - RPCs need to contain all state associated with the operation

13

Distinctions Between Stateful & Stateless Service

- Failure Recovery.
  - A stateful server loses all its volatile state in a crash.
    - Restore state by recovery protocol based on a dialog with clients, or abort operations that were underway when the crash occurred.
    - Server needs to be aware of client failures in order to reclaim space allocated to record the state of crashed client processes (orphan detection and elimination).
  - With stateless server, the effects of server failure and recovery are almost unnoticeable. A newly reincarnated server can respond to a self-contained request without any difficulty.

14

Distinctions (Cont.)

- Penalties for using the robust stateless service:
  - longer request messages
  - slower request processing
  - additional constraints imposed on DFS design
- Some environments require stateful service.
  - A server employing server-initiated cache validation cannot provide stateless service, since it maintains a record of which files are cached by which clients.
  - UNIX use of file descriptors and implicit offsets is inherently stateful; servers must maintain tables to map the file descriptors to inodes, and store the current offset within a file.

15

File Replication

- Replicas of the same file reside on failure-independent machines.
- Improves availability and can shorten service time.
- Naming scheme maps a replicated file name to a particular replica.
  - Existence of replicas should be invisible to higher levels.
  - Replicas must be distinguished from one another by different lower-level names.
- Updates – replicas of a file denote the same logical entity, and thus an update to any replica must be reflected on all other replicas.
- Demand replication – reading a nonlocal replica causes it to be cached locally, thereby generating a new nonprimary replica.

16

The Sun Network File System (NFS)

- An implementation and a specification of a software system for accessing remote files across LANs (or WANs).
- Built on top of an unreliable datagram protocol (UDP/IP)

NFS (Cont.)

- Client-server model
  - A remote directory is mounted over a local file system directory. The mounted directory looks like an integral subtree of the local file system, replacing the subtree descending from the local directory
  - Specification of the remote directory for the mount operation is nontransparent; the host name of the remote directory has to be provided. Files in the remote directory can then be accessed in a transparent manner
  - Subject to access-rights accreditation, potentially any file system (or directory within a file system), can be mounted remotely on top of any local directory

NFS (Cont.)

- NFS is designed to operate in a heterogeneous environment of different machines, operating systems, and network architectures; the NFS specifications independent of these media.
- This independence is achieved through the use of RPC primitives built on top of an External Data Representation (XDR) protocol used between two implementation-independent interfaces.

NFS Mount Protocol

- Establishes initial logical connection between server and client.
- Mount operation includes name of remote directory to be mounted and name of server machine storing it.
  - Mount request is mapped to corresponding RPC and forwarded to mount server running on server machine.
  - *Export list* – specifies local file systems that server exports for mounting, along with names of machines that are permitted to mount them.
- Following a mount request that conforms to its export list, the server returns a *filesystem handle*—a key for further accesses.
- Filesystem handle – a file-system identifier, and an inode number to identify the mounted directory within the exported file system.
- The mount operation changes only the user's view and does not affect the server side.

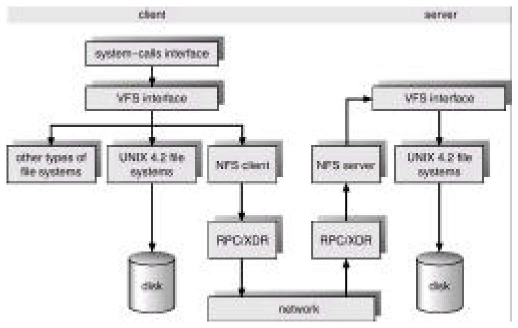
NFS Protocol

- Provides a set of remote procedure calls for remote file operations. The procedures support the following operations:
  – searching for a file within a directory
  – reading a set of directory entries
  – manipulating links and directories
  – accessing file attributes
  – reading and writing blocks within files
- NFS servers are *stateless*; each request has to provide a full set of arguments.
- Modified data must be committed to the server's disk before results are returned to the client (lose advantages of caching).
- The NFS protocol does not provide concurrency-control mechanisms.

Three Major Layers of NFS Architecture

- UNIX file-system interface (based on the open, read, write, and close calls, and file descriptors).
- *Virtual File System* (VFS) layer – distinguishes local files from remote ones, and local files are further distinguished according to their file-system types.
  – The VFS activates file-system-specific operations to handle local requests according to their file-system types.
  – Calls the NFS protocol procedures for remote requests.
- NFS service layer – bottom layer of the architecture; implements the NFS protocol.

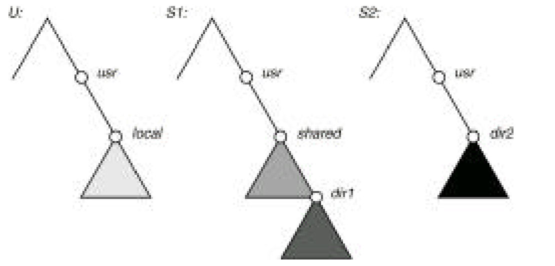
Schematic View of NFS Architecture



NFS Path-Name Translation

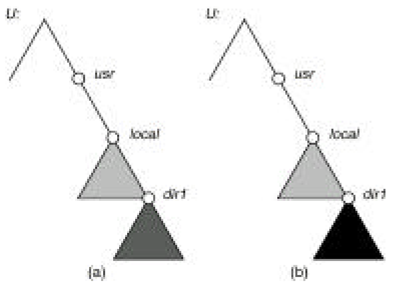
- Performed by breaking the path into component names and performing a separate NFS lookup call for every pair of component name and directory vnode.
- To make lookup faster, a directory name lookup cache on the client's side holds the vnodes for remote directory names.

Three Independent File Systems



25

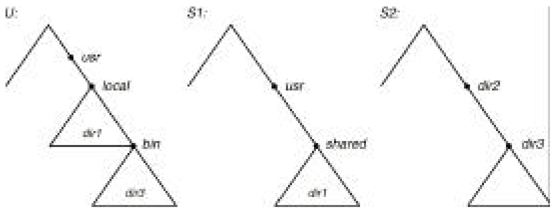
Mounting in NFS



Mounts Cascading mounts

26

Path-name Translation



27

NFS Remote Operations

- Nearly one-to-one correspondence between regular UNIX system calls and the NFS protocol RPCs (except opening and closing files).
- NFS adheres to the remote-service paradigm, but employs buffering and caching techniques for the sake of performance.
- File-blocks cache – when a file is opened, the kernel checks with the remote server whether to fetch or revalidate the cached attributes. Cached file blocks are used only if the corresponding cached attributes are up to date.
- File-attribute cache – the attribute cache is updated whenever new attributes arrive from the server.
- Clients do not free delayed-write blocks until the server confirms that the data have been written to disk.

28

NFS and Locking

- File locks are a useful abstraction
  - Consider mail delivery
- Impossible to implement locks in a stateless way
  - The whole point of a lock is to have some state that protects the file in question
  - NFS makes an attempt
  - Cannot offer strong guarantees
  - Implementation was always ‘buggy’ – a euphemism

29

ANDREW Filesystem

- Andrew filesystem (AFS) is designed to be highly scalable
  - The system is designed to be able to name and access all AFS servers in the world
- Client-server model
- Simple interface
  - GET file
  - PUT file
  - Other calls for manipulating access controls, volumes, etc.
- Whole file caching is the central idea behind AFS
  - Later amended with block operations
  - Simple, effective
- AFS is stateful
  - Servers keep track of which clients have which files
  - Recall files when they have been modified

30

ANDREW (Cont.)

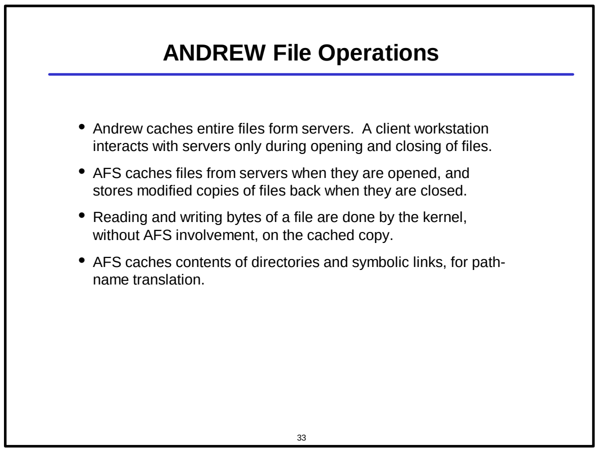
- Dedicated servers present an homogeneous, identical, and location transparent file hierarchy to clients
- Clients are required to have local disks where they store
  - their local files
  - the result of GET operations

31

ANDREW Shared Name Space

- Servers arrange storage in logical volumes,
- Files and directories are named by an fid. A fid identifies a file or directory. A fid is 96 bits long and has three equal-length components:
  - volume number
  - vnode number – index into an array containing the inodes of files in a single volume.
  - uniquifier – allows reuse of vnode numbers, thereby keeping certain data structures, compact.
- Fids are location transparent; therefore, file movements from server to server do not invalidate cached directory contents.
- Location information is kept on a volume basis, and the information is replicated on each server.

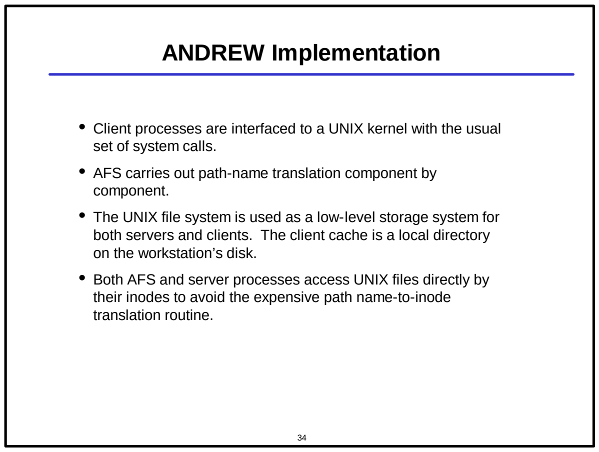
32



ANDREW File Operations

- Andrew caches entire files form servers. A client workstation interacts with servers only during opening and closing of files.
- AFS caches files from servers when they are opened, and stores modified copies of files back when they are closed.
- Reading and writing bytes of a file are done by the kernel, without AFS involvement, on the cached copy.
- AFS caches contents of directories and symbolic links, for path-name translation.

33



ANDREW Implementation

- Client processes are interfaced to a UNIX kernel with the usual set of system calls.
- AFS carries out path-name translation component by component.
- The UNIX file system is used as a low-level storage system for both servers and clients. The client cache is a local directory on the workstation's disk.
- Both AFS and server processes access UNIX files directly by their inodes to avoid the expensive path name-to-inode translation routine.

34