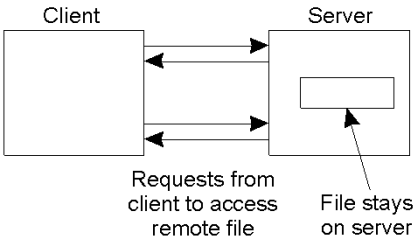
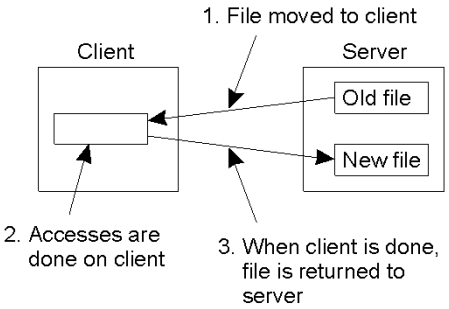
Today: Distributed File Systems

- Issues in distributed file systems
- Case studies:
  - NFS
  - Coda
  - xFS + RAID
  - Log-structured file systems (time permitting)

Distributed File Service





- Remote access model
  - Work done at the server
- Stateful server (e.g., databases)
- Consistent sharing (+)
- Server may be a bottleneck (-)
- Need for communication (-)

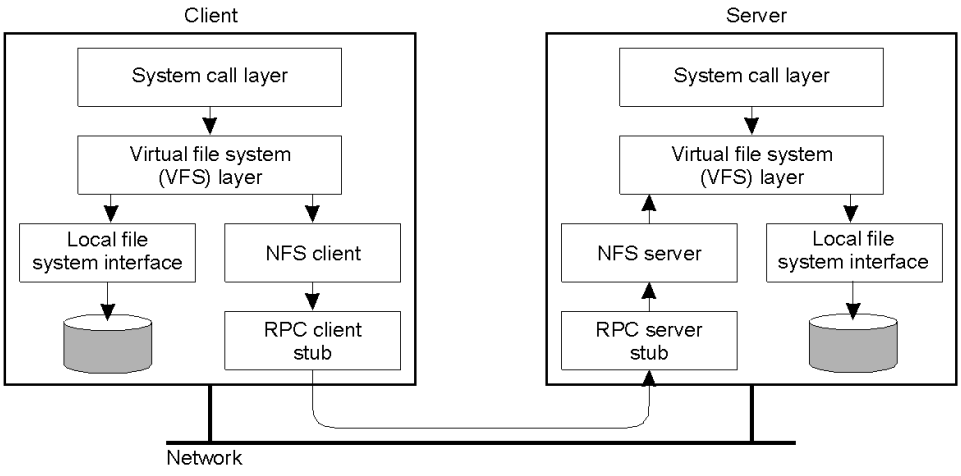
- Upload/download mode
  - Work done at the client
- Stateless server
- Simple functionality (+)
- Moves files/blocks, need storage (-)

System Structure: Server Type

- Stateless server
  - No information is kept at server between client requests
  - All information needed to service a requests must be provided by the client with each request *(what info?)*
  - More tolerant to server crashes
- Stateful server
  - Server maintains information about client accesses
  - Shorted request messages
  - Better performance
  - Idempotency easier
  - Consistency is easier to achieve

NFS Architecture

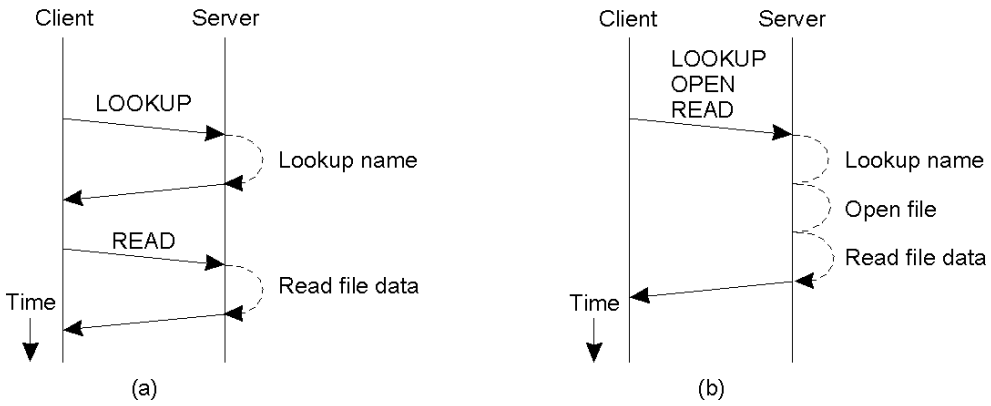
- Sun's Network File System (NFS) – widely used distributed file system
- Uses the virtual file system layer to handle local and remote files



NFS Operations

Operation	v3	v4	Description
Create	Yes	No	Create a regular file
Create	No	Yes	Create a nonregular file
Link	Yes	Yes	Create a hard link to a file
Symlink	Yes	No	Create a symbolic link to a file
Mkdir	Yes	No	Create a subdirectory in a given directory
Mknod	Yes	No	Create a special file
Rename	Yes	Yes	Change the name of a file
Rmdir	Yes	No	Remove an empty subdirectory from a directory
Open	No	Yes	Open a file
Close	No	Yes	Close a file
Lookup	Yes	Yes	Look up a file by means of a file name
Readdir	Yes	Yes	Read the entries in a directory
Readlink	Yes	Yes	Read the path name stored in a symbolic link
Getattr	Yes	Yes	Read the attribute values for a file
Setattr	Yes	Yes	Set one or more attribute values for a file
Read	Yes	Yes	Read the data contained in a file
Write	Yes	Yes	Write data to a file

Communication



a) Reading data from a file in NFS version 3.

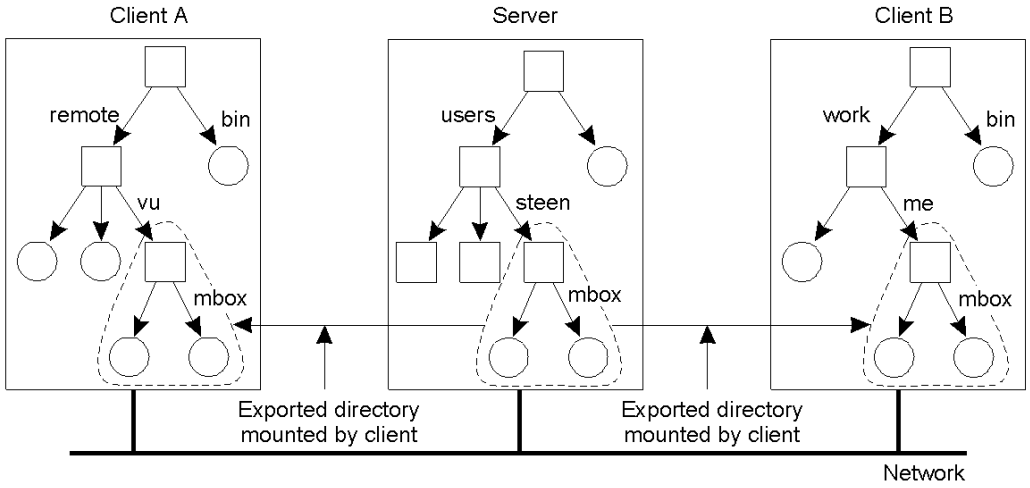
b) Reading data using a compound procedure in version 4.

Both versions use Open Network Computing (ONC) RPCs

- One RPC per operation (NFS v3); multiple operations supported in v4.

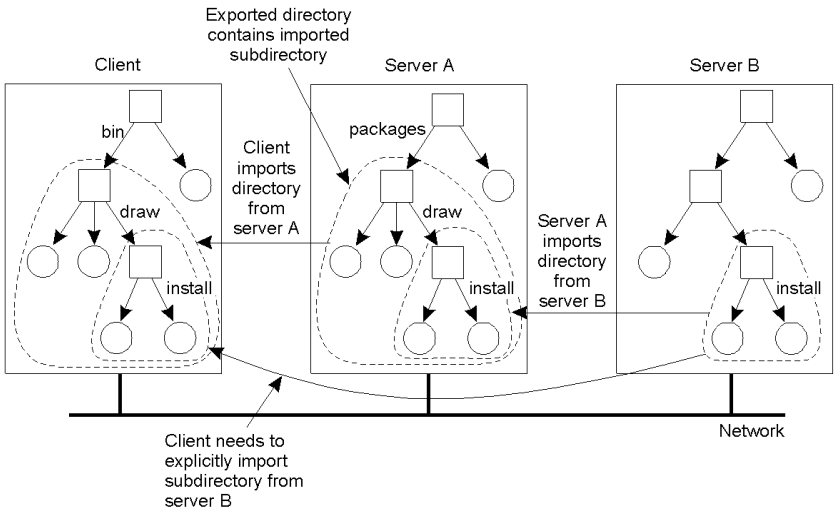
Naming: Mount Protocol

- NFS uses the mount protocol to access remote files
  - Mount protocol establishes a local name for remote files
  - Users access remote files using local names; OS takes care of the mapping

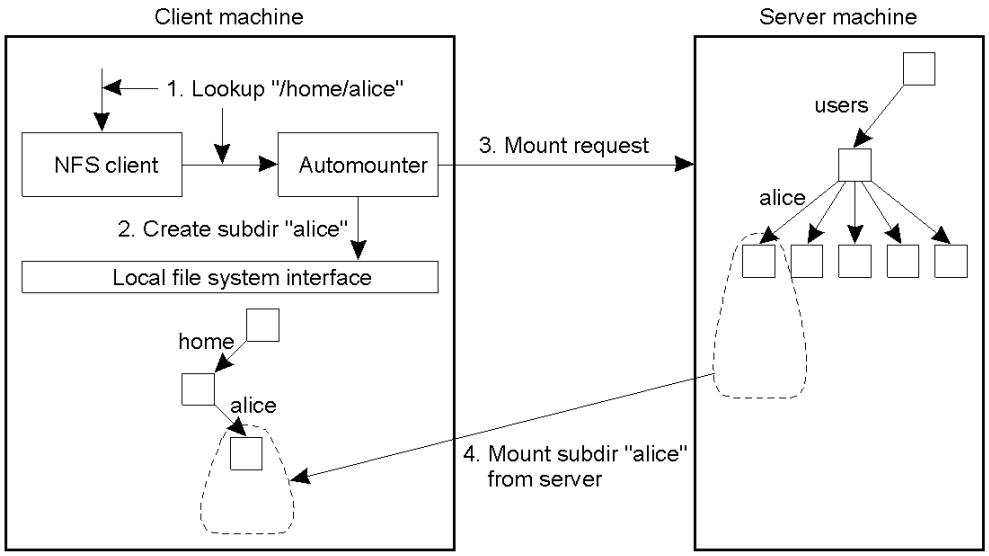


Naming: Crossing Mount Points

- Mounting nested directories from multiple servers
- NFS v3 does not support transitive exports (for security reasons)
  - NFS v4 allows clients to detects crossing of mount points, supports recursive lookups



Automounting



- Automounting: mount on demand

Semantics of File Sharing

Method	Comment
UNIX semantics	Every operation on a file is instantly visible to all processes
Session semantics	No changes are visible to other processes until the file is closed
Immutable files	No updates are possible; simplifies sharing and replication
Transaction	All changes occur atomically

- Four ways of dealing with the shared files in a distributed system.
  - NFS implements session semantics
    - Can use remote/access model for providing UNIX semantics (expensive)
    - Most implementations use local caches for performance and provide session semantics

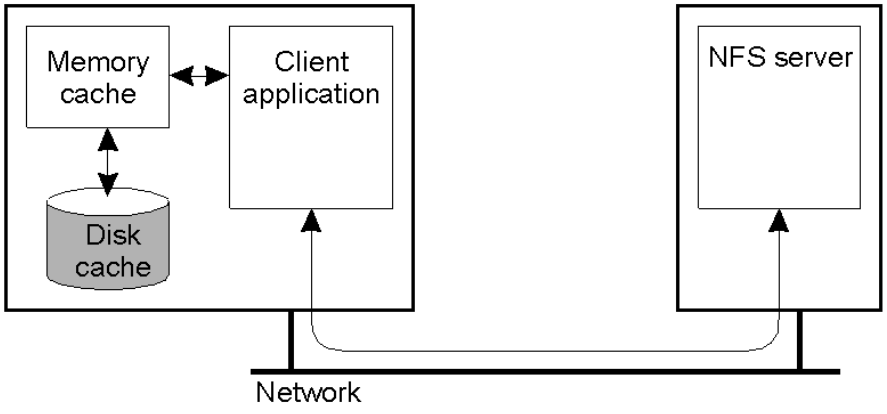
File Locking in NFS

Operation	Description
Lock	Creates a lock for a range of bytes (non-blocking_
Lockt	Test whether a conflicting lock has been granted
Locku	Remove a lock from a range of bytes
Renew	Renew the lease on a specified lock

- NFS supports file locking
  - Applications can use locks to ensure consistency
  - Locking was not part of NFS until version 3
  - NFS v4 supports locking as part of the protocol (see above table)

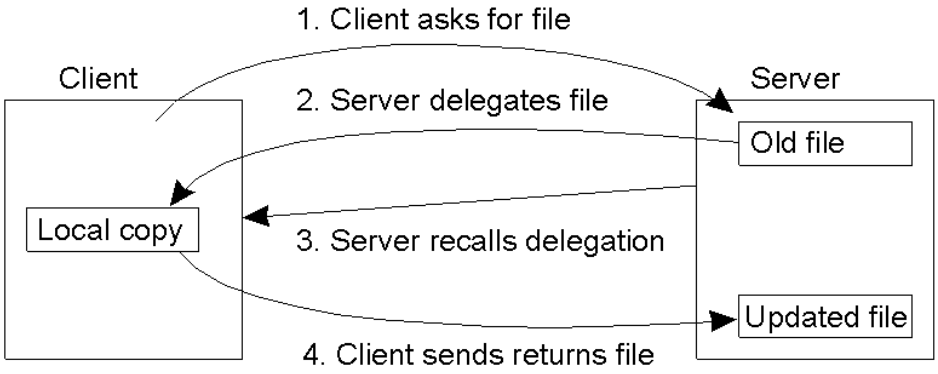
Client Caching

- Client-side caching is left to the implementation (NFS does not prohibit it)
  - Different implementation use different caching policies
    - Sun: allow cache data to be stale for up to 30 seconds

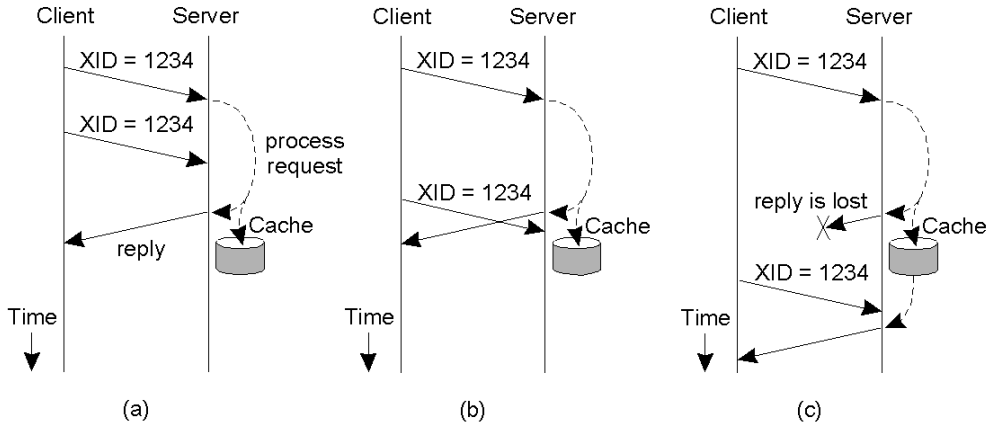


Client Caching: Delegation

- NFS V4 supports open delegation
  - Server delegates local open and close requests to the NFS client
  - Uses a callback mechanism to recall file delegation.



RPC Failures



- Three situations for handling retransmissions: use a duplicate request cache

a) The request is still in progress

b) The reply has just been returned

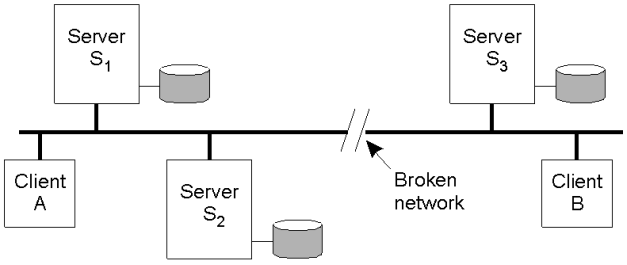
c) The reply has been some time ago, but was lost.

Use a duplicate-request cache: transaction Ids on RPCs, results cached

Coda Overview

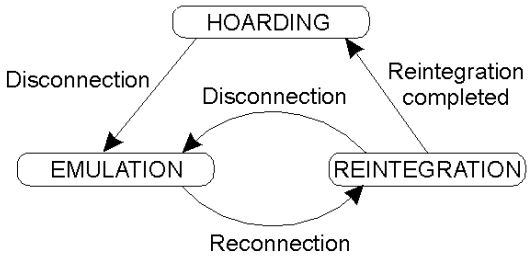
- DFS designed for mobile clients
  - Nice model for mobile clients who are often disconnected
    - Use file cache to make ***disconnection*** transparent
    - At home, on the road, away from network connection
- Coda supplements file cache with user preferences
  - E.g., always keep this file in the cache
  - Supplement with system learning user behavior
- How to keep cached copies on disjoint hosts consistent?
  - In mobile environment, "simultaneous" writes can be separated by hours/days/weeks

Server Replication



- Use replicated writes: read-once write-all
  - Writes are sent to all AVSG (all accessible replicas)
- How to handle network partitions?
  - Use optimistic strategy for replication
  - Detect conflicts using a Coda version vector
  - Example: [2,2,1] and [1,1,2] is a conflict => manual reconciliation

Disconnected Operation



- The state-transition diagram of a Coda client with respect to a volume.
- Use hoarding to provide file access during disconnection
  - Prefetch all files that may be accessed and cache (hoard) locally
  - If AVSG=0, go to emulation mode and reintegrate upon reconnection

xFS Summary

- Distributes data storage across disks using software RAID and log-based network striping
  - RAID == Redundant Array of Independent Disks
- Dynamically distribute control processing across all servers on a per-file granularity
  - Utilizes serverless management scheme
- Eliminates central server caching using cooperative caching
  - Harvest portions of client memory as a large, global file cache.

RAID Overview

- Basic idea: files are "striped" across multiple disks
- Redundancy yields high data availability
  - Availability: service still provided to user, even if some components failed
- Disks will still fail
- Contents reconstructed from data redundantly stored in the array
  - Capacity penalty to store redundant info
  - Bandwidth penalty to update redundant info

Slides courtesy David Patterson

Array Reliability

- Reliability of N disks = Reliability of 1 Disk ÷ N

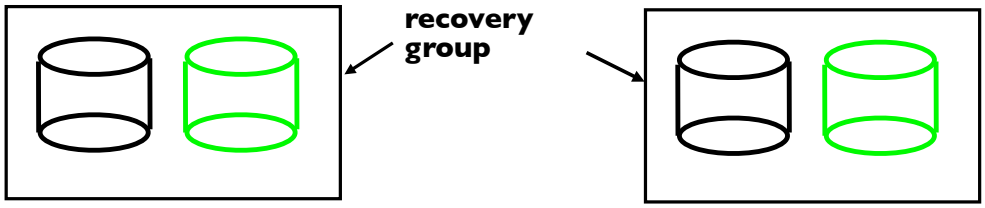
50,000 Hours ÷ 70 disks = 700 hours

Disk system MTTF: Drops from 6 years to 1 month!

- Arrays (without redundancy) too unreliable to be useful!

Hot spares support reconstruction in parallel with access: very high media availability can be achieved

Redundant Arrays of Inexpensive Disks
RAID 1: Disk Mirroring/Shadowing

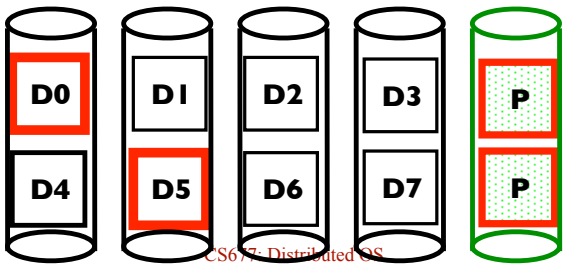


- Each disk is fully duplicated onto its "mirror"
  - Very high availability can be achieved
- Bandwidth sacrifice on write:
  - Logical write = two physical writes
  - Reads may be optimized
- Most expensive solution: 100% capacity overhead

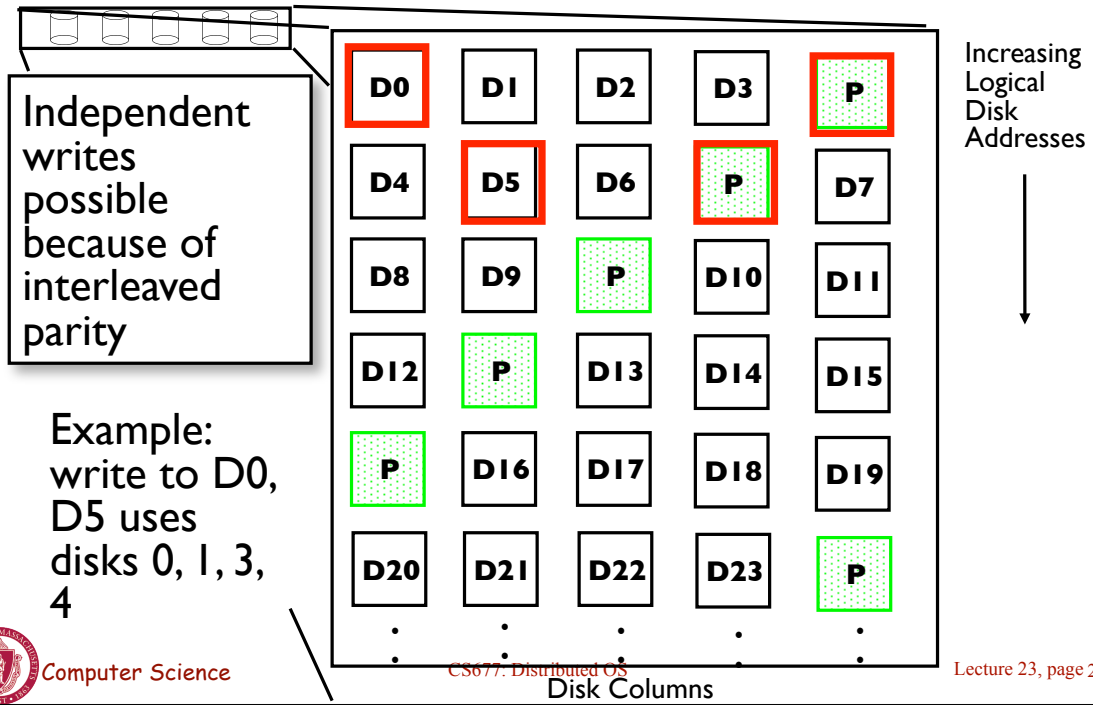
- (RAID 2 not interesting, so skip…involves Hamming codes)

Inspiration for RAID 5

- Use parity for redundancy
  - D0 ⊗ D1 ⊗ D2 ⊗ D3 = P
  - If any disk fails, then reconstruct block using parity:
    - e.g., D0 = D1 ⊗ D2 ⊗ D3 ⊗ P
- RAID 4: all parity blocks stored on the same disk
  - Small writes are still limited by Parity Disk: Write to D0, D5, both also write to P disk
  - Parity disk becomes bottleneck



Redundant Arrays of Inexpensive Disks
RAID 5: High I/O Rate Interleaved Parity



xFS uses software RAID

- Two limitations
  - Overhead of parity management hurts performance for small writes
    - Ok, if overwriting all N-1 data blocks
    - Otherwise, must read old parity+data blocks to calculate new parity
    - Small writes are common in UNIX-like systems
  - Very expensive since hardware RAIDS add special hardware to compute parity