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

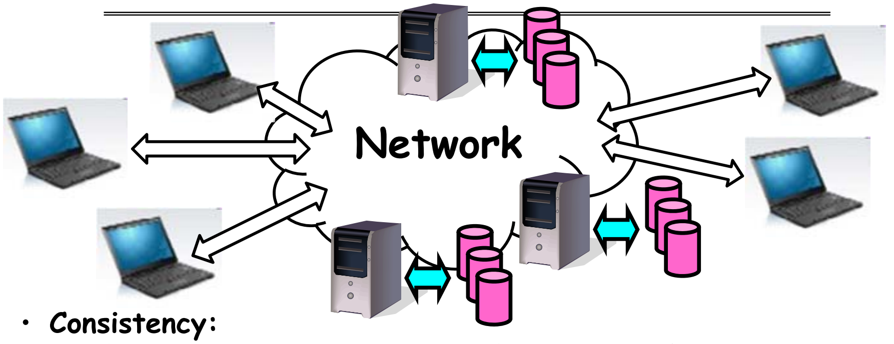
Goals for Today

- Distributed file systems
- Peer-to-Peer Systems
- Application-specific file systems

Interactive is important!
Ask Questions!

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

Recall: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"

Network File System (NFS)

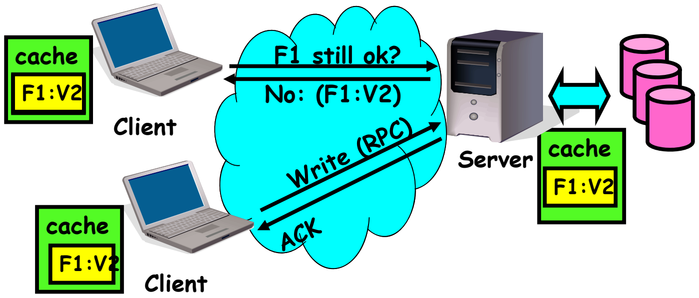
- Three Layers for NFS system
  - UNIX file-system interface: open, read, write, close calls + file descriptors
  - VFS layer: distinguishes local from remote files
    - » Calls the NFS protocol procedures for remote requests
  - NFS service layer: bottom layer of the architecture
    - » Implements 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

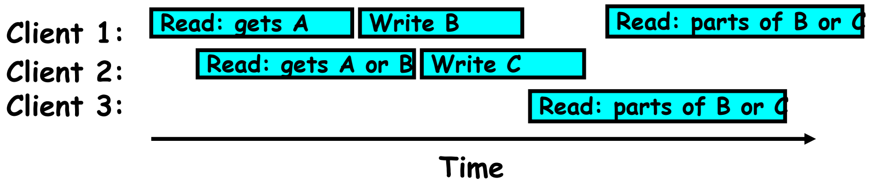
- NFS protocol: weak consistency
  - Client polls server periodically to check for changes
    - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
    - » Thus, when file is changed on one client, server is notified, but other clients use old version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


Client 1: Read: gets A | Write B | Read: parts of B or C

Client 2: Read: gets A or B | Write C

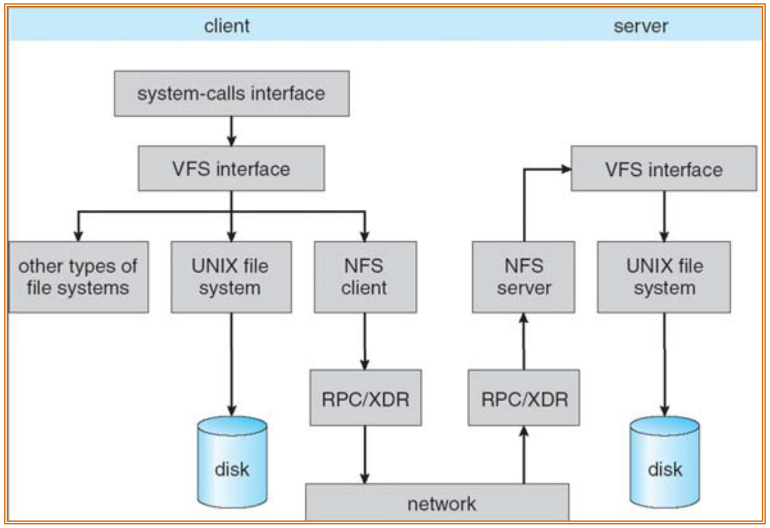
Client 3: Read: parts of B or C

Time

- 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

---

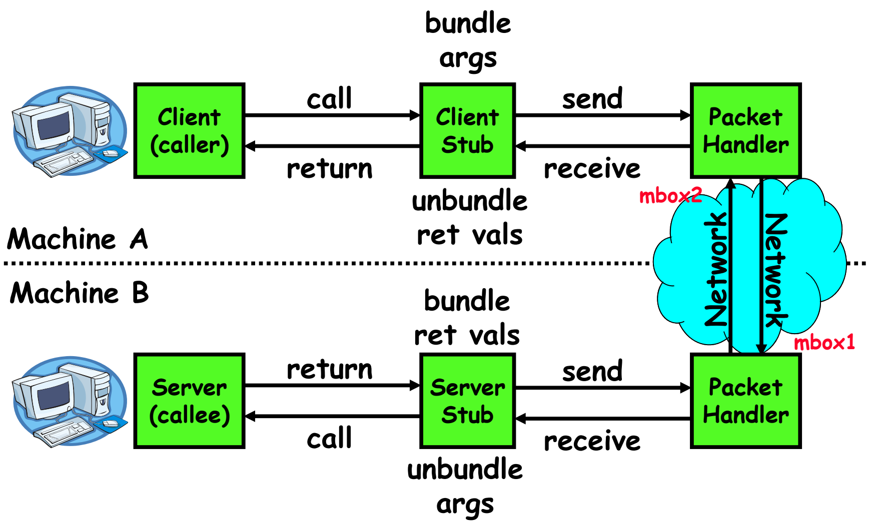
Schematic View of NFS Architecture



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?

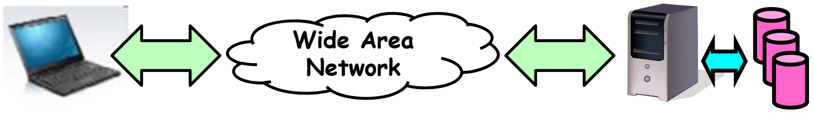
- Can we get better performance by relaxing consistency?
  - More extensive use of caching
  - No need to check frequently to see if data up to date
  - No need to forward changes immediately to readers
    - AFS fixes this problem with "update on close" behavior
  - Frequent rewriting of an object does 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

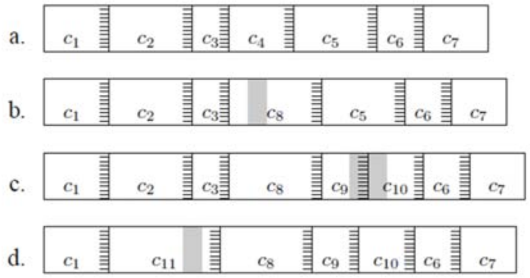
---

Data Deduplication



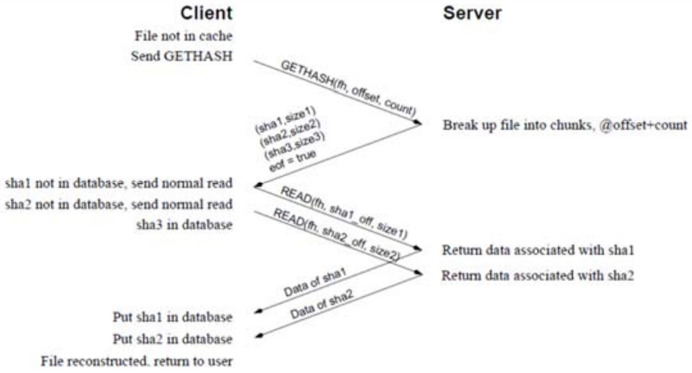
- How to address performance issues with network file systems over wide area? What about caching?
  - Files are often opened multiple times
    - » Caching works
  - Files are often changed incrementally
    - » Caching less works less well
  - Different files often share content or groups of bytes
    - » 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")



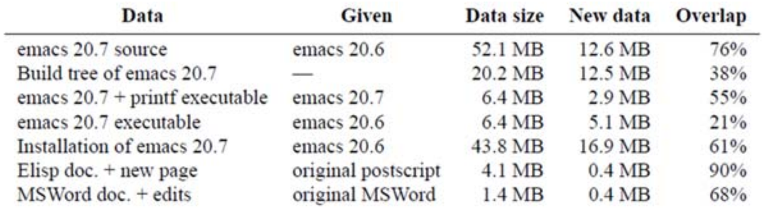
- Use a sliding-window hash function to break files into chunks
  - Rabin Fingerprint: randomized function of data window
    - » Pick sensitivity: e.g. 48 bytes at a time, lower 13 bits = 0 $\Rightarrow 2^{-13}$ probability of happening, expected chunk size 8192
    - » Need minimum and maximum chunk 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?



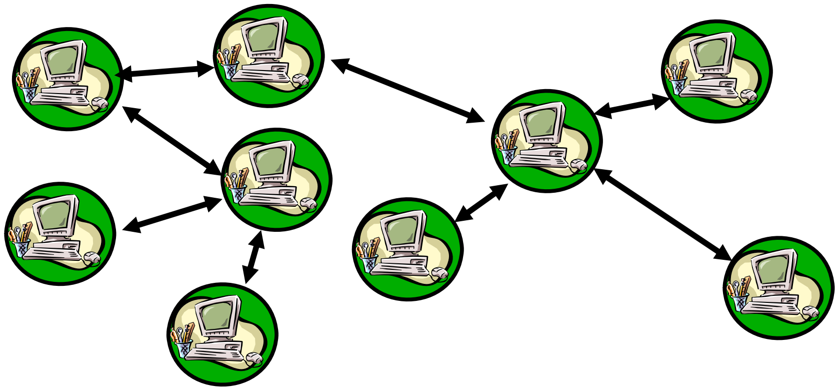
Data	Given	Data size	New data	Overlap
emacs 20.7 source	emacs 20.6	52.1 MB	12.6 MB	76%
Build tree of emacs 20.7	—	20.2 MB	12.5 MB	38%
emacs 20.7 + printf executable	emacs 20.7	6.4 MB	2.9 MB	55%
emacs 20.7 executable	emacs 20.6	6.4 MB	5.1 MB	21%
Installation of emacs 20.7	emacs 20.6	43.8 MB	16.9 MB	61%
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

A different take: Why Peer-to-Peer ideas for storage?

- Incremental Scalability
  - Add or remove nodes as necessary
    - » Systems stays online during changes
  - With many other systems:
    - » Must add large groups of nodes at once
    - » System downtime during change in active set of nodes
- Low Management Overhead (related to first property)
  - System automatically adapts as nodes die or are added
  - Data automatically migrated to avoid failure or take advantage of new nodes
- Self Load-Balance
  - Automatic partitioning of data among available nodes
  - Automatic rearrangement of information or query loads to avoid hot-spots
- Not bound by commercial notions of semantics
  - Can use weaker consistency when desired
  - Provide flexibility to vary semantics on a per-application basis
  - Leads to higher efficiency or performance

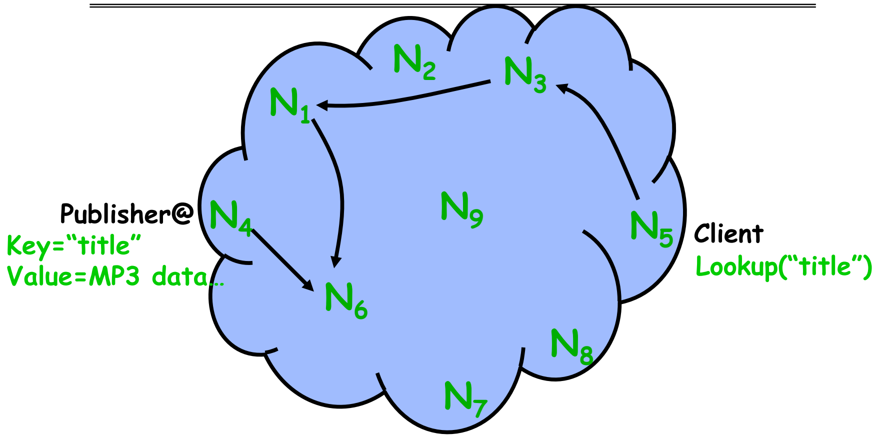
Peer-to-Peer: Fully equivalent components



- Peer-to-Peer has many interacting components
  - View system as a set of equivalent nodes
    - » "All nodes are created equal"
  - Any structure on system must be self-organizing
    - » Not based on physical characteristics, location, or ownership

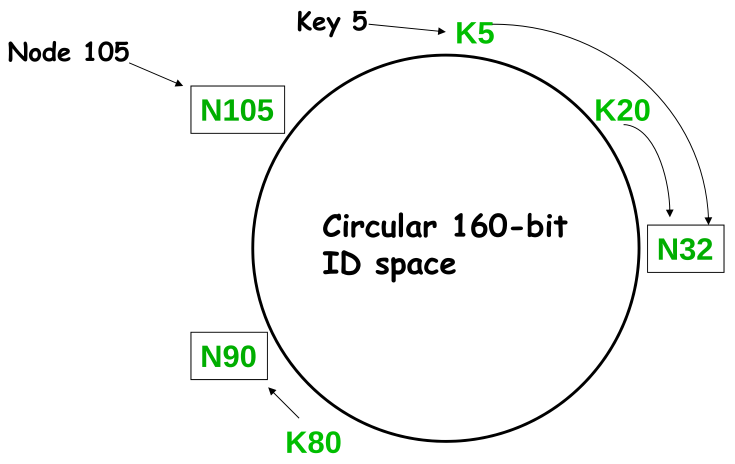
---

Routed queries (Freenet, Chord, Tapestry, etc.)



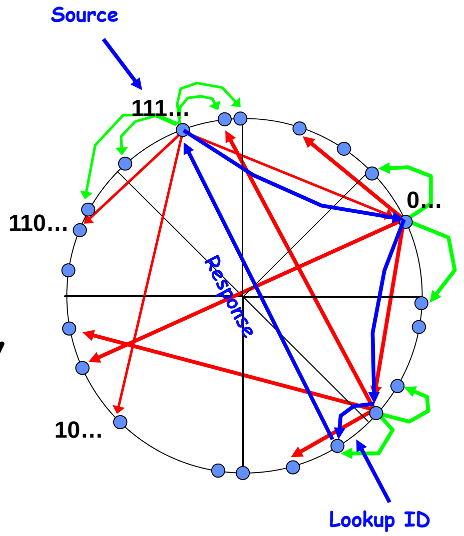
Can be O(log N) messages per lookup (or even O(1))
Potentially complex routing state and maintenance.

Consistent hashing [Karger 97]



A key is stored at its successor: node with next higher ID

Lookup with Leaf Set



- Assign IDs to nodes
  - Map hash values to node with closest ID
- Leaf set is successors and predecessors
  - All that's needed for correctness
- Routing table matches successively longer prefixes
  - Allows efficient lookups
- Data Replication:
  - On leaf set

Advantages/Disadvantages of Consistent Hashing

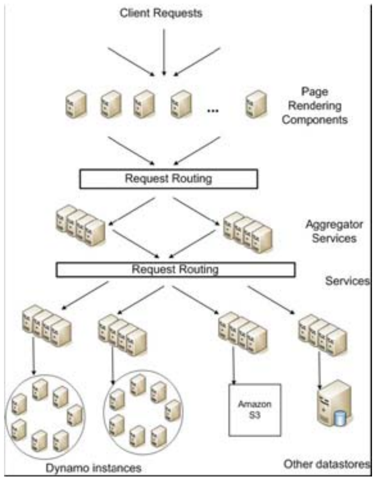
- Advantages:
  - Automatically adapts data partitioning as node membership changes
  - Node given random key value automatically "knows" how to participate in routing and data management
  - Random key assignment gives approximation to load balance
- Disadvantages
  - Uneven distribution of key storage natural consequence of random node names ⇒ Leads to uneven query load
  - Key management can be expensive when nodes transiently fail
    - » Assuming that we immediately respond to node failure, must transfer state to new node set
    - » Then when node returns, must transfer state back
    - » Can be a significant cost if transient failure common
- Disadvantages of "Scalable" routing algorithms
  - More than one hop to find data ⇒ O(log N) or worse
  - Number of hops unpredictable and almost always > 1
    - » Node failure, randomness, etc

Dynamo Assumptions

- Query Model – Simple interface exposed to application level
  - Get(), Put()
  - No Delete()
  - No transactions, no complex queries
- Atomicity, Consistency, Isolation, Durability
  - Operations either succeed or fail, no middle ground
  - System will be eventually consistent, no sacrifice of availability to assure consistency
  - Conflicts can occur while updates propagate through system
  - System can still function while entire sections of network are down
- Efficiency – Measure system by the 99.9th percentile
  - Important with millions of users, 0.1% can be in the 10,000s
- Non Hostile Environment
  - No need to authenticate query, no malicious queries
  - Behind web services, not in front of them

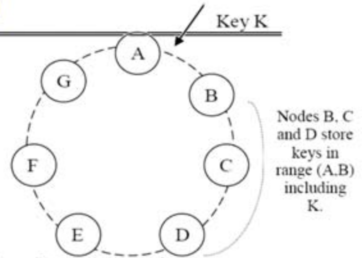
Service Level Agreements (SLA)

- Application can deliver its functionality in a bounded time:
  - Every dependency in the platform needs to deliver its functionality with even tighter bounds.
- Example: service guaranteeing that it will provide a response within 300ms for 99.9% of its requests for a peak client load of 500 requests per second
- Contrast to services which focus on mean response time



Service-oriented architecture of Amazon's platform

Replication

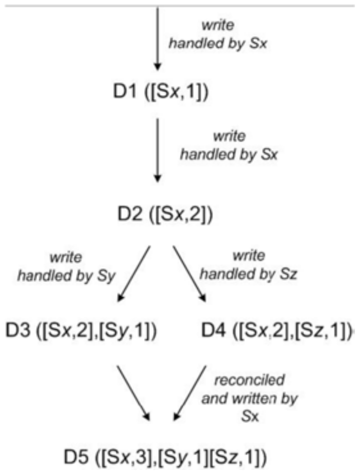


- Each data item is replicated at N hosts
- "*preference list*": The list of nodes responsible for storing a particular key
  - Successive nodes not guaranteed to be on different physical nodes
  - Thus preference list includes physically distinct nodes
- Sloppy Quorum
  - R (or W) is the minimum number of nodes that must participate in a successful read (or write) operation.
  - Setting R + W > N yields a quorum-like system.
  - Latency of a get (or put) is dictated by the slowest of the R (or W) replicas. For this reason, R and W are usually configured to be less than N, to provide better latency.
- Replicas synchronized via anti-entropy protocol
  - Use of Merkle tree for each unique range
  - Nodes exchange root of trees for shared key range

Data Versioning

- A put() call may return to its caller before the update has been applied at all the replicas
- A get() call may return many versions of the same object.
- **Challenge:** an object having distinct version sub-histories, which the system will need to reconcile in the future.
- **Solution:** uses vector clocks in order to capture causality between different versions of the same object
  - A vector clock is a list of (node, counter) pairs
  - Every version of every object is associated with one vector clock
  - *If the counters on the first object's clock are less-than-or-equal to all of the nodes in the second clock, then the first is an ancestor of the second and can be forgotten.*

Vector clock example



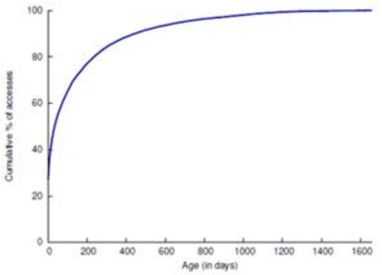
Conflicts (multiversion data)

- Client must resolve conflicts
  - Only resolve conflicts on reads
  - Different resolution options:
    - » Use vector clocks to decide based on history
    - » Use timestamps to pick latest version
  - Examples given in paper:
    - » For shopping cart, simply merge different versions
    - » For customer's session information, use latest version
  - Stale versions returned on reads are updated ("read repair")
- Vary N, R, W to match requirements of applications
  - High performance reads: R=1, W=N
  - Fast writes with possible inconsistency: W=1
  - Common configuration: N=3, R=2, W=2
- When do branches occur?
  - Branches uncommon: 0.06% of requests saw > 1 version over 24 hours
  - Divergence occurs because of high write rate (more coordinators), not necessarily because of failure

---

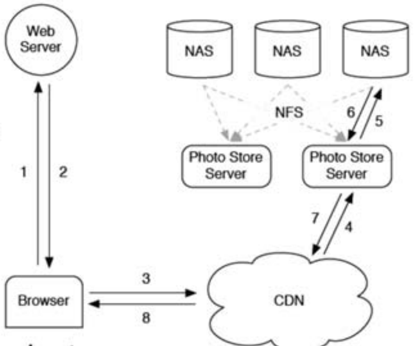
Haystack File System

- Does it ever make sense to adapt a file system to a particular usage pattern?
  - Perhaps
- Good example: Facebook's "Haystack" filesystem
  - Specific application (Photo Sharing)
    - » Large files!, Many files!
    - » 260 Billion images, 20 PetaBytes (10^{15} bytes!)
    - » One billion new photos a week (60 TeraBytes)
  - Presence of Content Delivery Network (CDN)
    - » Distributed caching and distribution network
    - » Facebook web servers return special URLs that encode requests to CDN
    - » Pay for service by bandwidth
  - Specific usage patterns:
    - » New photos accessed a lot (caching well)
    - » Old photos accessed little, but likely to be requested at any time ⇒ NEEDLES



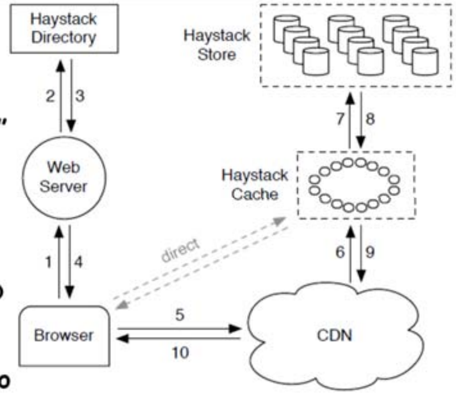
Number of photos requested in day

Old Solution: NFS



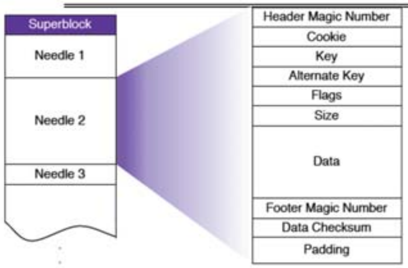
- Issues with this design?
- Long Tail ⇒ Caching does not work for most photos
  - Every access to back end storage must be *fast* without benefit of caching!
- Linear Directory scheme works badly for many photos/directory
  - Many disk operations to find even a single photo
  - Directory's block map too big to cache in memory
  - "Fixed" by reducing directory size, however still not great
- Meta-Data (FFS) requires ≥ 3 disk accesses per lookup
  - Caching all iNodes in memory might help, but iNodes are big
- Fundamentally, Photo Storage different from other storage:
  - Normal file systems fine for developers, databases, etc

New Solution: Haystack



- Finding a needle (old photo) in Haystack
- Differentiate between old and new photos
  - How? By looking at "Writeable" vs "Read-only" volumes
  - New Photos go to Writeable volumes
- Directory: Help locate photos
  - Name (URL) of photo has embedded volume and photo ID
- Let CDN or Haystack Cache Serve new photos
  - rather than forwarding them to Writeable volumes
- Haystack Store: Multiple "Physical Volumes"
  - Physical volume is large file (100 GB) which stores millions of photos
  - Data Accessed by Volume ID with offset into file
  - Since Physical Volumes are large files, use XFS which is optimized for large files

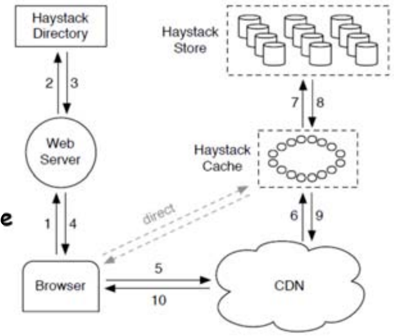
Haystack Details



Field	Explanation
Header	Magic number used for recovery
Cookie	Random number to mitigate brute force lookups
Key	64-bit photo id
Alternate key	32-bit supplemental id
Flags	Signifies deleted status
Size	Data size
Data	The actual photo data
Footer	Magic number for recovery
Data Checksum	Used to check integrity
Padding	Total needle size is aligned to 8 bytes

- Each physical volume is stored as single file in XFS
  - Superblock: General information about the volume
  - Each photo (a "needle") stored by appending to file
- Needles stored sequentially in file
  - Naming: [Volume ID, Key, Alternate Key, Cookie]
  - Cookie: random value to avoid guessing attacks
  - Key: Unique 64-bit photo ID
  - Alternate Key: four different sizes, 'n', 'a', 's', 't'
- Deleted Needle Simply marked as "deleted"
  - Overwritten Needle – new version appended at end

Haystack Details (Con't)



- Replication for reliability and performance:
  - Multiple physical volumes combined into logical volume
    - » Factor of 3
  - Four different sizes
    - » Thumbnails, Small, Medium, Large
- Lookup
  - User requests Webpage
  - Webserver returns URL of form:
    - » http://<CDN>/<Cache>/<Machine id>/<Logical volume,photo>
    - » Possibly reference cache only if old image
  - CDN will strip off CDN reference if missing, forward to cache
  - Cache will strip off cache reference and forward to Store
- In-memory index on Store for each volume map:

  [Key, Alternate Key] ⇒ Offset

Summary (2/2)

- 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
- Application-Specific File Systems (e.g. Haystack):
  - Optimize system for particular usage pattern