Manage Big Data in the Cloud -- First Set of Systems

Google File System

- Design goal of GFS
  - Scalable, distributed file system to support growing data processing needs of Google
    - Multi-GB files and billions of objects
- Observations
  - Failures are the norm out of 1000s of components ⇒ Need monitoring, error detection, fault tolerance, recovery
    - Bugs, human errors, and hardware failures
  - Bandwidth is more important than latency
    - Random writes are practically nonexistent
    - Many files are written once, and read sequentially
    - Two types of read, large streaming reads and small random reads

GFS Architecture

- GFS cluster
  - One master server (state replicated on backups)
    - Provides naming service, maintains access control info
    - Controls system-wide activities such as garbage collection
    - Manage chunk servers and their status
  - Many chunk servers (100s – 1000s)
    - Each chunk is stored as a regular Unix files
    - Each chunk is replicated, generally on 3 chunk servers
    - Chunk: 64 MB with 64-bit globally unique ID & 64 bits checksum
      - Spread across racks (bandwidth: intra-rack > inter-rack)
    - Advantages of large sized file blocks
      - Reduced client-master communication, once a chunk is located, client can perform read/write on the same chunk for a long time
      - Smaller size of metadata stored on master

Used by almost all cloud storage systems

GFS Architecture

- GFS cluster architecture

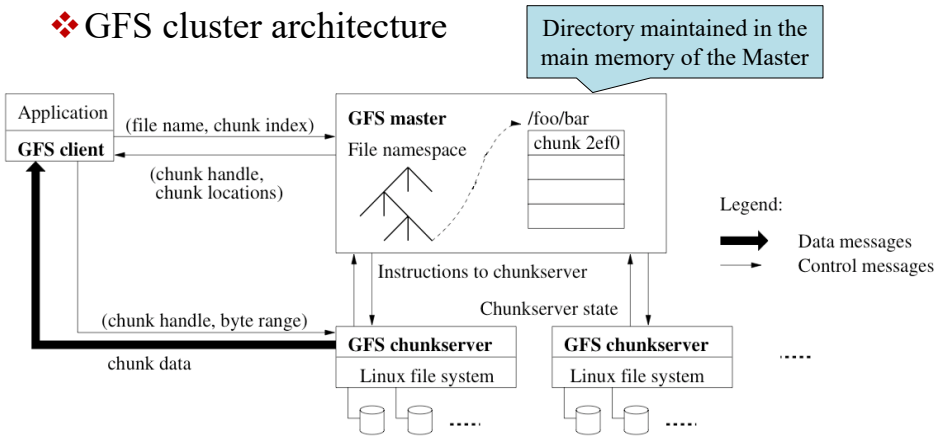


Figure 1: GFS Architecture

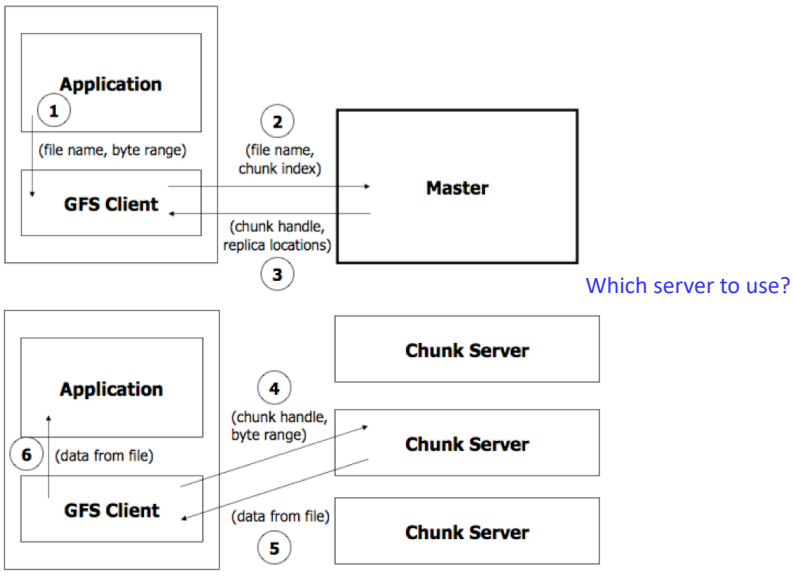
Access Protocols

- Update chunks
  - Primary-backup to assure sequential write orders
- Read chunks
  - Read any replica
- Atomic record append
  - Similar logic as write, but add additional logic
    - Client cannot decide where to append, may have concurrent append
- File delete
- Snapshot
- Atomic metadata update

Read Protocol

- Read chunks
  - Application invokes GFS client, which then checks local cache to find chunk location
  - If not cached, then
    - Client sends to master: read(file name, chunk index)
      - Chunk index = offset / chunk size
    - Master replies: chunk ID, chunk version, replica locations
  - In all cases
    - Client sends to the "closest" chunk server: read(chunk ID, byte range)
      - "Closest" is determined by IP address on simple rack-based network topology
    - Chunk server sends back the data

Read Protocol



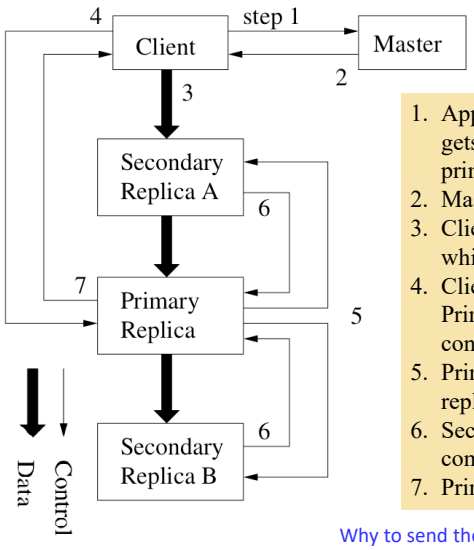
Write Protocol

- Update chunks

GFS uses primary-back, why this protocol?

  - Master grants lease to a replica (typically for 60 sec.)
    - Leases renewed using periodic heartbeat messages
  - Update is performed by the primary-backup model
    - Primary decides the order of updates
  - Two level consistent ordering of updates
    - Master lease and primary ordering
  - Failures of the primary can be recovered by leasing to another node
    - Timestamp is used for conflict resolution in the case of failures

Write Protocol



A passive replication protocol

1. Application invokes GFS client, which then gets the list of chunk servers and the primary from the Master (including IPs)
2. Master replies with the information
3. Client sends the chunk to all replicas, which gets cached by the replicas
4. Client sends a write request to the primary; Primary logs the request; If there are concurrent updates, primary orders them
5. Primary forwards the write request(s) to all replicas
6. Secondaries reply to primary indicating to commit the operation
7. Primary replies to the client

Why to send the updated chunk first, then update protocol?

Write Protocol

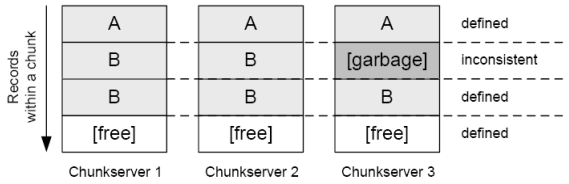
- Protocol
  - Client sends the chunk to all replicas
    - Data is sent in a pipeline
      - In this stage, does not consider primary or backup
      - Client sends to nearest server, say S1
      - S1 forwards the data to the server that is nearest to S1, say S2
      - S2 forwards the data to the next server …
  - Primary replies to the client
    - After received the commit messages from all secondary servers, primary replies with a success message
      - Does GFS allow group commit for chunk write? (not documented)
      - Assume it does for better performance
    - If some secondary servers fail to respond, then primary responds to the client with an error
    - Client may retry to write upon error

Record Append Protocol

- Protocol
  - Application invokes GFS client, which then pushes data to all servers of the last chunk
  - If the data can fit in the last chunk
    - Similar control flow as for write
    - Primary gives secondaries the byte offset in the chunk
    - The order for concurrent appends is defined by the primary
    - Primary responds to the client about success or failure
  - If the data cannot fit in the last chunk
    - Primary fills the chunk with padding and tells secondaries to do the same
    - Primary replies to client, “retry” (retry on next trunk)

Record Append Protocol

- If a record append fails at any replica
  - Client has to retry
  - Record may have already been written to some replicas
  - Client retries, the record got written again
  - GFS gives "at least once" guarantee



File Delete

- Client sends delete request to Master
- Master
  - Records deletion in its log
  - File is renamed to a hidden name, add a deletion timestamp
- Scan and removal
  - Master scans file namespace in background
    - Removes files with hidden names if deleted for longer than 3 days (configurable)
  - Master scans chunk namespace in background
    - Removes unreferenced chunks from chunk servers

Metadata Update

- Metadata
  - Only the master has the metadata information
    - Has directory path names in the master metadata, but no actual directory file that contains all files in the directory
  - Atomic metadata update
    - By locking and logging
  - No persistency protocol for chunk location data
    - Chunk server keeps track of what it hosts, can always resolve inconsistency with the Master or a backup, if Master fails

Metadata: directory structure, not data whereabouts
Protocol: master-backup
logging on master and backup, commit to client after logging
periodically checkpoint (i.e., merge log to previous checkpoint)
Concurrent update on master: use locking, what to lock is important

Metadata Update

- Master maintains all metadata
  - Stored in main memory
    - In B-tree like form, facilitate fast search
    - Use prefix compression
  - Atomic metadata update protocol
    - Master logs new metadata updates to its own disk
      - Ordered sequentially
      - Can flush multiple logs to improve performance
    - Master replicates log entries to remote backup servers
    - Backup servers store the logs on disks and confirm with master
    - The update is committed and the master replies to client only after log entries are safe on the disks of the master and backups

Metadata Update

- Master maintains all metadata
  - If master fails, metadata can be recovered from the logs
    - But recovery can be too inefficient if the log is long
  - Master periodically checkpoint the metadata on disk
    - Checkpoint is organized in B-tree form, so can be read in quickly to create the in-memory metadata (which is in B-tree form)
    - Handling regular traffic to the master at a higher priority
    - Upon checkpointing, first create a new log file
      - One thread performs logging, log new updates to the new file
      - A background thread performs checkpointing
        - » Merge the previous log into a previous checkpoint
        - » Checkpointing for a few million files can be done in a minute
        - » Delete the older checkpoint and log file

Metadata Update

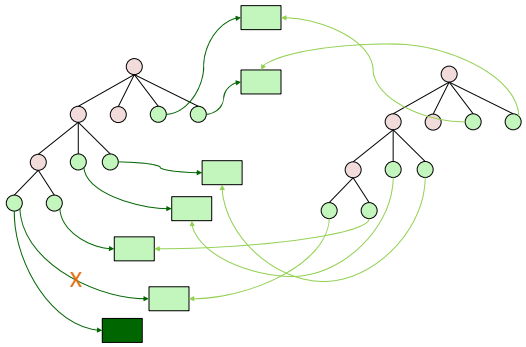
- Locking for metadata updates
  - What is to be locked, why???
    - Each metadata entry has a read-write lock
    - Update to a metadata node
      - Read lock on parent directories, write lock on the node
  - Example 1
    - Create or delete a file /home/usr/foo
      - Read lock /home, /home/usr; write lock /home/usr/foo
      - Read lock can prevent the directory from being deleted
      - No need to write lock /home/usr, because there is no directory content
      - Multiple files under the same directory can be created concurrently, but no two files with the same name can be created concurrently

Later:
Compare with Ceph's metadata update

Snapshot

- Handled using copy-on-write Used by almost all cloud storage systems
  - Reuse the technique from Andrew file system (AFS)
  - Revoke all leases for all chunks
    - To prevent new writes to go on (no primary, no write)
  - Duplicate the metadata
    - The duplicated metadata and all the chunks it points to form the snapshot (but chunks are not copied to the snapshot space)
  - Release leases (lease can be renewed)
  - Track new updates after the snapshot is taken
    - Updates to the chunks ⇒ Snapshot becomes inconsistent
    - Write is performed with the copy-on-write protocol
    - Copy the original block to the snapshot space and the snapshot link is updated to point to the copy
    - Update can now be performed on the original block

Snapshot



Fault Tolerance and Recovery

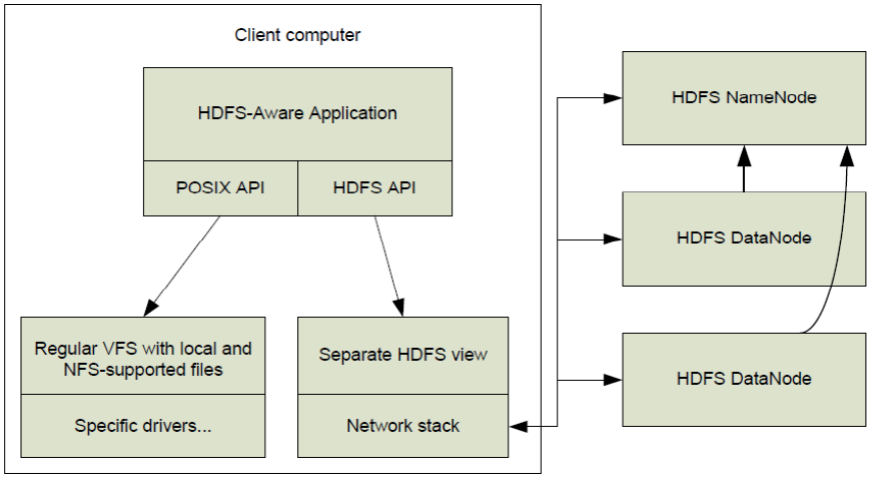
- Fast recovery
  - Master and chunk server are designed to restore their states and start in seconds
- Chunk replication makes chunk recovery easy
- Master backup (logs)
  - Shadow masters provide read-only access when the primary master is down
    - i.e., allows read, not write to the metadata
    - Backup may be lagging, this way can avoid complex consistency recovery protocols

Replica Placement

- When a new chunk is created, replicas are placed based on the following factors
  - Chunk servers with below average disk utilization
  - Avoid chunk server with many new chunk creation
    - Chunk creation requires a lot of data transfer ⇒ Busy server
  - Try to distribute replicas across multiple racks
- Master re-replicates a chunk
  - When #replicas falls below a user-specified threshold
- Master migrates chunk replicas periodically
  - To balance the workload and disk space utilization

Hadoop HDFS

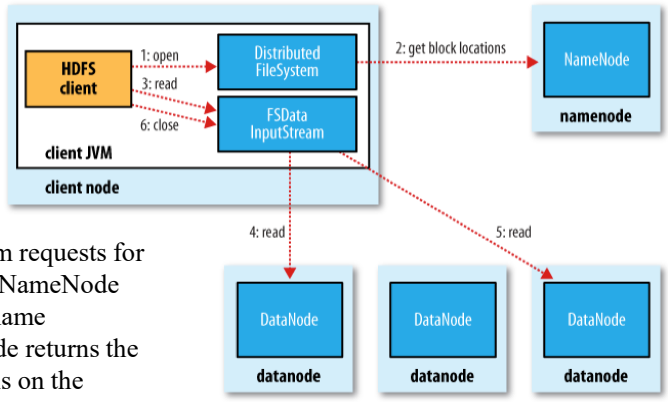
- Similar to GFS, some discrepancies



Hadoop HDFS

- Similar to GFS, some discrepancies
- Architecture
  - Namenode (Master)
  - Datanode (Chunk server)
    - Store data blocks (chunks)
  - Client

HDFS Read

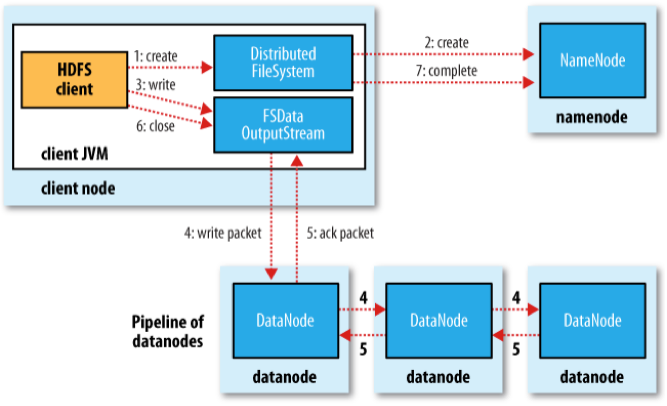


- Client Program requests for data from the NameNode using the filename
- The NameNode returns the block locations on the DataNodes
- The Client then accesses each block individually
  - Can have parallel access if on different nodes

Replica Selection

- Replica selection for read
  - Principle: Minimize bandwidth consumption and latency
  - Distance from node n1 on rack r1 in data center d1 (d1/r1/n1)
    - distance(/d1/r1/n1 , /d1/r1/n1) = 0 -- same node
    - distance(/d1/r1/n1 , /d1/r1/n2) = 2 -- same rack, different node
    - distance(/d1/r1/n1 , /d1/r2/n3) = 4 -- same DC, different rack
    - distance(/d1/r1/n1 , /d2/r3/n4) = 6 -- cross DC
  - If configured properly to provide this information, Hadoop will optimize file reads and writes and select the replica closest to the client

HDFS Write



HDFS Write

- File creation
  - HDFS client writes the data into cache, when it reaches HDFS block size, the HDFS client contacts the Namenode
    - Namenode inserts the file-name/block-id into its hierarchy and allocates DataNodes for the block and the replicas
  - The DataNodes form a pipeline, in the order to minimize the total network distance
  - The client puts data to DataNodes
    - Data is sent in 4KB packets
    - Each data packet is acknowledged, but client can continue to send data without waiting for the ACKs
    - Resend packets for missing ACKs after timeout

HDFS Write

- File creation
  - After finish pushing all packets to DataNodes and obtained ACKs for all packets
    - The client tells the NameNode that the block is written and closed
  - Namenode commits the file creation and log in the journal
    - If the Namenode dies before the file is closed, the file is lost
- Similar protocol for write
  - Always writing to a new block (client cache the original and update it by write-on-close)
- Only support write-append
  - Cannot write at any offset

HDFS Write

- **Single writer, exclusive write** Will there be deadlock?
  - Client needs to acquire the lock (lease) from NameNode
    - The lock needs to be maintained by having an Alive message
    - Have a soft deadline for the lock (1 minute) Why soft deadline?
      - If soft deadline expires before the client finishes writing all the data, other clients can preempt the current write
    - Have a hard deadline Why hard deadline?
      - If hard deadline expires (1 hour) and writer does not renew the lock ⇒ Assume client is dead, close the file, release the lock
    - Write lock at the file level, not the block level ⇒ Consistent when writing multiple blocks of one file ⇒ but lose concurrency
  - Reader does not care about write locks
- **Write on close for each block** What is the purpose of this?
    - Client can write on its cached block, write only go to DataNode when the block is closed

Replica Placement

- HDFS considers optimal replica placement
  - Is it really optimal?
- Goal: Improve reliability, availability and network bandwidth utilization
- Replicas are placed
  - One on a node in a local rack, one on a different node in the local rack, and one on a node in a different rack

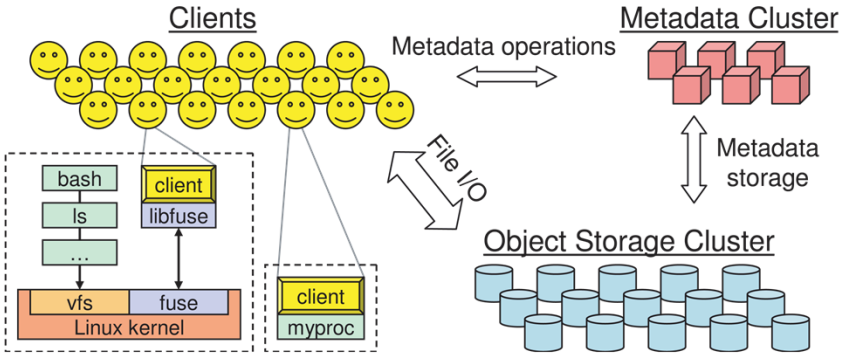
Ceph

- Major features in Ceph
  - Object-based storage
    - Files are split into objects (just like blocks) and object is the unit of storage
  - Separation of data and metadata
    - Metadata operations are independent to file accesses
      - E.g., ls, cd, etc. has nothing to do with open/read foo.dat
    - Layout of the files on distributed nodes can be independent of metadata management (done by OSD)
  - CRUSH for data distribution
    - Absolutely no directory management needed, and do not have the problem of simple hashing
      - Simple hashing: node = h(x) % N, massive data movement with changed N

Ceph

- Ceph system architecture

MDS (metadata servers)
- Maintain directories for file systems



OSD (object storage daemon)
- Store files/objects
- Maintain OSD Map

Object Placement on OSDs

- Client accesses a file
  - Sends a request to the MDS cluster
  - MDS returns
    - i-node number, file size, layout and striping strategy, capability
  - Compute OSD index based on object id, cluster map, and replica placement rules
    - Object id = (file id, object number)
      - Object number is computed based on layout and striping strategy
    - OSD cluster map
      - A cluster is the set of physical nodes for a file system
    - Replica placement rules
      - E.g., level of replication, place two replicas in different fault groups, in different data centers, etc.

CRUSH - RUSH

- Rush (replication under scalable hashing)
  - Predecessor of Crush
    - Target to solve the massive file migration problem in simple deterministic hashing due to OSD addition or deletion
  - Topology
    - In a cluster, consider a list of sub-clusters
    - When a new sub-cluster is added to the cluster
      - Add it to the head of the list
      - E.g., a system (C_2, C_1, C_0), -- C_0 is the oldest sub-cluster
      - A new cluster C_3 is added, the system becomes (C_3, C_2, C_1, C_0)
      - Now some objects needs to be moved from (C_2, C_1, C_0) to C_3 ⇒ How to do it so that object migration is minimized and search method stays the same

CRUSH - RUSH

- Rush (replication under scalable hashing)
  - Allocation among sub-clusters in a list
    - Allocate to C_3 if $h\ (x, r, cid(C_3)) < {}^{W(C_3)}/_{\sum_{i=0}^{3} W(C_i)}$
      - $h(x, r, cid)$ is in [0,1)
      - x: file name, r: replica id, *cid:* id of the sub-cluster
      - $cid(C_3)$ is used only for making hashing more random
      - W(C_i) is the weight of sub-cluster C_i, which is determined by its capacity (disk capacity, CPU power, …)
    - Go on recursively down the list till allocation is determined

Node:	C3	C2	C1	C0	weight = 1 for all nodes
Prob:	1/4	1/4	1/4	1/4	h(x, 3) = 0.5 > 1/4 ⇒ Not on C3
		1/3	1/3	1/3	h(x, 2) = 0.6 > 1/3 ⇒ Not on C2
			1/2	1/2	h(x, 1) = 0.3 < 1/2 ⇒ allocate to C1

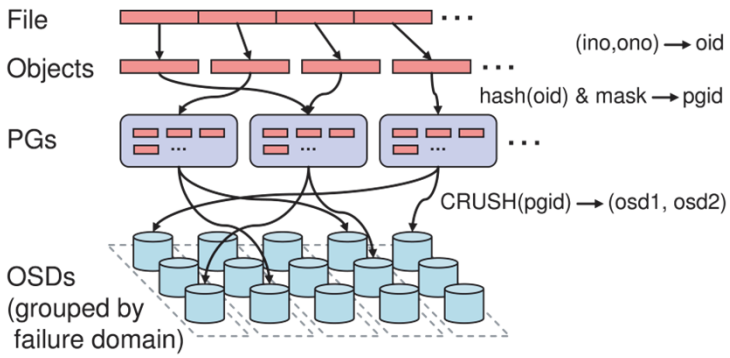
If h(x, 1) > 1/2 ⇒ Not on C1, only choice is C0

CRUSH - RUSH

- Rush
  - Allocation based on a tree
    - The list based approach is not scalable, so consider organizing sub-clusters as a tree
    - If $h\ (x, r, cid) < \frac{W(C_{left})}{W(C_{left})+W(C_{right})}$ then go to left subtree; otherwise, go to right subtree
      - Could have more child nodes from each parent
    - Recursively go down the tree till a leaf node is reached
    - *cid* is the cluster id of the parent

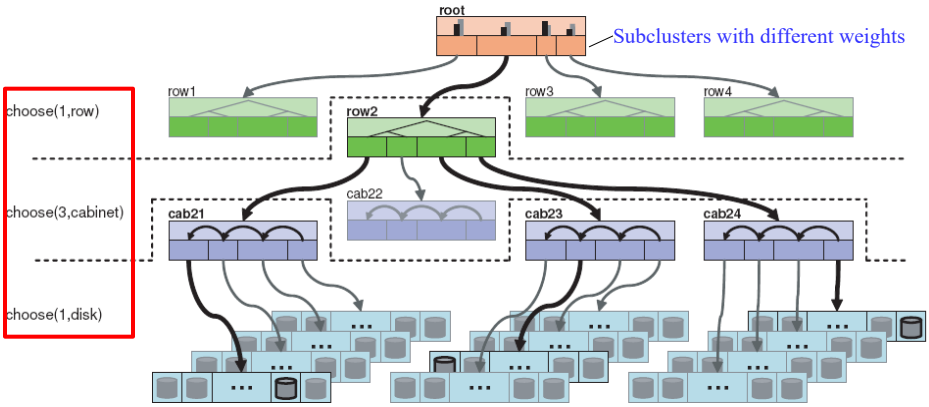
CRUSH

- Crush (Controlled Rush)
  - Objects are hashed into placement groups (PGs)
    - PG is just the hash value, or the virtual node
    - Each OSD is roughly mapped to 100s of PGs



CRUSH

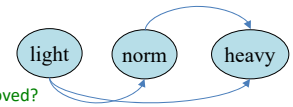
- Placement of PGs to OSDs
  - Done hierarchically according to the cluster map and …
    - Cluster map is the OSD hierarchy (as below), OSDs form shelves, shelves form cabinets, then rows, …



CRUSH – OSD Failure and Overload

- Crush resharding
  - Consider failed and overloaded nodes
    - Failed nodes are still in the map, to avoid massive migration
    - Failed nodes are marked, if selected, a reselection will be done
    - Failed replica
      - Consider *f* failed attempts (hit failed nodes)
      - Allocate for replicas *n*+1, …, *n*+*f*
        - » Do exactly the same allocation in any type of bucket, starting from the root (to best balance the load)
    - Load balancing
      - Change weights
        - » Induce more movements than necessary
      - Virtual node???

Change weights in Ceph is not dynamic
Manually editing a CRUSH Map — Ceph Documentation



How many more files will be moved?

CRUSH - RUSH

- Rush-Crush benefits?
  - File movement characteristics for adding/deleting a node
    - #files to be moved???
    - How to know which files should be moved???
    - Consider the expected computation and communication overhead
      - Computation: rehashing computation
      - Communication: #file to be moved
    - Consider the same for the DHT ring and other DHT mechanisms we will cover and compare them
  - File movement characteristics for load balancing

OSD Cluster Map Management

- Ceph monitor(s)
  - http://ceph.com/docs/master/rados/configuration/mon-config-ref/
  - Maintain the OSD cluster map
    - In paper, no spec of monitor (a later decision), and no spec on who shall host the cluster map
  - Provide authentication service
  - At least one monitor in a Ceph cluster, can have more for fault tolerance and load sharing
  - Consistency of the map on multiple monitors is maintained through Paxos
    - https://en.wikipedia.org/wiki/Paxos_(computer_science)
    - http://libpaxos.sourceforge.net/
  - Ceph client knows how to contact "all" Ceph monitors
  - Ceph client caches the OSD cluster map

OSD Cluster Map Management

- Map updates
  - An epoch number is maintained for the map
    - Each time the map changes, the epoch number is incremented
    - Essentially version number
  - Updates are piggybacked to the clients
    - Each client data request sent to OSD should be tagged with the epoch number of its cluster map
    - If the client map is outdated (from epoch number), the updates are piggybacked with the response message
  - Updates need to be spread to OSDs
    - Each update requires the update of the epoch number stored on the OSDs (to check the epoch number sent by the client and to know who to transfer files to upon failure or load balancing)
    - How it is spread is not specified

Ceph IO and Consistency

- Follow POSIX semantics
  - Atomic register, fully synchronized
  - Managed by RADOS: Ceph's Reliable Autonomic Distributed Object Store
- Read with cache
  - MDS grants the client the capability to read and cache
    - File content may be cached at the client side for further reading
  - Client also caches file metadata
    - With inode number, layout, file size, OSD map $\Rightarrow$ Can locate all objects
  - Read object directly from the OSD, if not cached

Ceph IO and Consistency

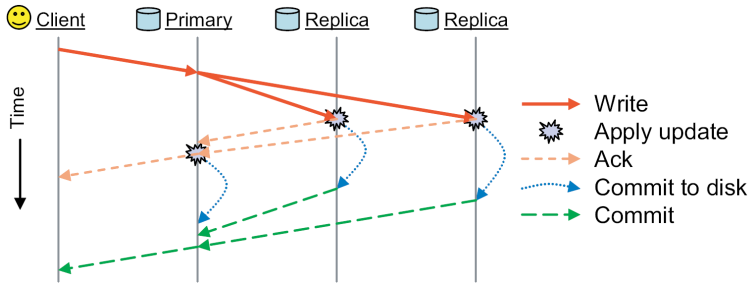
- Write with buffer
  - MDS grants the capability to write with buffering
    - Write is always at OSD, but may be in OSD's memory buffer
    - Since there are replicas, buffer flush can be delayed without losing write information
    - Write buffer off/on ≅ Regular files and shared files on Unix
- When a write spans multiple objects
  - Lock all the objects involved, but lock happens after the data to be written are cached at the OSDs
    - Here the lock is for the buffered writes

Ceph IO and Consistency

- Write on replicas
  - Sent to the first non-failed OSD (the primary)
  - Primary assigns a new version number for the object
  - Primary forwards the write to secondary replica OSDs
  - Secondaries reply to the primary
  - Primary sends ack to the client
- Read on replicas
  - Also sent to the primary
- Advantages
  - Spares the client of any synchronization details
  - Confine the bandwidth consumed by replication in OSD internal network, where greater resources are available

Ceph IO and Consistency

- Write on replicas
  - Safe commit and ack
    - Primary responds with an ack after the write has been applied to the buffer caches on all OSDs with replicas
    - Safe commit only after writes are committed to disk



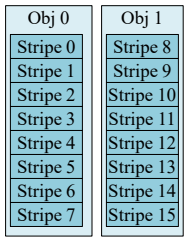
Metadata Servers

- Metadata servers (MDS)
  - Metadata accesses may make up half of file system workloads
  - Distribute workload over 10s to 1000s MDSs
  - Metadata does not contain object placements info
  - Also managed by RADOS
    - Reliable Autonomic Distributed Object Store
    - Directories are in both memory and the file system
  - Update ⇒ atomic update of metadata and data
    - E.g., create a new file and write data into it
      ⇒ Need to update object storage and metadata storage atomically

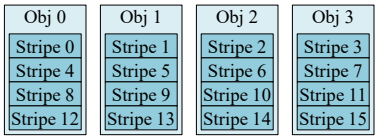
Metadata Fields

- Metadata of a file
  - Immutable fields
    - inode number: a unique identifier for the file
    - Creation time (ctime)
    - Layout
      - object_size (integer in bytes): Object size
        - » Object is the unit for allocation (like chunk)
      - stripe_unit (integer in bytes): Block size for striping
      - stripe_count:
        - » Number of consecutive blocks in a stripe
    - Stripe type
      - A. Stripe consecutively within each object
      - B. Stripe into multiple objects in one "object set"

Stripe Type A: e.g.
Block size = 64KB
Object size = 8 blocks
File size = 16 blocks



Stripe Type B: e.g.
Block = 64KB, Object = 8 blocks, File = 16 blocks



Metadata Fields

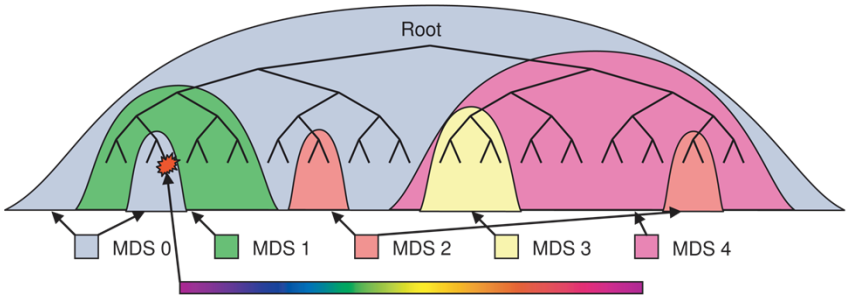
- Metadata of a file
  - Security fields: Can be changed, but rarely changed
    - Owner
    - Capability
      - Specifies authorized operations on file
      - Currently 4 bits: Read, Cache reads, Write, Buffer writes
        - » Support control of whether caching/buffering are allowed for r/w
    - Used for the security check during path traversal
    - …
  - Frequently changing fields
    - Size, mtime (last modified time)
      - Need to be changed during file writes
      - When there are concurrent writes of multiple objects of a file, …
    - …

Metadata Storage Structure

- Existing metadata storage solutions
  - Coarse grained
    - Existing solutions are static partitioning
    - ⇒ Cannot account for workload changes
  - Fine grained
    - Same as Unix systems, treat each directory as a file
    - Each directory file is hashed to specific locations, like other files
    - ⇒ Lose locality for path traversal

Metadata Storage Structure

- Adaptive scheme: coarse + fine grained
  - Partition the metadata tree into subtrees
    - Dynamic partitioning based on workloads
    - Preserve access locality by coarse grained partitioning
  - Avoiding hot spots
    - A very busy directory may be replicated by hashing (like a file)



Metadata Storage Structure

- MDS infrastructure
  - Maintains the metadata partitions
    - Partitions are distributed over multiple MDSs
    - A heavily read partition is replicated on multiple MDSs
    - Authority for a partition: like the primary for write
  - Tasks of the MDSs
    - Process metadata requests from clients and OSDs
      - E.g., Metadata read, such as readdir, stat
      - E.g., Metadata update, such as change last modified, file size, etc.
      - All operations on metadata are serialized
    - Performs access control based on security mode of the file and issues capability to the client for file/directory accesses

Metadata Access

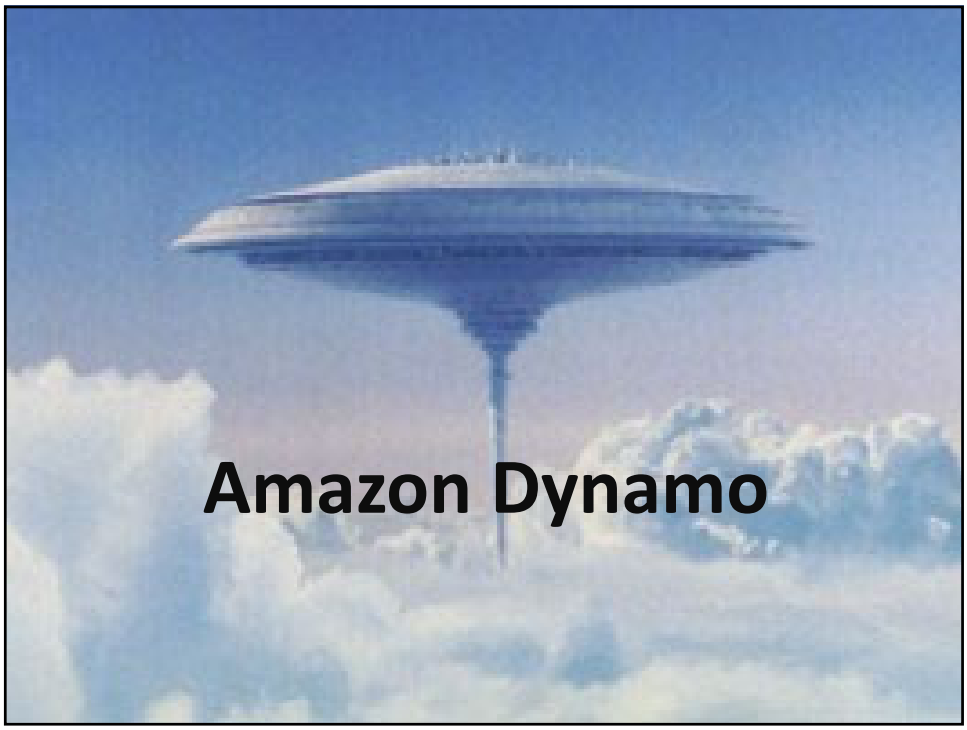
- Which MDS hosts the specific directory info?
  - MDS that hosts root is known to the clients/OSDs
  - For each request, starts from the root?
    - Partitioning does not help reduce load
  - Client/OSD cache the MDSs for metadata
    - For a certain inode, cloud be a file or a directory path
      - Cache its authority (for write) and the replicas
      - Cache the MDSs of the ancestors of the inode
  - Future metadata operations
    - Find the authority or a replica based on the deepest known prefix of the given path
      - E.g., update directory /a/b/c/d/e/f/g
      - Cache contains: authority MDS for /a/b/c/ = X $\Rightarrow$ start from X

Metadata Read/Write Consistency

- Consistency
  - Serialize read/write by locking
    - Unlike GFS, need to consider directory update
  - Need to consider replicas + cached copies on clients
- Authority and collaborative caching
    - Also called the primary-copy caching strategy
  - "Authority" MDS for each metadata piece
    - Responsible for replica consistency and cache coherence
    - For replicas: Order the access requests on a metadata piece
    - For each inode, maintain the list of clients with a cached copy
      - Client obtains a cached copy via authority
      - Client informs authority if a cached copy is deleted (collaborative)
      - Authority informs clients if metadata for the inode changes

Metadata Read/Write Consistency

- Complex metadata update
  - Example: "stat" on a currently opened file
    - Opened by multiple clients for writing
      - MDS revokes all write capabilities ⇒ Stop updates
      - MDS collect up-to-date size and m-time from all writers
        - » Each write may be performed on different objects of the file
        - » Multiple writes ⇒ cumulated size + latest modification
      - The highest values are returned as the stat reply
      - Capabilities are reissued after stat is responded
    - Opened by a single writer
      - No need to stop the update
    - Relaxation
      - Can use statlite to specify which fields of the metadata do not require coherence



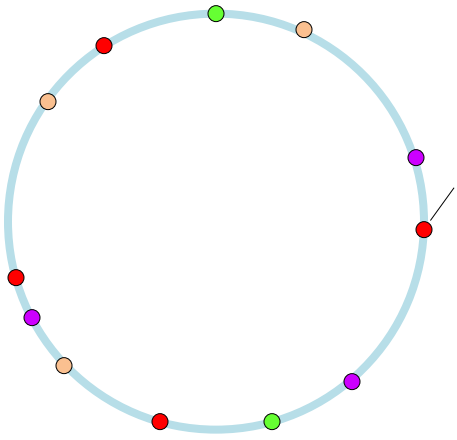
Dynamo

- Amazon's highly available key-value store
  - Dynamo: Amazon's Highly Available Key-value Store
  - Simple data structure: key-object-context
    - Simple read and write to data that are accessed by key only
      - No access involves multiple data entries
    - Object is the data to be stored
    - Context includes the timestamp and history (for conflict resolution)
  - Highly available
    - Always replicate
  - Guarantee service level agreements (SLA)
    - Response time measured at the 99 percentile
    - 99% of the requests will satisfy the response time bound at a maximum load of 500 req/sec

Dynamo: DHT

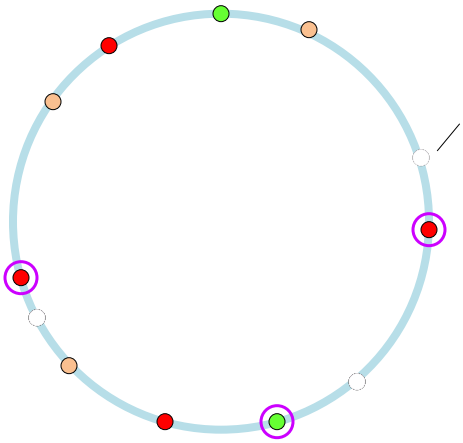
- Use a variation of DHT ring
  - DHT ring consists of virtual nodes
  - Virtual nodes to physical nodes mapping
    - Each physical node is mapped to multiple ids (virtual nodes)
  - Load balancing
    - More likely to have balanced load than the original case
    - If a node becomes unavailable, its load is dispersed across many physical nodes, instead of just the successor
    - When a new node is added, it can accept the load from many other nodes, instead of just from the successor
    - Consider heterogeneity in capacity: The number of virtual nodes that a node is responsible for is decided based on its capacity
    - Can use add/delete virtual nodes to balance load

Dynamo: DHT



One physical node is hashed to multiple virtual physical nodes
⇒ Easier in load balancing

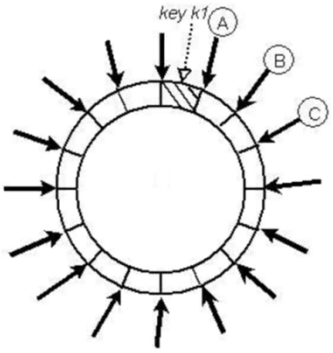
Dynamo: DHT



One physical node is hashed to multiple virtual physical nodes ⇒ Load of a failed node is distributed across multiple physical nodes

Dynamo DHT

- Virtual node to an extend
  - Ring is partitioned to Q equal sized slots (each slot has a token)
  - Each node randomly picks Q/N tokens
    - N: number of nodes
  - Add node
    - Takes tokens evenly from existing nodes
  - Delete node
    - Distribute tokens evenly to remaining nodes

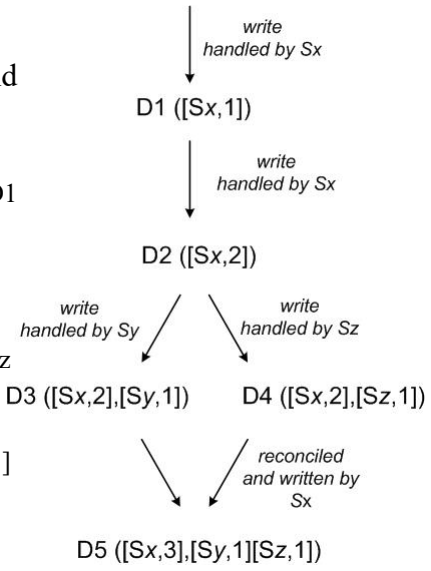


Dynamo: Access Protocol

- Consider eventual consistency
  - Use a vector timestamp (size N, indexed by node id)
- Sloppy quorum protocol
  - Configurable N, R, W
  - Principle of the protocol
    - Customer update has to be successful
    - Write should never be rejected, always allow write
    - Resolve conflicts during read
  - Always write to N copies and read from N copies
    - But only need to write/read available copies
    - At least write to W copies and at least read from R copies
    - Does not have to have R + W > N

Dynamo: Access Protocol

- Timestamp illustration
  - The node directly received and handled the write gives the timestamp
    - E.g., D1[Sx,1] means version D1 is handled by Sx and Sx gives time stamp 1 to D1
      - You can consider [Sx,1] as [Sx,1][Sy,0][Sz,0]
    - Sx will forward D1 to Sy and Sz
    - When Sy writes D3, Sy has already received D2[Sx,2], so timestamp becomes [Sx,2][Sy,1]



Dynamo: Access Protocol

- Write: put(key, timestamp, object)
  - Write to all available replicas
    - Specify a timestamp (derived from an earlier read)
  - For an unavailable copy, use a hinted handoff protocol
    - Assume N = 3 and (A,B,C) is the preference list
    - If A is temporarily down, send replica to D, D is hinted that the replica belongs to A
      - D periodically scans its hinted list and check whether the owner of the data has recovered, and deliver the data if recovered
- Read: get(key)
  - Read N copies, if inconsistent
    - If there is an absolute maximum timestamp ⇒ Use that replica
    - Otherwise, the application can decide how to resolve conflicts

Dynamo: Failure Detection

- Failure detection
  - Localized observation (A considers B failed if B does not respond to A's message)
  - Use gossip protocol to piggyback the failure list
    - Eventual consistency in failure detection
- Membership information
  - Mark failed node as above, but transient
    - (no specific discussions)
  - Manuel membership change
    - Node addition and deletion command is handled by a coordinator
    - Use gossip protocol to propagate the changes

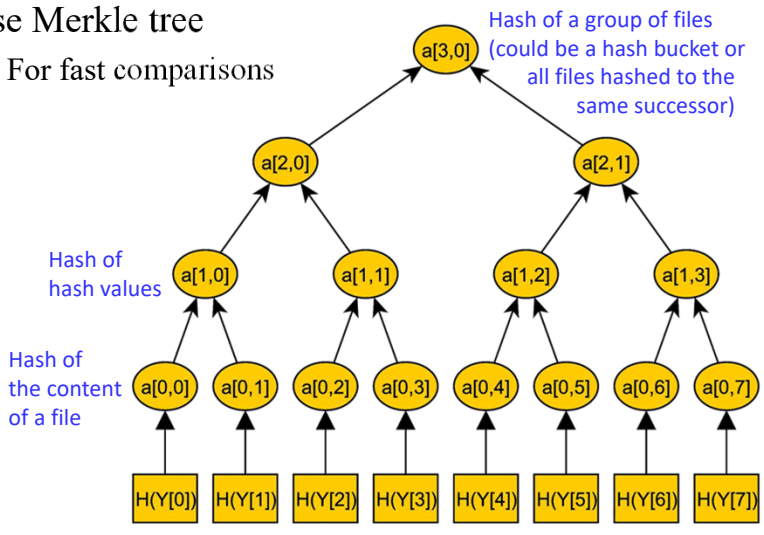
Automated detection does not seem to be available in Swift

Dynamo: Failure Recovery

- Hinted handoff can handle quick recoveries
  - Handoff replica may itself become unavailable
  - Failure for a long duration ⇒ Too many handoff replicas
- Replica synchronization
  - After a long-term failure
  - Detect inconsistency between replicas
  - Use Merkle tree for summary information exchange
    - First, only exchange hash values at the top of the tree
    - If inconsistent, recursively go down the tree and exchange the hash values of the top of the subtrees
    - Finally detect the inconsistent data objects and go for repair
      - Note: one tree for each set of replicas on the N nodes

Dynamo: Failure Recovery

- Use Merkle tree
  - For fast comparisons



Swift

- Open source implementation of Dynamo
  - Both Dynamo and Swift are initially designed to be an object storage (key value pairs)
  - But most use swift as a file system
    - Default file system in OpenStack
  - New swift has some enhancements
    - Three layer directory system
      - Accounts, containers, objects
        - » Account server stores and manages user accounts
        - » Container server stores and manages "directory" (containers)
        - » But container cannot be nested (single layer)
        - » Object server stores objects
      - Each of them has its own ring
    - Proxy: route read/write requests (same as Dynamo)

References

- GFS
  - Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, "The Google file system," ACM SOSP, 2003
- HDFS
  - K. Shvachko, Hairong Kuang, S. Radia, R. Chansler, "The Hadoop distributed file system," ACM Symposium on Mass Storage Systems and Technologies, May 2010, pp. 1-10.
- Amazon Dynamo
  - Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels, "Dynamo: Amazon's highly available key-value store," ACM SOSP, 2007.

References

- Ceph
  - Ceph: A Scalable, High-Performance Distributed File System
  - CRUSH: Controlled, Scalable, Decentralized Placement of Replicated Data
  - Replication Under Scalable Hashing: A Family of Algorithms for Scalable Decentralized Data Distribution
  - A Fast Algorithm for Online Placement and Reorganization of Replicated Data
  - http://ceph.com/
  - Load balancing

    Change weight
    - Data-driven Ceph performance optimizations (Patent)
    - Mantle- A programmable metadata load balancer for the Ceph file system

      These two are read load balancer
    - https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/load_balancer_administration/ch-keepalived-overview-vsa