CS194-24
Advanced Operating Systems Structures and Implementation
Lecture 13

File Systems (Con't)
RAID/Journaling/VFS

March 17th, 2014

Prof. John Kubiatowicz

http://inst.eecs.berkeley.edu/~cs194-24

Goals for Today

- Durability
  - RAID
  - Log-structured File System, Journaling
- VFS
- Distributed file systems
- Peer-to-Peer Systems

Interactive is important!

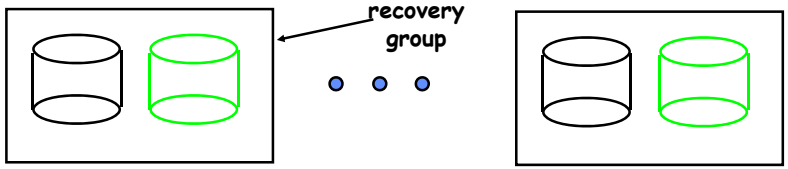
Ask Questions!

Note: Some slides and/or pictures in the following are adapted from slides ©2013

Recall: Important "ilities"

- Availability: the probability that the system can accept and process requests
  - Often measured in "nines" of probability. So, a 99.9% probability is considered "3-nines of availability"
  - Key idea here is independence of failures
- Durability: the ability of a system to recover data despite faults
  - This idea is fault tolerance applied to data
  - Doesn't necessarily imply availability: information on pyramids was very durable, but could not be accessed until discovery of Rosetta Stone
- Reliability: the ability of a system or component to perform its required functions under stated conditions for a specified period of time (IEEE definition)
  - Usually stronger than simply availability: means that the system is not only "up", but also working correctly
  - Includes availability, security, fault tolerance/durability
  - Must make sure data survives system crashes, disk crashes, other problems

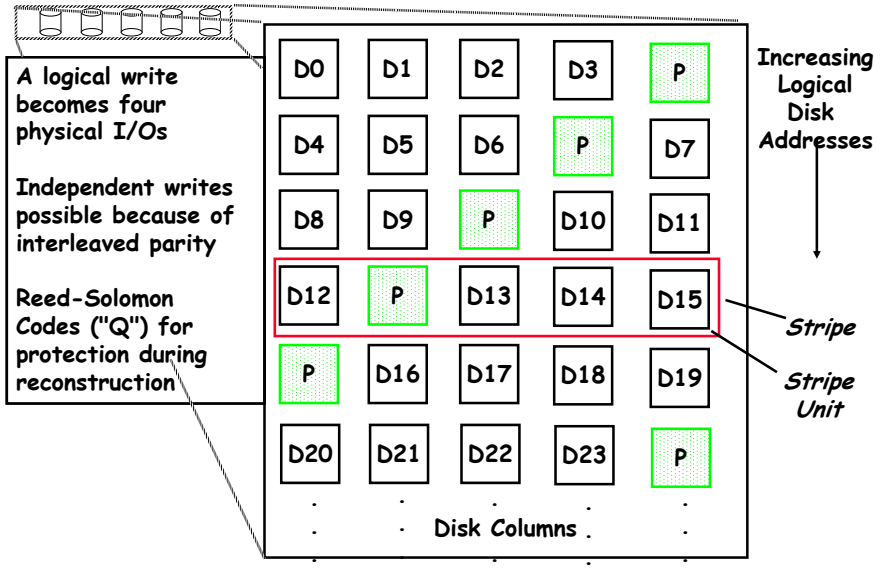
Recall: Redundant Arrays of Disks
RAID 1: Disk Mirroring/Shadowing



- Each disk is fully duplicated onto its "shadow"
  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

Targeted for high I/O rate , high availability environments

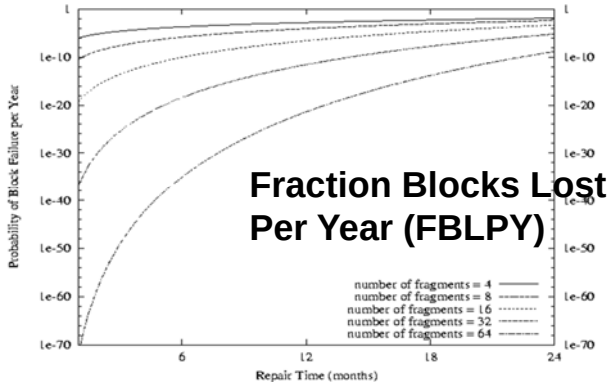
Recall: Redundant Arrays of Disks RAID 5



Reed-Solomon Codes

- RAID6: Capable of dealing with 2 disk failures
  - More complex than simple parity
  - Should not use only RAID5 with modern disk sizes
- Reed-Solomon codes:
  - Based on polynomials in GF(2^k) (I.e. k-bit symbols)
  - Data as coefficients, code space as values of polynomial:
  - $P(x)=a_0+a_1x^1+\ldots a_{k-1}x^{k-1}$
  - Coded: P(0),P(1),P(2)….,P(n-1)
  - Can recover polynomial as long as get *any* k of n
- Properties: can choose number of check symbols
  - Reed-Solomon codes are "maximum distance separable" (MDS)
  - Can add d symbols for distance d+1 code
  - Often used in "erasure code" mode: as long as no more than n-k coded symbols erased, can recover data

Aside: Why erasure coding? High Durability/overhead ratio!



- Exploit law of large numbers for durability!
- 6 month repair, FBLPY:
  - Replication: 0.03
  - Fragmentation: 10^{-35}

Log Structured and Journaled File Systems

- Better (write) performance through use of log
  - Optimized for writes to disk (log is contiguous)
  - Assume that reads handled through page cache
- Better reliability through use of log
  - All changes are treated as *transactions*
  - A transaction is *committed* once it is written to the log
    - » Data forced to disk for reliability
    - » Process can be accelerated with NVRAM
  - Although File system may not be updated immediately, data preserved in the log
- Difference between "Log Structured" and "Journaled"
  - In a Log Structured filesystem, data stays in log form
  - In a Journaled filesystem, Log used for recovery
- For Journaled system:
  - Log used to asynchronously update filesystem
    - » Log entries removed after used
  - After crash:
    - » Remaining transactions in the log performed ("Redo")
    - » Modifications done in way that can survive crashes
- Examples of Journaled File Systems:
  - Ext3 (Linux), XFS (Unix), etc.

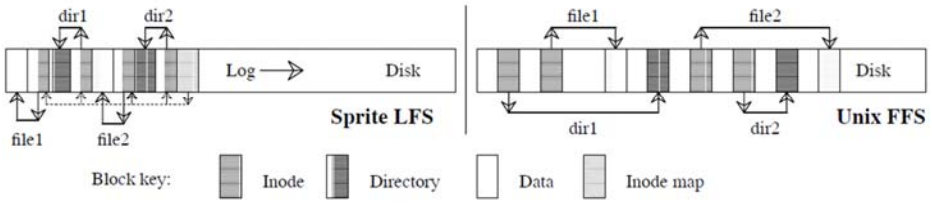
Log-Structured File System – Motivation

- Radically different file system design
- Technology motivations:
  - CPUs outpacing disks: I/O becoming more-and-more of a bottleneck
  - Large RAM: file caches work well, making most disk traffic writes
- Problems with (then) current file systems:
  - Lots of little writes
  - Synchronous: wait for disk in too many places – makes it hard to win much from RAIDs, too little concurrency
  - 5 seeks to create a new file: (rough order)
    1. file i-node (create)
    2. file data
    3. directory entry
    4. file i-node (finalize)
    5. directory i-node (modification time)

LFS Basic Idea

- Log all data and metadata with efficient, large, sequential writes
- Treat the log as the truth, but keep an index on its contents
- Rely on a large memory to provide fast access through caching
- Data layout on disk has "temporal locality" (good for writing), rather than "logical locality" (good for reading)
  - Why is this a better? Because caching helps reads but not writes!
- Two potential problems:
  - Log retrieval on cache misses
  - Wrap-around: what happens when end of disk is reached?
    - » No longer any big, empty runs available
    - » How to prevent fragmentation?

Comparison of LFS and FFS



- Comparison of LFS (Sprite) with FFS (Unix)
  - Creation of two single-block files named dir1/file1 and dir2/file2
  - Each writes new blocks and inodes for file 1 and file 2
  - Each writes new data blocks and inodes for directories
- FFS Traffic:
  - Ten non-sequential writes for new information
  - Inodes each written twice to ease recovery from crashes
- LFS Traffic:
  - Single large write
- For both when reading back: same number of disk accesses

LFS Log Retrieval

- Keep same basic file structure as UNIX (inode, indirect blocks, data)
- Retrieval is just a question of finding a file's inode
- UNIX inodes kept in one or a few big arrays, LFS inodes must float to avoid update-in- place
- Solution: an *inode map* that tells where each inode is (Also keeps other stuff: version number, last access time, free/allocated)
- inode map gets written to log like everything else
- Map of inode map gets written in special checkpoint location on disk; used in crash recovery

LFS Disk Wrap-Around

- Compact live info to open up large runs of free space
  - Problem: long-lived information gets copied over-and-over
- Thread log through free spaces
  - Problem: disk fragments, causing I/O to become inefficient again
- Solution: *segmented log*
  - Divide disk into large, fixed-size segments
  - Do compaction within a segment; thread between segments
  - When writing, use only clean segments (i.e. no live data)
  - Occasionally clean segments: read in several, write out live data in compacted form, leaving some fragments free
  - Try to collect long-lived info into segments that never need to be cleaned
  - Note there is not free list or bit map (as in FFS), only a list of clean segments

LFS Segment Cleaning

- Which segments to clean?
  - Keep estimate of free space in each segment to help find segments with lowest utilization
  - Always start by looking for segment with utilization=0, since those are trivial to clean…
  - If utilization of segments being cleaned is U:
    - » write cost = (total bytes read & written)/(new data written) = 2/(1-U) (unless U is 0)
    - » write cost increases as U increases: U = .9 => cost = 20!
    - » Need a cost of less than 4 to 10; => U of less than .75 to .45
- How to clean a segment?
  - Segment summary block contains map of the segment
  - Must list every i-node and file block
  - For file blocks you need {i-number, block #}

Analysis and Evolution of Journaling File Systems

- Write-ahead logging: commit data by writing it to log, synchronously and sequentially
- Unlike LFS, then later moved data to its normal (FFS-like) location – this write is called *checkpointing* and like segment cleaning, it makes room in the (circular) journal
- Better for random writes, slightly worse for big sequential writes
- All reads go the the fixed location blocks, not the journal, which is only read for crash recovery and checkpointing
- Much better than FFS (fsck) for crash recovery (covered below) because it is much faster
- Ext3/ReiserFS/Ext4 filesystems are the main ones in Linux

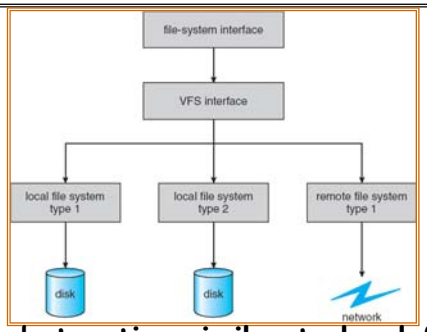
Three modes for a JFS

- *Writeback mode*:
  - Journal only metadata
  - Write back data and metadata independently
  - Metadata may thus have dangling references after a crash (if metadata written before the data with a crash in between)
- *Ordered mode*:
  - Journal only metadata, but always write data blocks before their referring metadata is journaled
  - This mode generally makes the most sense and is used by Windows NTFS and IBM's JFS
- *Data journaling mode*:
  - Write both data and metadata to the journal
  - Huge increase in journal traffic; plus have to write most blocks twice, once to the journal and once for checkpointing

What about remote file systems?

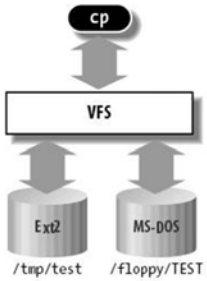
- Remote File System:
  - Storage available over a network
  - Local storage is only a cache on permanent storage
- Advantages?
  - Someone else worries about keeping data safe
  - Data Accessible from multiple physical locations
- Disadvantages?
  - Performance – may take one or more network roundtrips to fetch data
  - Privacy: your data is available over the network, others can possibly see your data
  - Integrity: without sufficient protections, others can overwrite/delete your data

Virtual Filesystem Switch (VFS)



- VFS: Virtual abstraction similar to local file system
  - Instead of "inodes" has "vnodes"
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

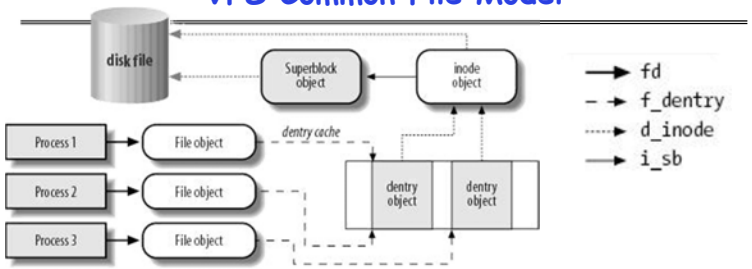
Virtual Filesystem Switch (VFS)




```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

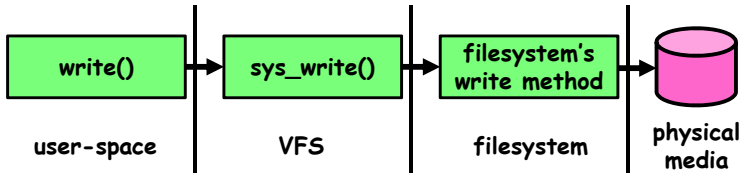
- VFS: Virtual abstraction similar to local file system
  - Provides virtual superblocks, inodes, files, etc
  - Compatible with a variety of local and remote file systems
    - » provides object-oriented way of implementing file systems
- VFS allows the same system call interface (the API) to be used for different types of file systems
  - The API is to the VFS interface, rather than any specific type of file system

VFS Common File Model



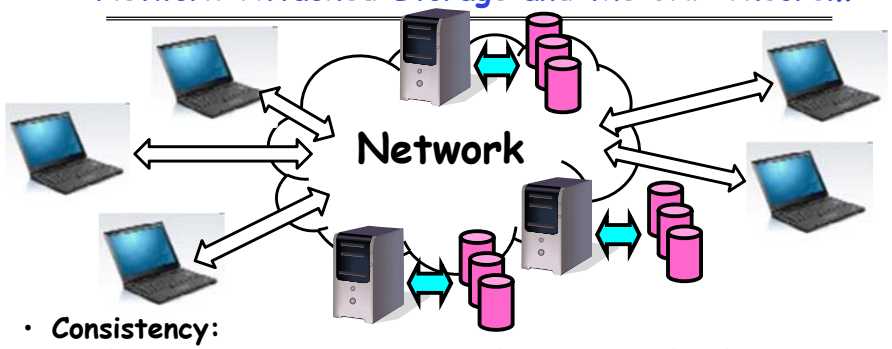
- Four primary object types for VFS:
  - superblock object: represents a specific mounted filesystem
  - inode object: represents a specific file
  - dentry object: represents a directory entry
  - file object: represents open file associated with process
- There is no specific directory object (VFS treats directories as files)
- May need to fit the model by faking it
  - Example: make it look like directories are files
  - Example: make it look like have inodes, superblocks, etc.

Linux VFS



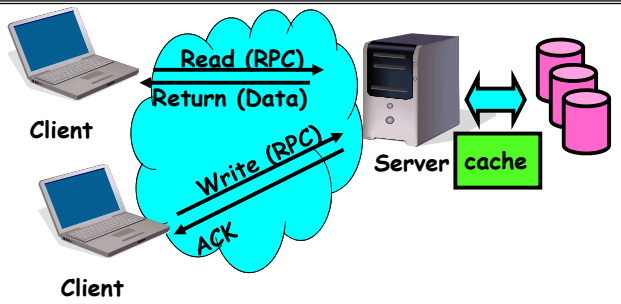
- An operations object is contained within each primary object type to set operations of specific filesystems
  - "super_operations": methods that kernel can invoke on a specific filesystem, i.e. `write_inode()` and `sync_fs()`.
  - "inode_operations": methods that kernel can invoke on a specific file, such as `create()` and `link()`
  - "dentry_operations": methods that kernel can invoke on a specific directory entry, such as `d_compare()` or `d_delete()`
  - "file_operations": methods that process can invoke on an open file, such as `read()` and `write()`
- There are a lot of operations
  - You need to read Bovet Chapter 12 and Love Chapter 13

Network-Attached Storage and the CAP Theorem



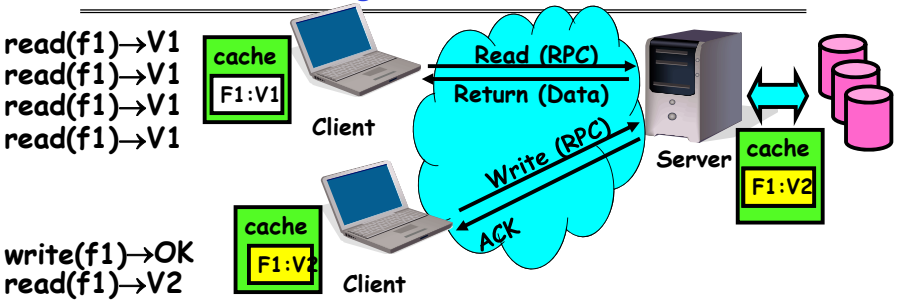
- Consistency:
  - Changes appear to everyone in the same serial order
- Availability:
  - Can get a result at any time
- Partition-Tolerance
  - System continues to work even when network becomes partitioned
- Consistency, Availability, Partition-Tolerance (CAP) Theorem: Cannot have all three at same time
  - Otherwise known as "Brewer's Theorem"

Simple Distributed File System



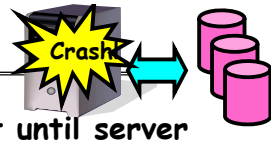
- Remote Disk: Reads and writes forwarded to server
  - Use RPC to translate file system calls
  - No local caching/can be caching at server-side
- Advantage: Server provides completely consistent view of file system to multiple clients
- Problems? Performance!
  - Going over network is slower than going to local memory
  - Lots of network traffic/not well pipelined
  - Server can be a bottleneck

Use of caching to reduce network load



- Idea: Use caching to reduce network load
  - In practice: use buffer cache at source and destination
- Advantage: if open/read/write/close can be done locally, don't need to do any network traffic…fast!
- Problems:
  - Failure:
    - » Client caches have data not committed at server
  - Cache consistency!
    - » Client caches not consistent with server/each other

Failures



- What if server crashes? Can client wait until server comes back up and continue as before?
  - Any data in server memory but not on disk can be lost
  - Shared state across RPC: What if server crashes after seek? Then, when client does "read", it will fail
  - Message retries: suppose server crashes after it does UNIX "rm foo", but before acknowledgment?
    - » Message system will retry: send it again
    - » How does it know not to delete it again? (could solve with two-phase commit protocol, but NFS takes a more ad hoc approach)
- **Stateless protocol:** A protocol in which all information required to process a request is passed with request
  - Server keeps no state about client, except as hints to help improve performance (e.g. a cache)
  - Thus, if server crashes and restarted, requests can continue where left off (in many cases)
- What if client crashes?
  - Might lose modified data in client cache

Network File System (NFS)

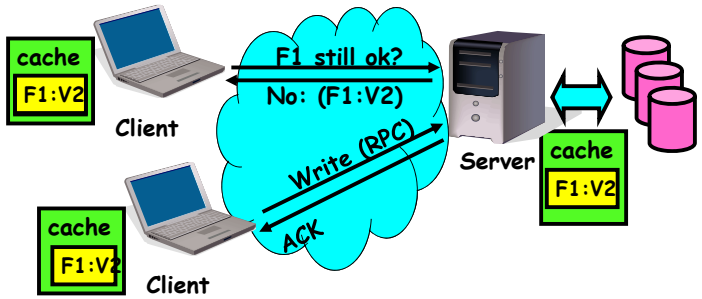
- Three Layers for NFS system
  - **UNIX file-system interface:** open, read, write, close calls + file descriptors
  - **VFS layer:** distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - **NFS service layer:** bottom layer of the architecture
    - » Implements the NFS protocol
- NFS Protocol: RPC for file operations on server
  - Reading/searching a directory
  - manipulating links and directories
  - accessing file attributes/reading and writing files
- **Write-through caching:** Modified data committed to server's disk before results are returned to the client
  - lose some of the advantages of caching
  - time to perform write() can be long
  - Need some mechanism for readers to eventually notice changes! (more on this later)

NFS Continued

- NFS servers are **stateless**; each request provides all arguments require for execution
  - E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
  - No need to perform network open() or close() on file – each operation stands on its own
- **Idempotent:** Performing requests multiple times has same effect as performing it exactly once
  - Example: Server crashes between disk I/O and message send, client resend read, server does operation again
  - Example: Read and write file blocks: just re-read or re-write file block – no side effects
  - Example: What about "remove"? NFS does operation twice and second time returns an advisory error
- Failure Model: Transparent to client system
  - Is this a good idea? What if you are in the middle of reading a file and server crashes?
  - Options (NFS Provides both):
    - » Hang until server comes back up (next week?)
    - » Return an error. (Of course, most applications don't know they are talking over network)

NFS Cache consistency

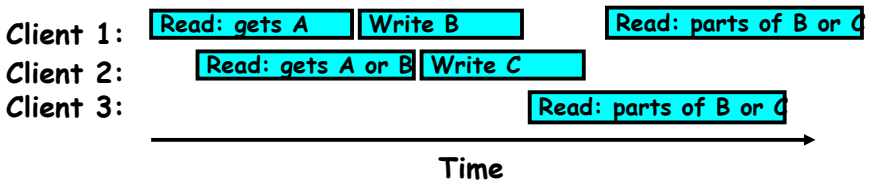
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.



  - What if multiple clients write to same file?
    - » In NFS, can get either version (or parts of both)
    - » Completely arbitrary!

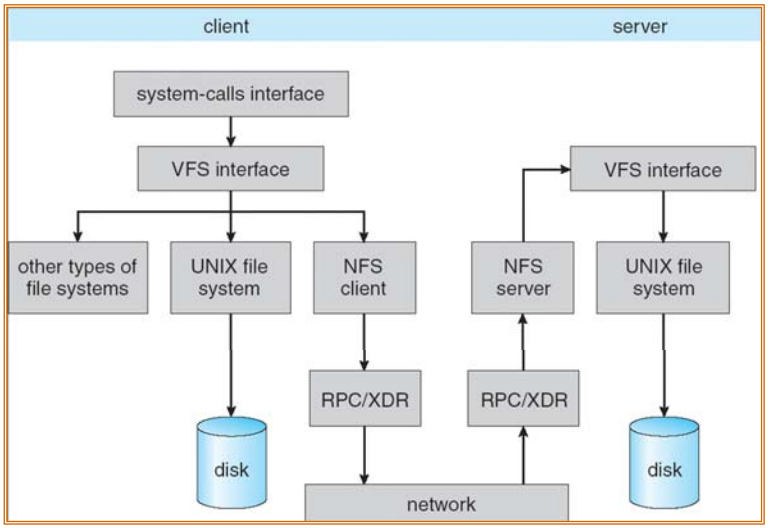
Sequential Ordering Constraints

- What sort of cache coherence might we expect?
  - i.e. what if one CPU changes file, and before it's done, another CPU reads file?
- Example: Start with file contents = "A"



- What would we actually want?
  - Assume we want distributed system to behave exactly the same as if all processes are running on single system
    - » If read finishes before write starts, get old copy
    - » If read starts after write finishes, get new copy
    - » Otherwise, get either new or old copy
  - For NFS:
    - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

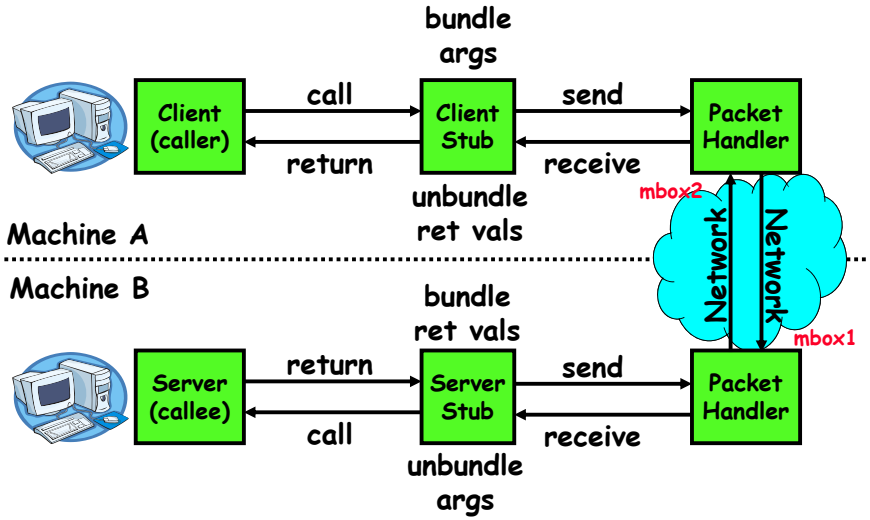
Schematic View of NFS Architecture



Remote Procedure Call

- Raw messaging is a bit too low-level for programming
  - Must wrap up information into message at source
  - Must decide what to do with message at destination
  - May need to sit and wait for multiple messages to arrive
- Better option: Remote Procedure Call (RPC)
  - Calls a procedure on a remote machine
  - Client calls:
    `remoteFileSystem→Read("rutabaga");`
  - Translated automatically into call on server:
    `fileSys→Read("rutabaga");`
- Implementation:
  - Request-response message passing (under covers!)
  - "Stub" provides glue on client/server
    - » Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
    - » Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.
- Marshalling involves (depending on system)
  - Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Information Flow



RPC Details

- Equivalence with regular procedure call
  - Parameters ⇔ Request Message
  - Result ⇔ Reply message
  - Name of Procedure: Passed in request message
  - Return Address: mbox2 (client return mail box)
- Stub generator: Compiler that generates stubs
  - Input: interface definitions in an "interface definition language (IDL)"
    - » Contains, among other things, types of arguments/return
  - Output: stub code in the appropriate source language
    - » Code for client to pack message, send it off, wait for result, unpack result and return to caller
    - » Code for server to unpack message, call procedure, pack results, send them off
- Cross-platform issues:
  - What if client/server machines are different architectures or in different languages?
    - » Convert everything to/from some canonical form
    - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions).

RPC Details (continued)

- How does client know which mbox to send to?
  - Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
  - Binding: the process of converting a user-visible name into a network endpoint
    - » This is another word for "naming" at network level
    - » Static: fixed at compile time
    - » Dynamic: performed at runtime
- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynamic translation of service→mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one
- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next
- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC

- Non-Atomic failures
  - Different failure modes in distributed system than on a single machine
  - Consider many different types of failures
    - » User-level bug causes address space to crash
    - » Machine failure, kernel bug causes all processes on same machine to fail
    - » Some machine is compromised by malicious party
  - Before RPC: whole system would crash/die
  - After RPC: One machine crashes/compromised while others keep working
  - Can easily result in inconsistent view of the world
    - » Did my cached data get written back or not?
    - » Did server do what I requested or not?
  - Answer? Distributed transactions/Byzantine Commit
- Performance
  - Cost of Procedure call « same-machine RPC « network RPC
  - Means programmers must be aware that RPC is not free
    - » Caching can help, but may make failure handling complex

Andrew File System

- Andrew File System (AFS, late 80's) → DCE DFS (commercial product)
- Callbacks: Server records who has copy of file
  - On changes, server immediately tells all with old copy
  - No polling bandwidth (continuous checking) needed
- Write through on close
  - Changes not propagated to server until close()
  - Session semantics: updates visible to other clients only after the file is closed
    - » As a result, do not get partial writes: all or nothing!
    - » Although, for processes on local machine, updates visible immediately to other programs who have file open
- In AFS, everyone who has file open sees old version
  - Don't get newer versions until reopen file

Andrew File System (con't)

- Data cached on local disk of client as well as memory
  - On open with a cache miss (file not on local disk):
    - » Get file from server, set up callback with server
  - On write followed by close:
    - » Send copy to server; tells all clients with copies to fetch new version from server on next open (using callbacks)
- What if server crashes? Lose all callback state!
  - Reconstruct callback information from client: go ask everyone "who has which files cached?"
- AFS Pro: Relative to NFS, less server load:
  - Disk as cache ⇒ more files can be cached locally
  - Callbacks ⇒ server not involved if file is read-only
- For both AFS and NFS: central server is bottleneck!
  - Performance: all writes→server, cache misses→server
  - Availability: Server is single point of failure
  - Cost: server machine's high cost relative to workstation

More Relaxed Consistency?

- Can we get better performance by relaxing consistency?
  - More extensive use of caching
  - No need to check frequently to see if data up to date
  - No need to forward changes immediately to readers
    - » AFS fixes this problem with "update on close" behavior
  - Frequent rewriting of an object does not require all changes to be sent to readers
    - » Consider Write Caching behavior of local file system – is this a relaxed form of consistency?
    - » No, because all requests go through the same cache
- Issues with relaxed consistency:
  - When updates propagated to other readers?
  - Consistent set of updates make it to readers?
  - Updates lost when multiple simultaneous writers?

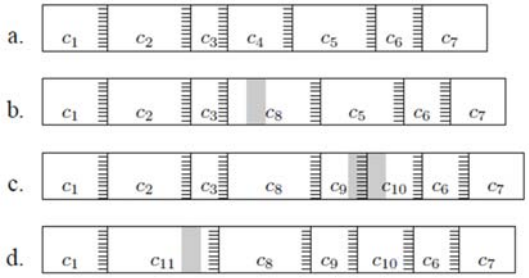
Possible approaches to relaxed consistency

- Usual requirement: Coherence
  - Writes viewed by everyone in the same serial order
- Free-for-all
  - Writes happen at whatever granularity the system chooses: block size, etc
- Update on close
  - As in AFS
  - Makes sure that writes are consistent
- Conflict resolution: Clean up inconsistencies later
  - Often includes versioned data solution
    - » Many branches, someone or something merges branches
  - At server or client
  - Server side made famous by Coda file system
    - » Every update that goes to server contains predicate to be run on data before commit
    - » Provide a set of possible data modifications to be chosen based on predicate

Data Deduplication

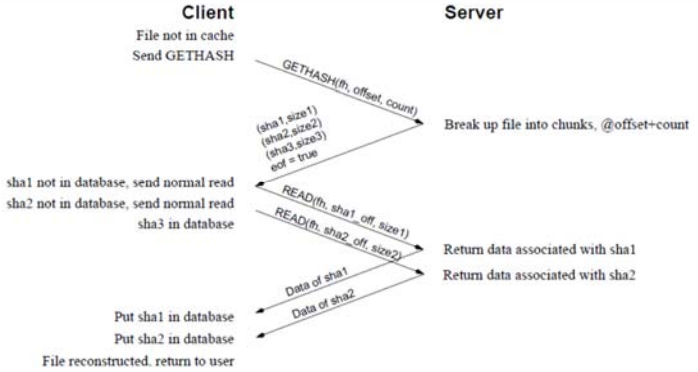
- How to address performance issues with network file systems over wide area? What about caching?
  - Files are often opened multiple times
    - » Caching works
  - Files are often changed incrementally
    - » Caching less works less well
  - Different files often share content or groups of bytes
    - » Caching doesn't work well at all!
- Why doesn't file caching work well in many cases?
  - Because it is based on *names* rather than *data*
    - » Name of file, absolute position within file, etc
- Better option? Base caching on contents rather than names
  - Called "Data de-duplication"

Data-based Caching (Data "De-Duplication")



- Use a sliding-window hash function to break files into chunks
  - Rabin Fingerprint: randomized function of data window
    - » Pick sensitivity: e.g. 48 bytes at a time, lower 13 bits = 0 ⇒ 2^{-13} probability of happening, expected chunk size 8192
    - » Need minimum and maximum chunk sizes
  - Now – if data stays same, chunk stays the same
- Blocks named by cryptographic hashes such as SHA-1

Low Bandwidth File System



- LBFS (Low Bandwidth File System)
  - Based on NFS v3 protocol
  - Uses AFS consistency, however
    - » Writes made visible on close
  - All messages passed through de-duplication process

How Effective is this technique?

Data	Given	Data size	New data	Overlap
emacs 20.7 source	emacs 20.6	52.1 MB	12.6 MB	76%
Build tree of emacs 20.7	—	20.2 MB	12.5 MB	38%
emacs 20.7 + printf executable	emacs 20.7	6.4 MB	2.9 MB	55%
emacs 20.7 executable	emacs 20.6	6.4 MB	5.1 MB	21%
Installation of emacs 20.7	emacs 20.6	43.8 MB	16.9 MB	61%
Elisp doc. + new page	original postscript	4.1 MB	0.4 MB	90%
MSWord doc. + edits	original MSWord	1.4 MB	0.4 MB	68%

- There is a remarkable amount of overlapping content in typical developer file systems
  - Great for source trees, compilation, etc
- Less commonality for binary file formats
- However, this technique is in use in network optimization appliances
- Also works really well for archival backup

Summary (1/2)

- Redundancy
  - RAID: Use of coding and redundancy to protect against failure of disks
  - Reed-Solomon: Erasure coding for even greater protection
- Log Structured File system (LFS)
  - The Log *is* the file system
  - All updates written sequentially in the log
  - Inode map tracks where inodes lie in the log
- Journaling File System (JFS, Ext3, …)
  - Use of log to help durability
  - Primary storage in read-optimized format

Summary (2/2)

- VFS: Virtual File System layer
  - Provides mechanism which gives same system call interface for different types of file systems
- Distributed File System:
  - Transparent access to files stored on a remote disk
  - Caching for performance
- Data De-Duplication: Caching based on data contents
- Peer-to-Peer:
  - Use of 100s or 1000s of nodes to keep higher performance or greater availability
  - May need to relax consistency for better performance
- Next Time: Application-Specific File Systems (e.g. Dynamo, Haystack):
  - Optimize system for particular usage pattern