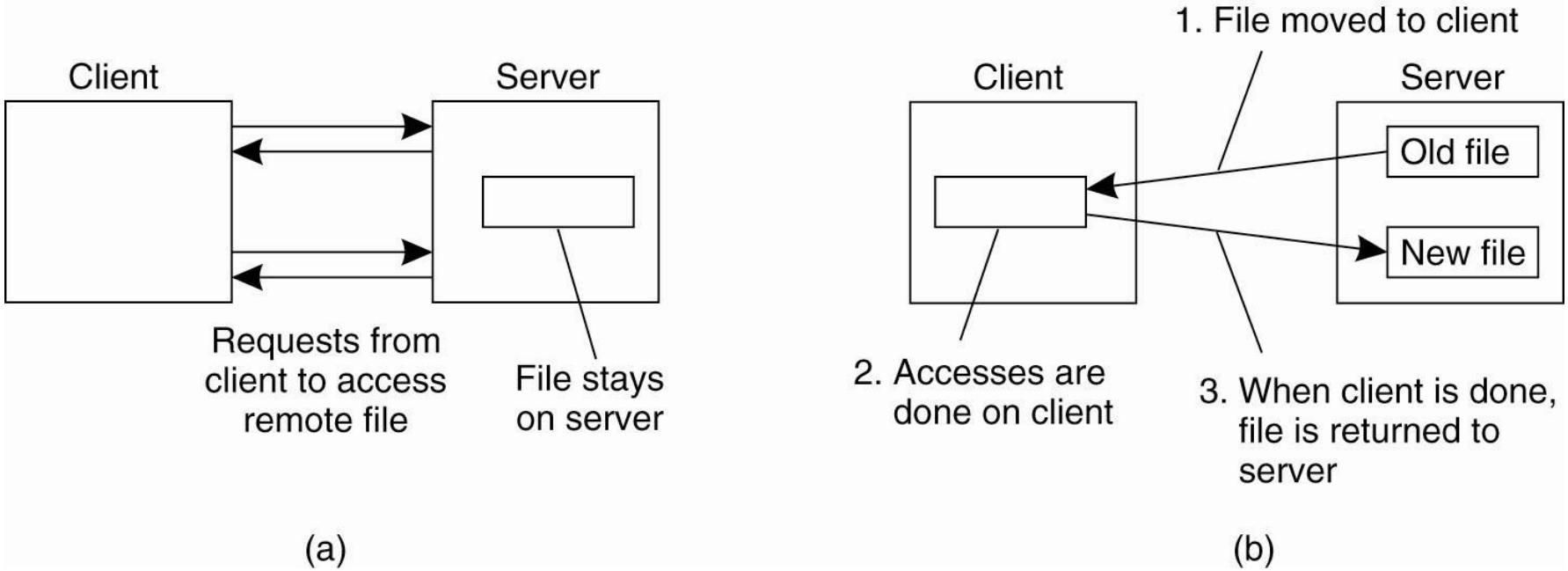
DISTRIBUTED FILE SYSTEMS & NFS

Dr. Yingwu Zhu

File Service Types in Client/Server

- File service
  - a specification of what the file system offers to clients
- File server
  - The implementation of a file service and runs on one or more machines.
- How to access files?
  - Upload/download model (entire files)
  - Remote access model (remote file operations)

Client-Server Architectures



(a) The remote access model.
(b) The upload/download model.

Upload/Download Model

- How:
  - Read file:copy file from server to client
  - Write file:copy file from client to server
- Advantage
  - Simple

- Problems
  - Wasteful: what if client needs small piece?
  - Problematic: what if client doesn't have enough space?
  - Consistency: what if others need to modify the same file?

Remote Access Model

- File service provides functional interface:
  - create, delete, read bytes, write bytes, etc…
- Advantages:
  - Client gets only what's needed
  - Server can manage coherent view of file system
- Problem:
  - Possible server and network congestion
    - Servers are accessed for duration of file access
    - Same data may be requested repeatedly
  - State in server?

File Service Components

- File Directory Service
  - Maps textual names for file to internal locations that can be used by file service
- File service
  - Provides file access interface to clients
- Client modular (driver)
  - Client-side interface for file and directory service
  - if done right, helps provide access transparency e.g. under vnode layer

Semantics of File Sharing

- Sequential Semantics
- Session Semantics
- Other solutions

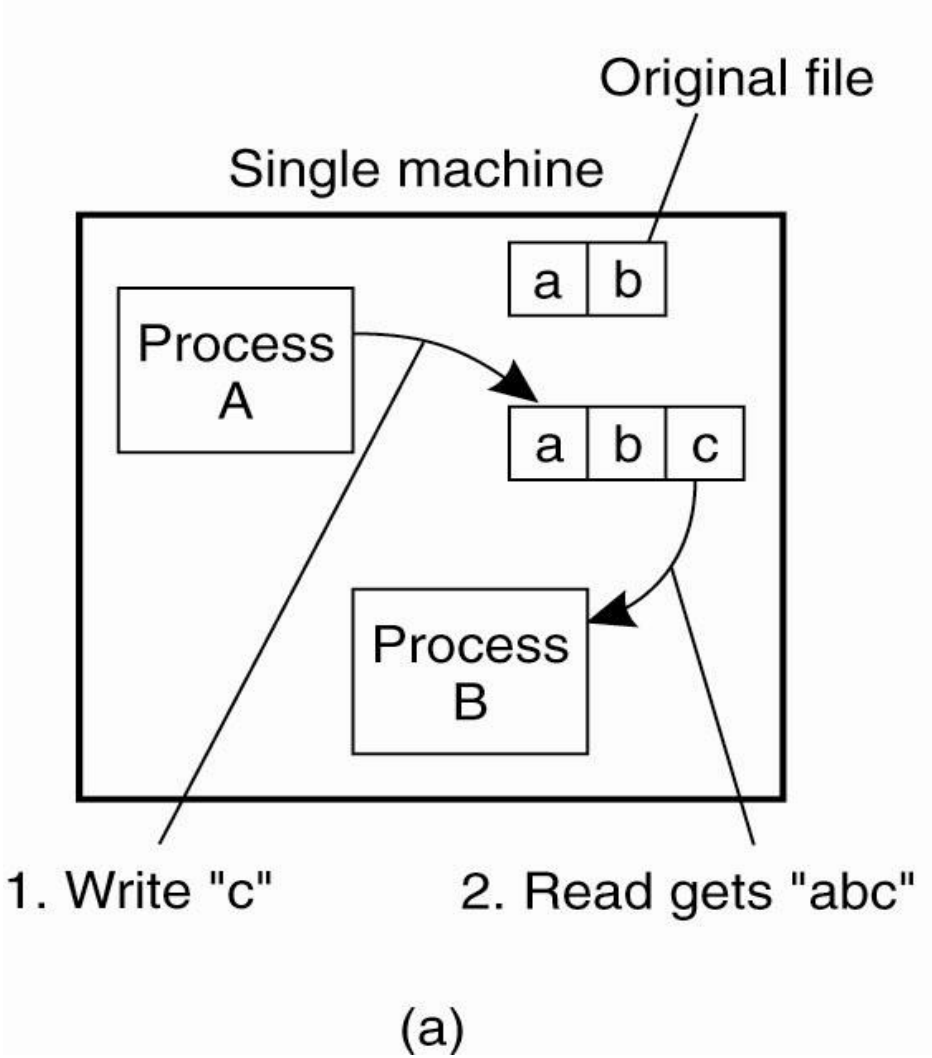
Sequential Semantics

- Read returns result of last write
- Easily achieved if
  - Only one server
  - Clients do not cache data

- BUT…
  - Performance problems if no cache
    - Obsolete data if w/ cache
  - We can **write-through** to the server w/ cache, but
    - Must notify clients holding copies
    - Requires extra state, generates extra traffic

Sequential Semantics

- (a) On a single processor, when a read follows a write, the value returned by the read is the value just written.



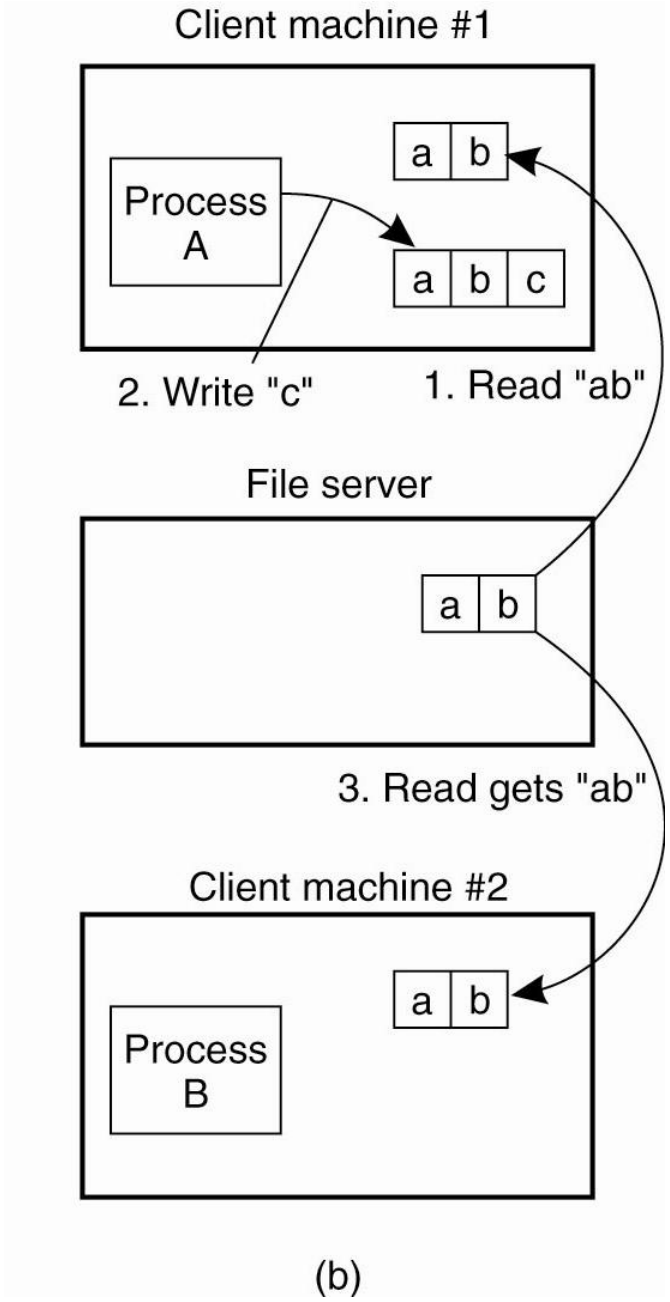
Session Semantics

Relax the rules (not same as the local FS)

- Changes to an open file are initially visible only to the process (or machine) that modified it.
- Last process to modify the file wins.

Session Semantics

(b) In a distributed system with caching, obsolete values may be returned.



(b)

Other Solutions

Make files **immutable**

- Modification is achieved by creating a new file under the old name
- Aids in replication
- Does not help with detecting modification

Or... Use **atomic transactions**

- Each file access is an atomic transaction
- If multiple transactions start concurrently
- Resulting modification is serial

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
Transactions	All changes occur atomically

Four ways of dealing with the shared files in a distributed system.

File Usage Patterns

- We can't have the best of all worlds
- Where to compromise?
  - Semantics vs. efficiency
  - Efficiency = client performance, network traffic, server load
- Understand how files are used
- 1981 study by *Satyanarayanan*

File Usage

- **Most files are <10 KB**
  - Feasible to transfer entire files (simpler)
  - Still have to support long files
- **Most files have short lifetimes**
  - Many temporary files created by editors and compilers
  - Perhaps keep them local
- **Few files are shared**
  - Overstated problem
  - Session semantics will cause no problem most of the time

System Design Issues

How do you access them?

- Access remote files as local files
- Remote FS name space should be syntactically consistent with local name space
  1. redefine the way all files are named and provide a syntax for specifying remote files
     - e.g. //server/dir/file
     - Can cause legacy applications to fail
  2. use a file system mounting mechanism
     - Overlay portions of another FS name space over local name space
     - This makes the remote name space look like it’s part of the local name space

Should servers maintain state? Stateless vs. Stateful

- A stateless system is:
  - one in which the client sends a request to a server, the server carries it out, and returns the result. Between these requests, *no client-specific information is stored on the server.*
- A stateful system is:
  - one where information about client connections is maintained on the server.

Stateless

- Each request must identify file and offsets
- Server can crash and recover
  - No state to lose
- Client can crash and recover
- No open/close needed
  - They only serve to establish state
- No server space used for state
  - Don’t worry about supporting many clients
- Problems if file is deleted on server
- File locking not possible

Caching

- Hide latency to improve performance for repeated accesses, driven by access locality!
- Four places
  - Server's disk
  - Server's buffer cache
  - Client's buffer cache
  - Client's disk

Warning:
Cache consistency problem!

Approaches to Cache Consistency

- **Write-through**
  - What if another client reads its own (out-of-date) cached copy?
  - All accesses will require checking with server
  - Or … server maintains state and sends invalidations

- **Delayed writes (write-behind)**
  - Data can be buffered locally (watch out for consistency – others won't see updates!)
  - Remote files updated periodically
  - One bulk wire is more efficient than lots of little writes
  - Problem: semantics become ambiguous

Approaches to Cache Consistency

- **Read-ahead (prefetch)**
  - Request chunks of data before it is needed.
  - Minimize wait when it actually is needed.
- **Write on close**
  - Admit that we have session semantics.
- **Centralized control**
  - Keep track of who has what open and cached on each node.
  - Stateful file system with signaling traffic.

Case Study

- SUN NFS (Network File Systems)

NFS Design Goals (1)

- Any machine can be a **client or server**
- Must support **diskless workstations**
  - Diskless workstations were Sun's major product line.
- **Heterogeneous systems must be supported**
  - Different HW, OS, underlying file system
- **Access transparency**
  - Remote files accessed as local files through normal file system calls (via VFS in UNIX)
- **Recovery from failure**
  - Stateless, UDP, client retries
- **High Performance**
  - use caching and read-ahead

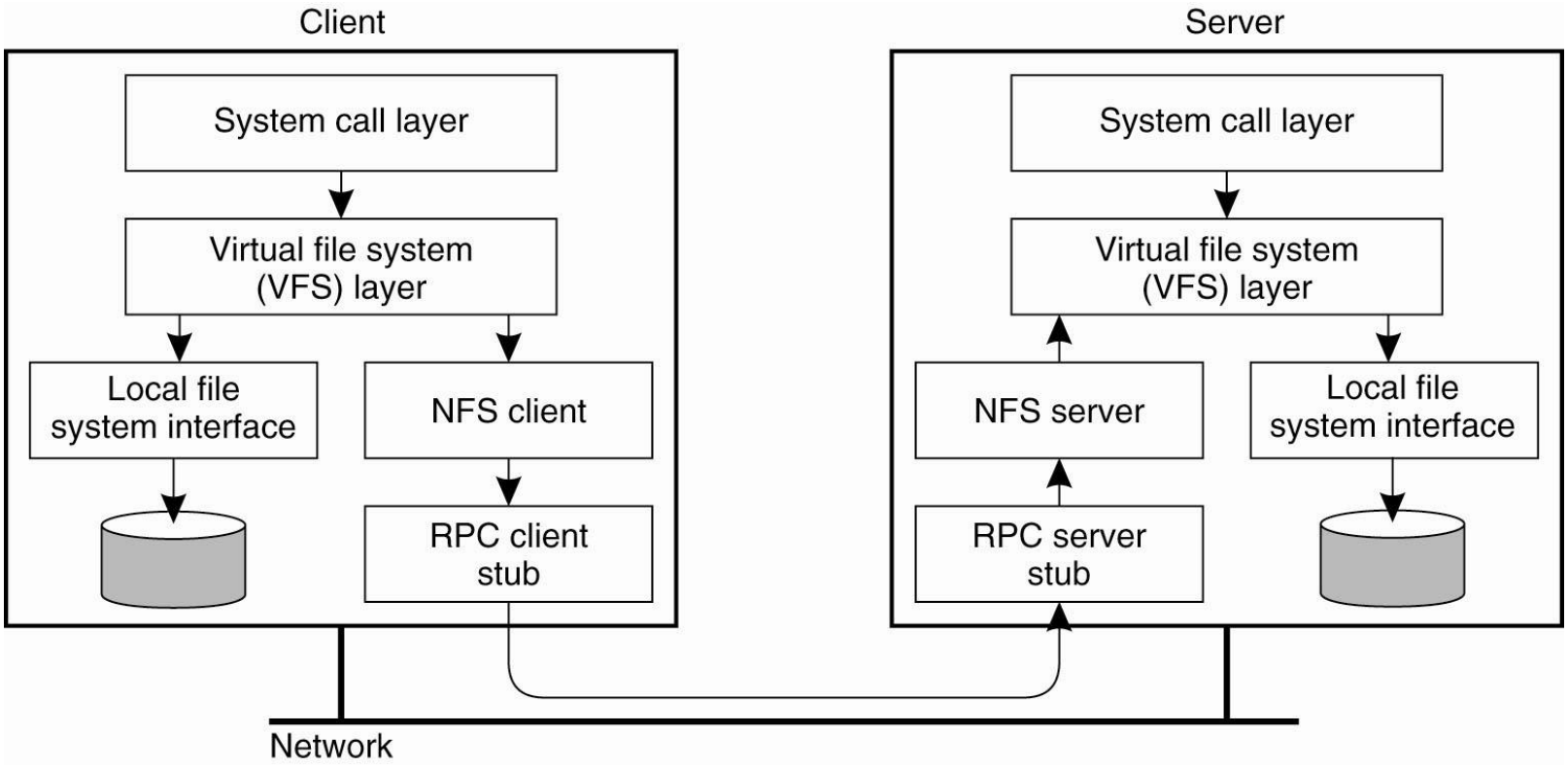
NFS Design Goals (2)

- No migration transparency
  - If resource moves to another server, client must remount resource.
- No support for UNIX file access semantics
  - Stateless design: file locking is a problem.
  - All UNIX file system controls may not be available.
- Devices
  - **must support diskless workstations where every file** is remote.
  - Remote devices refer back to local devices.

NFS Design Goals (3)

- Transport Protocol
  - Initially NFS ran over **UDP using Sun RPC**
- Why UDP?
  - Slightly faster than TCP
  - No connection to maintain (or lose)
  - NFS is designed for Ethernet LAN environment – relatively reliable
  - Error detection but no correction.
    - NFS retries requests

NFS Architecture



The basic NFS architecture for UNIX systems.

Two NFS Protocols

- **Mounting protocol**
  - Request access to exported directory tree
- **Directory & File access protocol**
  - Access files and directories
  - (read, write, mkdir, readdir, …)

Mounting Protocols

- Send pathname to server
- Request permission to access contents

client: parses pathname
contacts server for file handle

- Server returns **file handle**
  - File device #, inode #, instance #

client: create in-core vnode at
mount point.
(points to inode for local files)
points to **rnode** for remote files
- stores state on client

Mounting Protocols

- **static mounting**
  - mount request contacts server
- Server: edit /etc/exports
- Client: mount fluffy:/users/paul /home/paul

Directory and File Access Control

- First, perform a **lookup RPC**
  - returns **file handle and attributes**
- Not like open
  - No information is stored on server
- Handle passed as a parameter for other file access functions
  - e.g. **read(handle, offset, count)**

NFS File Access Control

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
Remove	Yes	Yes	Remove a file from a file system
Rmdir	Yes	No	Remove an empty subdirectory from a directory

NFS File Access Control

Operation	v3	v4	Description
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	Get 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

NFS Performance

- Usually slower than local
- Improve by caching at client
  - Goal: reduce number of remote operations
  - Cache results of
    - read, readlink, getattr, lookup, readdir
  - Cache file data at client (buffer cache)
  - Cache file attribute information at client
  - Cache pathname bindings for faster lookups
- Server side
  - Caching is “automatic” via buffer cache
  - All NFS writes are write-through to disk to avoid unexpected data loss if server dies

Inconsistencies may arise

- Try to resolve by validation
  - Save timestamp of file (assume clock synchronization)
  - When file opened or server contacted for new block
    - Compare last modification time
    - If remote is more recent, invalidate cached data

Validation

- Always invalidate data after some time
  - After 3 seconds for open files (data blocks)
  - After 30 seconds for directories
- If data block is modified, it is:
  - Marked dirty
  - Scheduled to be written
  - Flushed on file close

Improving read performance

- Transfer data in large chunks
  - 8K bytes by default
- Read-ahead
  - Optimize for sequential file access
  - Send requests to read disk blocks before they are requested by the application

Problems with NFS

- File consistency
- Assumes clocks are synchronized
- Open with append cannot be guaranteed to work
- Locking cannot work
  - Separate lock manager added (stateful)
- No reference counting of open files
  - You can delete a file you (or others) have open!
- Global UID space assumed

Problems with NFS

- File permissions may change
  - Invalidating access to file
  - Stateless, still can access the cached copy even after being revoked
- No encryption
  - Requests via unencrypted RPC
  - Authentication methods available
    - Diffie-Hellman, Kerberos, Unix-style
  - Rely on user-level software to encrypt

Improving NFS: version 2

- User-level lock manager
  - Monitored locks
    - status monitor: monitors clients with locks
    - Informs lock manager if host inaccessible
    - If server crashes: status monitor reinstates locks on recovery
    - If client crashes: all locks from client are freed
- NV RAM support
  - Improves write performance
  - Normally NFS must write to disk on server before responding to client write requests
  - Relax this rule through the use of non-volatile RAM

Improving NFS: version 2

- Adjust RPC retries dynamically
  - Reduce network congestion from excess RPC retransmissions under load
  - Based on performance
- Client-side disk caching
  - cacheFS (due to limited RAM for cache)
  - Extend buffer cache to disk for NFS
    - Cache in memory first
    - Cache on disk in 64KB chunks

More improvements… NFS v3

- Updated version of NFS protocol
- Support 64-bit file sizes
- TCP support and large-block transfers
  - UDP caused more problems on WANs (errors)
  - All traffic can be multiplexed on one connection
    - Minimizes connection setup
  - No fixed limit on amount of data that can be transferred between client and server
- Negotiate for optimal transfer size
- Server checks access for entire path from client

More improvements… NFS v3

- New commit operation
  - Check with server after a write operation to see if data is committed
  - If commit fails, client must **resend data**
  - Reduce number of write requests to server
  - Speeds up write requests
    - Don’t require server to write to disk immediately
- Return file attributes with each request
  - Saves extra RPCs