Distributed File Systems

Distributed Files Systems (DFS)

- Allows multi-computer systems to share files
  - Even when no other IPC or RPC is needed
- Sharing devices
  - Special case of sharing files
- E.g.,
  - NFS (Sun's Network File System)
  - Windows NT, 2000, XP
  - Andrew File System (AFS) & others …

Distributed File Systems

- One of most common uses of distributed computing

- *Goal:* provide common view of centralized file system, but distributed implementation.
  - Ability to open & update *any* file on any machine on network
  - All of synchronization issues and capabilities of shared local files

Distributed File System Requirements

- First needs were: access transparency and location transparency.
- Performance, scalability, concurrency control, fault tolerance and security requirements emerged and were met in the later phases of DFS development.

Transparency

- Access transparency: Client programs should be unaware of the the distribution of files.
- Location transparency: Client program should see a uniform namespace. Files should be able to be relocated without changing their path name.
- Mobility transparency: Neither client programs nor system admin program tables in the client nodes should be changed when files are moved either automatically or by the system admin.

Transparency

- Performance transparency: Client programs should continue to perform well on load within a specified range.
- Scaling transparency: increase in size of storage and network size should be transparent.

Other Requirements

- Concurrent file updates is protected (record locking).
- File replication to allow performance.
- Hardware and operating system heterogeneity.
- Fault tolerance
- Consistency : Unix uses on-copy update semantics. This may be difficult to achieve in DFS
- Security and Efficiency

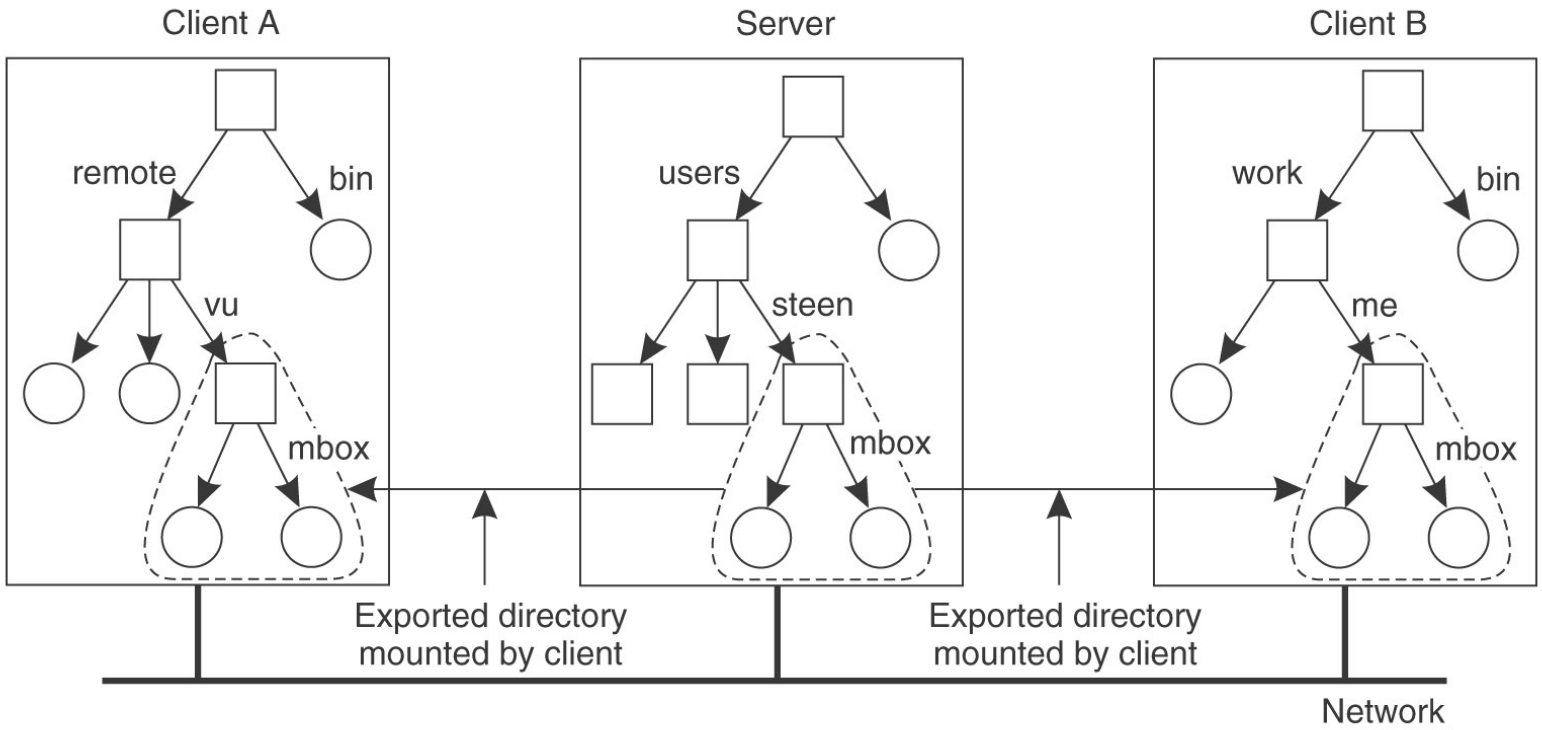
Naming of Distributed Files

- *Naming* – mapping between logical and physical objects
- A *transparent* DFS hides the location where in the network the file is stored.
- **Location transparency** – file name does not reveal the file's physical storage location.
- **Location independence** – file name does not need to be changed when the file's physical storage location changes.
  - Better file abstraction.
  - Separates the naming hierarchy from the storage-devices hierarchy

DFS – Three Naming Schemes

1. *Mount* remote directories to local directories, giving the appearance of a coherent local directory tree
   - *Mounted* remote directories can be accessed transparently.
   - Unix/Linux with NFS; Windows with mapped drives
2. Files named by combination of *host name* and *local name*;
   - Guarantees a unique system wide name
   - Windows *Network Places*, Apollo Domain
3. Total integration of component file systems.
   - A single global name structure spans all the files in the system.

Mounting Remote Directories (NFS)



Mounting Remote Directories

- Note:– *names* of files are not unique
  - As represented by *path names*
- E.g.,
  - Server A sees : */users/steen/mbox*
  - Client A sees: */remote/vu/mbox*
  - Client B sees: */work/me/mbox*
- Consequence:– Cannot pass file "names" around haphazardly

DFS – File Access Performance

- Reduce network traffic by retaining recently accessed disk blocks in local *cache*
- Repeated accesses to the same information can be handled locally.
  - All accesses are performed on the cached copy.
- If needed data not already cached, copy of data brought from the server to the local cache.
  - Copies of parts of file may be scattered in different caches.
- *Cache-consistency* problem – keeping the cached copies consistent with the master file.
  - Especially on write operations

DFS – File Caches

- In client memory
  - Performance speed up; faster access
  - Good when local usage is transient
  - Enables diskless workstations

- On client disk
  - Good when local usage dominates (e.g., AFS)
  - Caches larger files
  - Helps protect clients from server crashes

DFS –Cache Update Policies

- When does the client update the master file?
  - i.e. when is cached data written from the cache to the file?
- *Write-through* – write data through to disk ASAP
  - I.e., following *write*() or *put*(), same as on local disks.
  - Reliable, but poor performance.
- *Delayed-write* – cache and then written to the server later.
  - Write operations complete quickly; some data may be overwritten in cache, saving needless network I/O.
  - Poor reliability
    - unwritten data may be lost when client machine crashes
    - Inconsistent data
  - Variation – scan cache at regular intervals and flush *dirty* blocks.

DFS – File Consistency

- Is locally cached copy of the data consistent with the master copy?

- *Client*-initiated approach
  - Client initiates a validity check with server.
  - Server verifies local data with the master copy
    - E.g., time stamps, etc.

- *Server*-initiated approach
  - Server records (parts of) files cached in each client.
  - When server detects a potential inconsistency, it reacts

DFS – Remote Service *vs.* Caching

- *Remote Service* – all file actions implemented by server.
  - RPC functions
  - Use for small memory diskless machines
  - Particularly applicable if large amount of write activity
- *Cached System*
  - Many "remote" accesses handled efficiently by the local cache
    - Most served as fast as local ones.
  - Servers contacted only occasionally
    - Reduces server load and network traffic.
    - Enhances potential for scalability.
  - Reduces total network overhead

DFS – File Server Semantics

- *Stateless* Service
  - Avoids *state* information in server 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.
  - Poor support for locking or synchronization among concurrent accesses
  - E.g. NFS

DFS – File Server Semantics

- *Stateful* Service
  - Client *opens* a file (as in Unix & Windows).
  - Server fetches information about file from disk, stores in server memory,
    - Returns to client a *connection identifier* unique to client and open file.
    - Identifier used for subsequent accesses until session ends.
  - Server must reclaim space used by no longer active clients.
  - Increased performance; fewer disk accesses.
  - Server retains knowledge about file
    - E.g., read ahead next blocks for sequential access
    - E.g., file locking for managing writes
      - Windows

DFS – Server Semantics Comparison

- Failure Recovery: S*tateful server* loses all volatile state in a crash.
  - Restore state by recovery protocol based on a dialog with clients.
  - Server needs to be aware of crashed client processes
    - orphan detection and elimination.
- Failure Recovery: *Stateless server* failure and recovery are almost unnoticeable.
  - Newly restarted server responds to self-contained requests without difficulty.

DFS – Server Semantics Comparison

- Penalties for using the robust stateless service: –
  - longer request messages
  - slower request processing
- Some environments require stateful service.
  - Server-initiated cache validation cannot provide stateless service.
  - File locking (one writer, many readers).

DFS – Replication

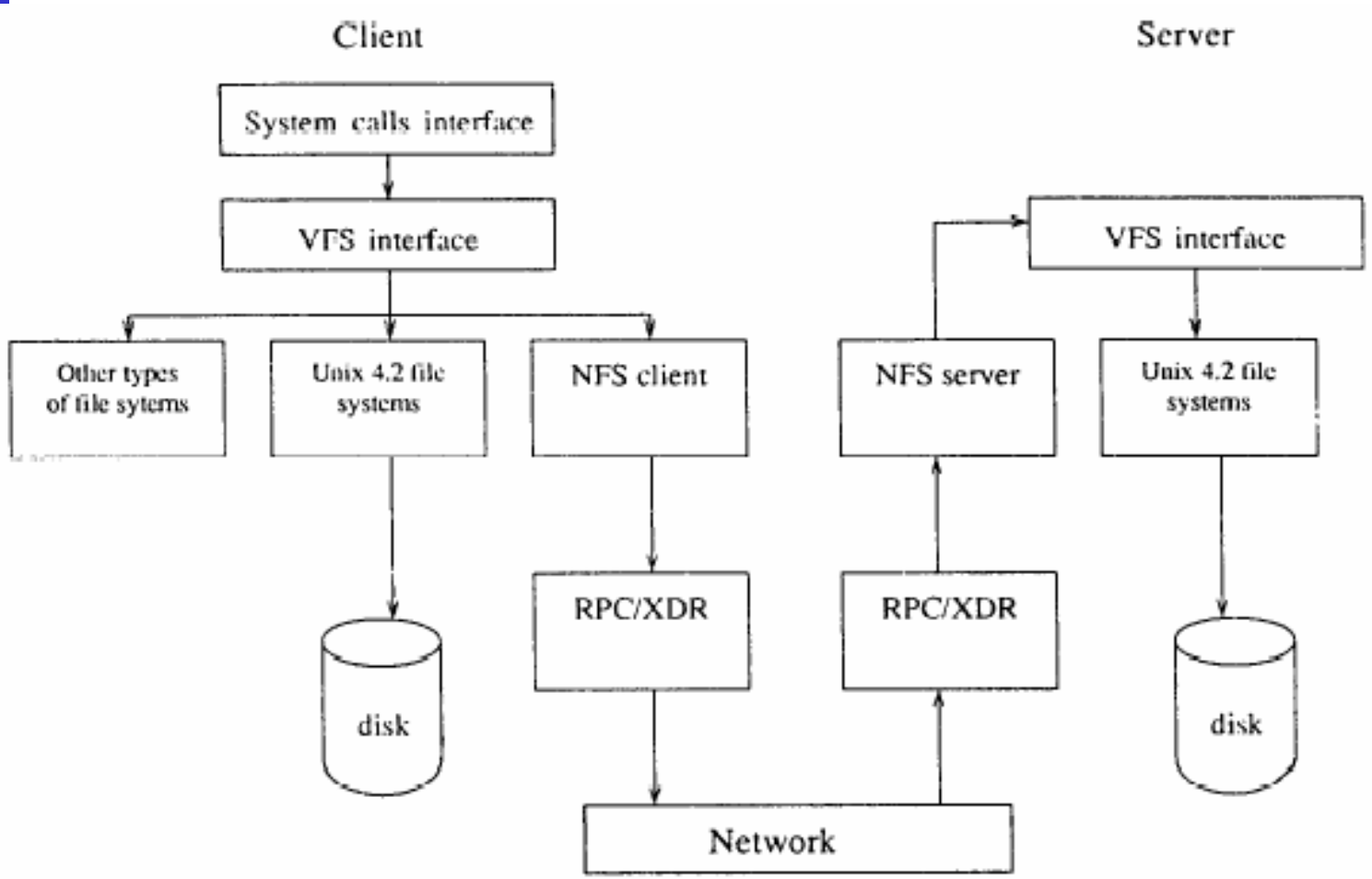
- *Replicas* of the same file reside on failure-independent machines.
- Improves availability and can shorten service time.
- Naming scheme maps a replicated file name to a particular replica.
  - Existence of replicas should be invisible to higher levels.
  - Replicas must be distinguished from one another by different lower-level names.
- Updates
  - Replicas of a file denote the same logical entity
  - Update to any replica *must* be reflected on all other replicas.

A Look at NFS

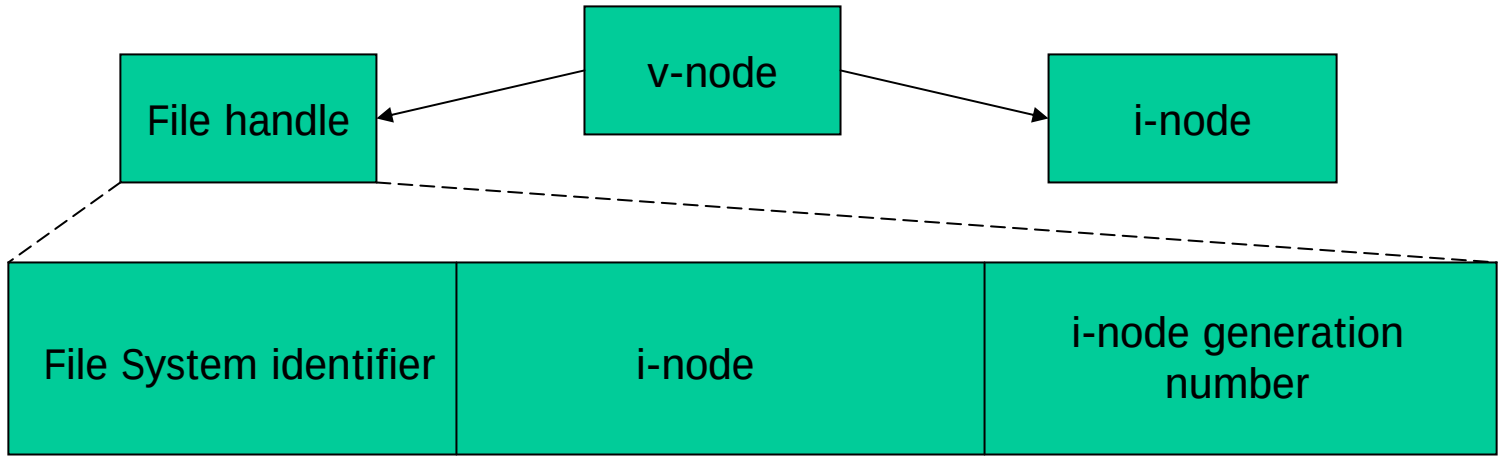
NFS

- Sun Network File System (NFS) has become *de facto* standard for distributed UNIX file access.
- NFS runs over LAN
  - even WAN (slowly)
- Any system may be both a client and server
- Basic idea:
  - Remote directory is *mounted* onto local directory
  - Remote directory may contain mounted directories within

NFS – overview



NFS – v-nodes



- v-node contains a reference to a file handle if the file is remote or an i-node if the file is local
- File system identifier
  - Unique number generated for each file system (in UNIX stored in super block)
- i-node and i-node generation number

NFS – transparency

- Access transparency
  - After mount API same as for UNIX
- Location transparency
  - File names does not reveal anything about their locations (other than the mount points)

NFS – pathname translation (1)

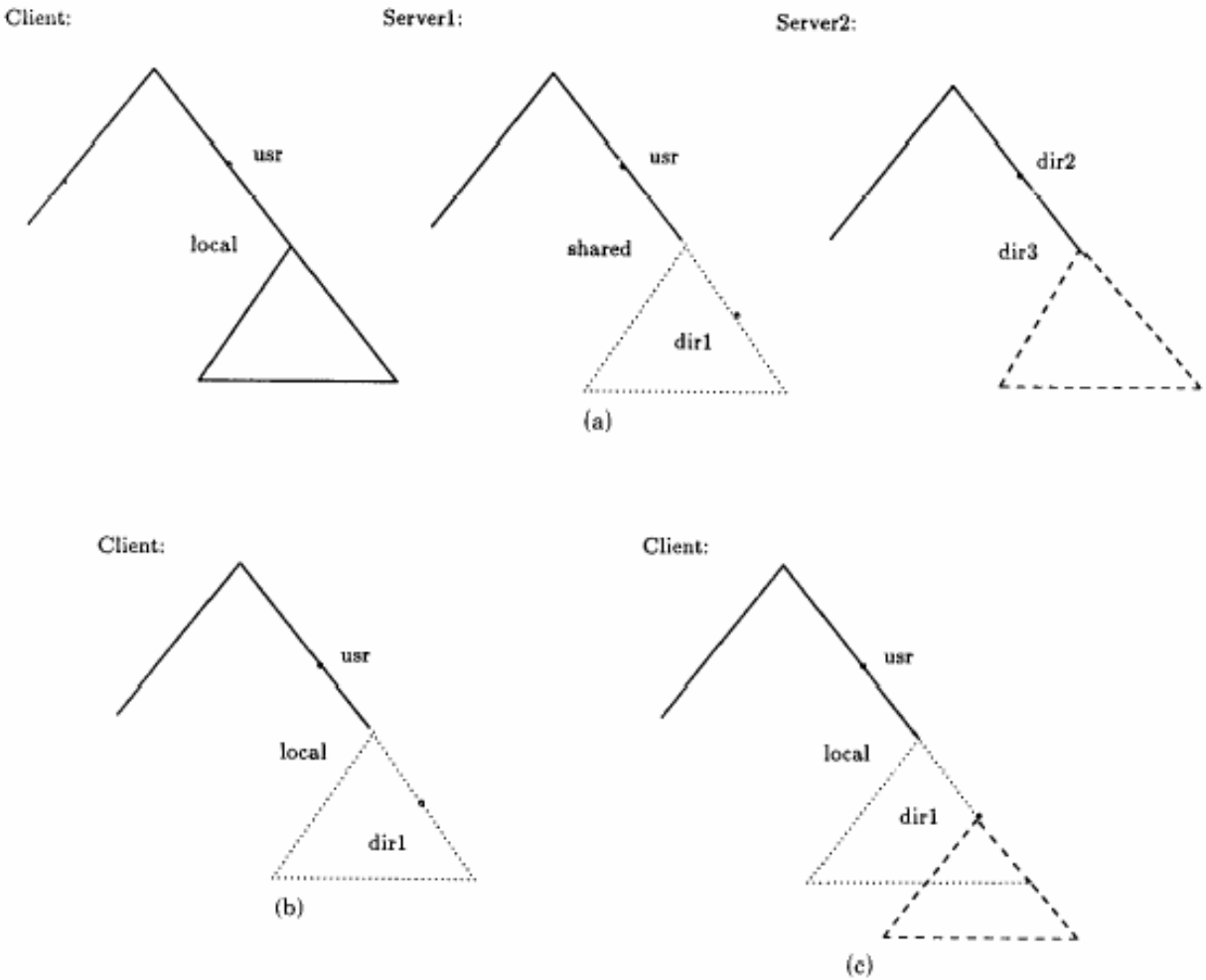


Figure 5. NFS joins independent file systems (a), by mounts (b), and cascading mounts (c).

NFS – pathname translation (2)

- Is done iteratively by client
- /usr/local/dir1/myfile
  - Lookup(/ I-node, usr) → /usr I-node
  - Lookup(/usr I-node, local) → /usr/local file handle
    - Server 1 is contacted
  - Lookup(/usr/local file handle, dir1) → /usr/local/dir1 file handle
    - Server 2 is contacted
  - Lookup(/usr/local/dir1 file handle, myfile) → /usr/local/dir1/myfile file handle
    - Server 2 is contacted
- Server 1 cannot lookup dir1 for client because dir1 is something else on server 1 than on client
- Lookups are cached

NFS – server caching

- Reads
  - Uses the local file system cache (for example UNIX read-ahead)
- Writes
  - Write-through (synchronously, no cache)
  - Commit on close (standard behaviour in v3)

NFS – client caching (reads)

- Clients are responsible for validating cache entries (one of the reasons why the server is stateless)
- Timestamp system used
  - All timestamps are issued by server
- A cache entry is valid if one of the following are true:
  - Cache entry is less than *t* seconds old
  - Modified time at server is the same as modified time on client
- *t* is 3-30 s for files, 30-60 s for directories

NFS – client caching (writes)

- Delayed writes:
  - Modified files are marked dirty and flushed to server on close (or sync)

- Bio-daemons (block input-output):
  - Read-ahead requests are done asynchronously
  - A write request is submitted when a block is filled

NFS Operations

- *Lookup*
  - Fundamental NFS operation
  - Takes pathname, returns *file handle*
- *File Handle*
  - Unique identifier of file within server
  - Persistent; never reused
  - Storable, but opaque to client
    - 64 bytes in NFS v3; 128 bytes in NFS v4
- Most other operations take *file handle* as argument

Other NFS Operations (version 3)

- read, write
- link, symlink
- mknod, mkdir
- rename, rmdir
- readdir, readlink
- getattr, setattr
- create, remove

- Conspicuously absent
  - open, close

NFS v3 — A *Stateless* Service

- Server retains no knowledge of client
  - Server crashes invisible to client
- All hard work done on client side
- Every operation provides *file handle*
- Server caching
  - Performance only
  - Based on recent usage
- Client caching
  - Client checks validity of cached files
  - Client responsible for writing out caches

NFS v3 — A *Stateless* Service

- No locking! No synchronization!
- *Unix file semantics* not guaranteed
  - E.g., *read* after *write*
- *Session semantics* not even guaranteed
  - E.g., *open* after *close*

NFS Implementation

- Remote procedure calls for all operations
  - Implemented in Sun ONC
  - XDR is interface definition language
- Network communication is client-initiated
  - RPC based on UDP (non-reliable protocol)
  - Response to remote procedure call is *de facto* acknowledgement
- Lost requests are simply re-transmitted
  - As many times as necessary to get a response!

NFS Failure Recovery

- Server crashes are transparent to client
  - Each client request contains all information
  - Server can re-fetch from disk if not in its caches
  - Client retransmits request if interrupted by crash
    - (i.e., no response)
- Client crashes are transparent to server
  - Server maintains no record of which client(s) have cached files.

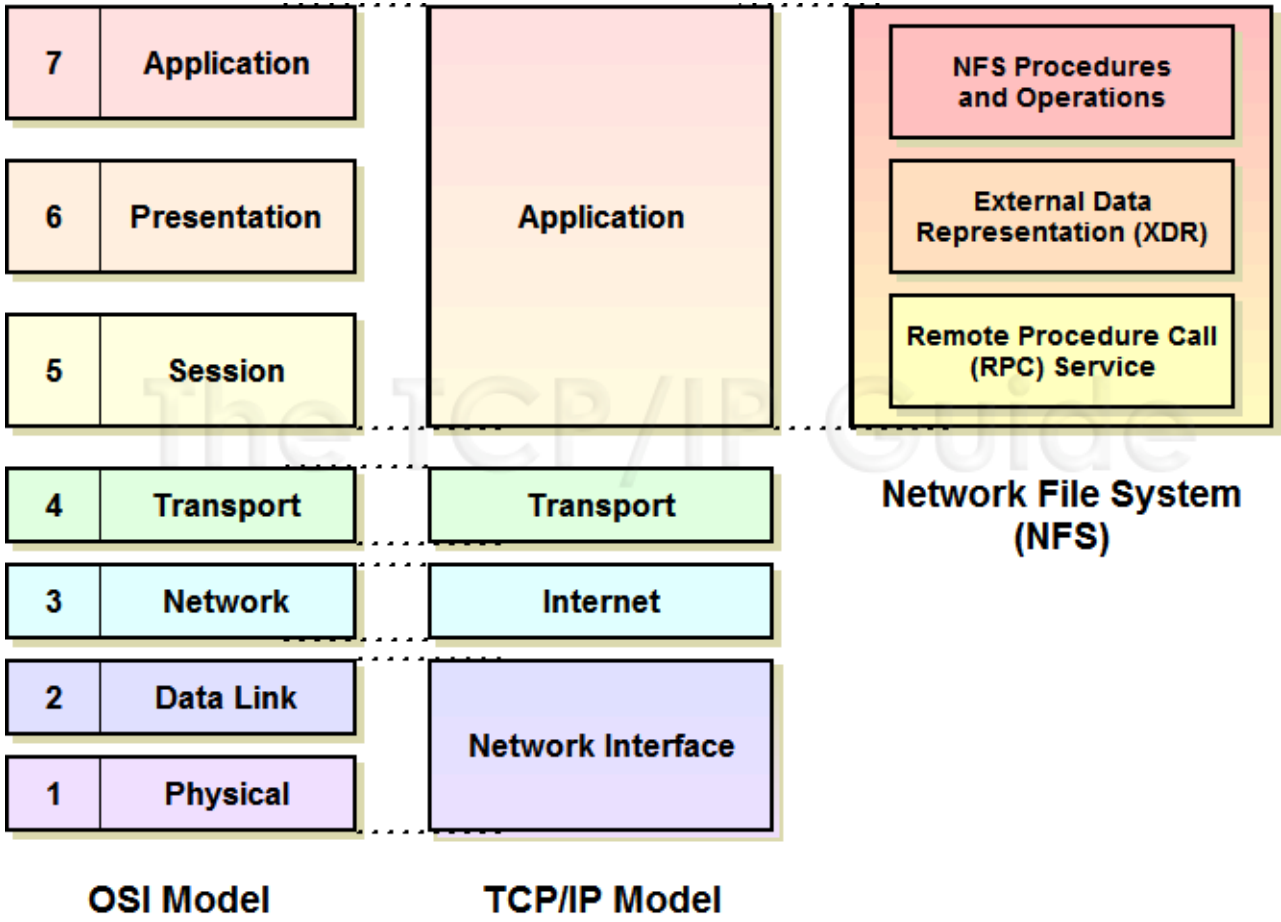
Summary NFS

- Version 3 of NFS
  - *Stateless* file system
  - High performance, simple protocol
- Many things have changed in NFS 4
  - First published in 2000
  - Clarifications published in 2003
  - Almost complete rewrite of NFS

NFS Version 4

- *Stateful* file service
- Based on TCP – reliable transport protocol
- More ways to access server
- Compound requests
  - I.e., multiple RPC calls in same packet
- More emphasis on security
- Mount protocol integrated with rest of NFS protocol

NFS Version 4



NFS Version 4 (continued)

- Additional RPC operations
  - Long list for managing files, caches, validating versions, etc.
  - Also security, permissions, etc.
- Also
  - *Open*() and *close*().
  - With a server crash, some information may have to be recovered

Andrew File System (AFS)

- Completely different kind of file system
- Developed at CMU to support all student computing.
- Consists of workstation clients and dedicated file server machines.

Andrew File System (AFS)

- Stateful
- Single name space
  - File has the same names everywhere in the world.
- Lots of local file caching
  - On workstation disks
  - For long periods of time
  - Originally whole files, now 64K file chunks.
- Good for distant operation because of local disk caching

AFS

- Need for scaling led to reduction of client-server message traffic.
  - Once a file is cached, all operations are performed locally.
  - On close, if the file is modified, it is replaced on the server.
- The client assumes that its cache is up to date!
- Server knows about all cached copies of file
  - *Callback* messages from the server saying otherwise.

AFS

- On file *open*()
  - If client has received a callback for file, it must fetch new copy
  - Otherwise it uses its locally-cached copy.
- Server crashes
  - Transparent to client if file is locally cached
  - Server must contact clients to find state of files

Distributed File Systems

- *Performance* is always an issue
  - Tradeoff between performance and the semantics of file operations (especially for shared files).
- *Caching* of file blocks is crucial in any file system, distributed or otherwise.
  - As memories get larger, most read requests can be serviced out of file buffer cache (local memory).
  - Maintaining coherency of those caches is a crucial design issue.
- Current research addressing disconnected file operation for mobile computers.