Big Data Storage I: Distributed File System, Google File System(GFS) & Colossus(GSF2)

Shiow-yang Wu (吳秀陽)

CSIE, NDHU, Taiwan, ROC

Lecture material is mostly home-grown, partly taken with permission and courtesy from Professor Shih-Wei Liao of NTU.

Outline

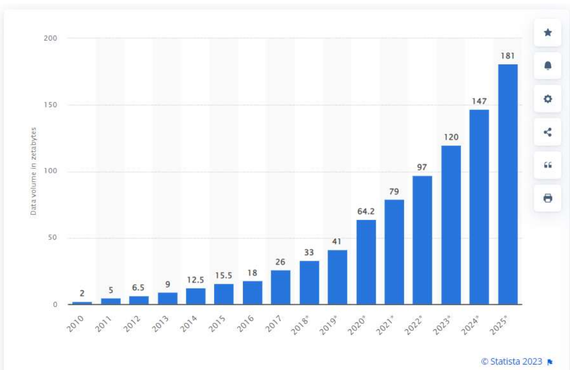
- Big data storage overview
- File systems overview
- Distributed File Systems (DFS)
- Google File System (GFS)
  - Motivations
  - Architecture
  - System Interactions
  - Fault Tolerance
  - Conclusion
- Colossus (GFS2)

The GFS Paper

- Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System, *SOSP'03,* October 19–22, 2003, Bolton Landing, New York, USA.
- Mysterious successor - GFS2: Colossus

Big Data Storage Challenges

- Data is exploding!
- Replication systems have security weaknesses
- RAID at petabyte scale leads to data loss
- Multiple copies equals multiple everything

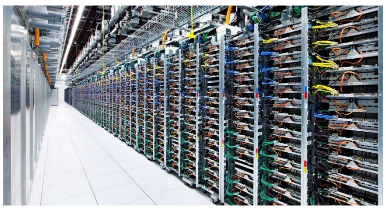


Problems with Traditional Storage

- Traditional storage architectures just weren't designed to handle big data.
- **They can't scale.**
- **They're not secure.**
- **They're not reliable.**
- **They're expensive.**

Storage Then and Now

Early days…

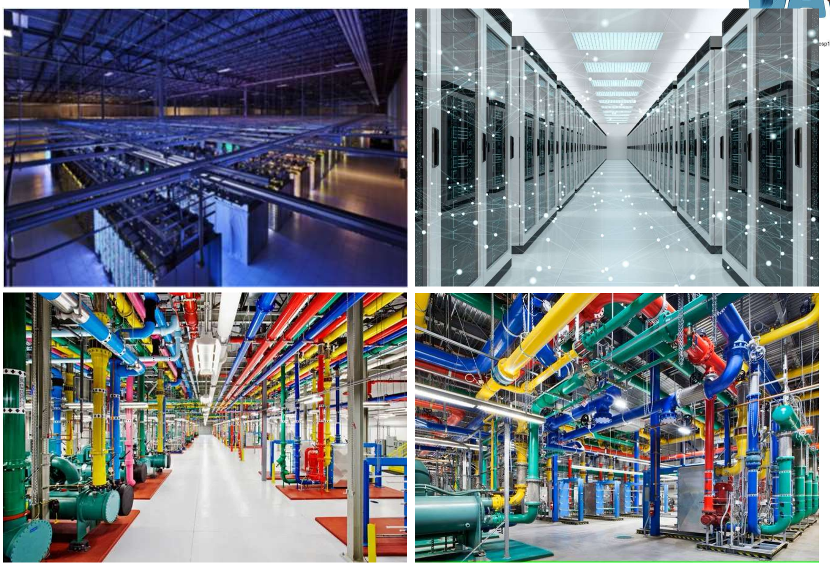


…today (Google datacenter at Pryor Oklahoma)

Google The Dalles Oregon

- Google The Dalles Oregon is it’s first data center.

Inside Google’s Data Center



World's Largest DC

China Telecom DC at Mongolia Information Park (10,763,910 total square footage with 42 data center buildings and 19 support buildings)

File System Overview

- Permanently stores data
- Usually layered on top of a lower-level (physical storage)
- Divided into logical units called "files"
  - Addressable by a filename ("foo.txt")
  - Usually supports hierarchical nesting (directories)
- A file path = relative (or absolute) directory + filename
  - /dir1/dir2/foo.txt

Distributed File Systems

- Support access to files on remote servers
- Must support concurrency
  - Make varying guarantees about locking, who "wins" with concurrent writes, etc...
  - Must gracefully handle dropped connections
- Can offer support for replication and local caching
- Different implementations offer different degrees of complexity / features

Motivation

- Need storage to support the crawling and indexing of the whole Web
- Store it all on "one big disk(array)"
- Process user searches on "one big CPU"

Doesn't scale!

Motivation

- Google needed a good distributed file system
  - Redundant storage of massive amounts of data on cheap and unreliable computers
- Why not use an existing file system?
  - Google's problems are different from anyone else's
    - Different workload and design priorities
  - GFS is designed for Google apps and workloads
  - Google apps are designed for GFS

Need for New Solutions

- Need storage server that supports (at then)
  - many 100TB of data
  - distributed along many 1000 servers of cheap hardware (scale-out vs scale up)
  - serving many 100 of clients at the same time
- The work load/problems were expected to be more and more severe as Google's business expanding quickly.

Design Assumptions

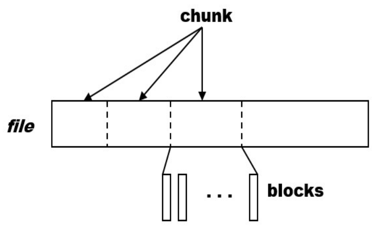
- Component failures are the norm
  - Built from 1000s of inexpensive commodity components
  - Constant monitoring, error detection, fault tolerance, automatic recovery must be integral to the system
- Stores "LARGE" files
  - A few millions files, multi-GB files are common
  - Need not optimize for small files
- Huge storage needs

Design Assumptions

- Files are write-once, mostly appended to
  - Perhaps concurrently to one file
- Workload
  - Large streaming reads (1MB+)
  - Small random reads (a few KBs)
  - Many large sequential writes that append
- High sustained throughput is more important than low latency

File Structure

- File
  - Divided into 64 MB chunks
- Chunk
  - Divided into 64 KB blocks
  - Replicated (default 3 replicas)
  - Identified by 64-bit handle
- Block
  - Has a 32-bit checksum

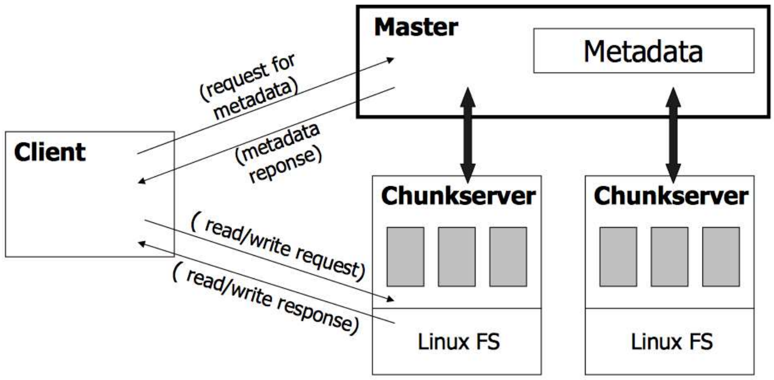


Chunk Size

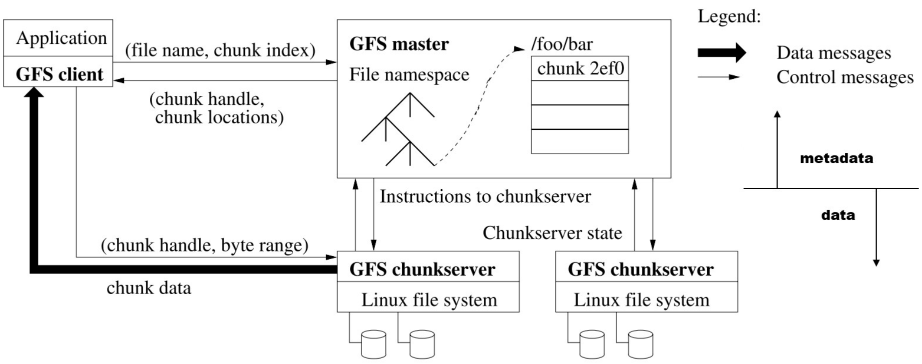
- 64MB
  - Much larger than typical file system block sizes
- Advantages from large chunk size
  - Reduce interaction between client and master
  - Client can perform many operations on a given chunk
    - Reduces network overhead by keeping persistent TCP connection
  - Reduce size of metadata stored on the master
    - The metadata can reside in memory

GFS Architecture

- Single Master (maintain metadata)
- Multiple chunkservers (store chunks)



GFS Architecture



Can anyone see a potential weakness in this design?

Master

- A single process running on a separate machine
  - Stores all metadata
  - File and chunk namespace(directory hierarchy)
  - File to chunk mappings
  - Chunk location information
  - Access control information
  - Chunk version numbers
  - Lease Management (more on this later)
  - Manage Garbage Collection
  - Stale Replica Detection
  - Periodically communicate with chunkservers(heartbeat)

Lease Management

- A crucial part of concurrent write/append operation
  - Design to minimize master's management overhead
- One lease per chunk
  - Granted to chunkserver, which becomes the primary
  - Granting a lease increases the version number of the chunk
- The primary can renew the lease before it expire(default 60s)
- The master can grant the lease to another replica if the current lease expires(primary crashed)

Garbage Collection(GC)

- Storage reclaimed lazily by GC
- File first renamed to a hidden name
- Hidden files removed if more than three days old
- When hidden file removed, in-memory metadata is removed
- Regularly scans chunk namespace, identifying orphaned chunks. These are removed.

Stale Replica Detection

- Whenever new lease granted, chunk version number is incremented
- A chunkserver that is down will not get the chunk version incremented
- The master removes stale replicas in its regular garbage collection

Heartbeat

- Master issues HeartBeat messages to chunkservers regularly
  - if too much strike, then you're out
  - give instructions: delete chunk, etc
  - collect chunk status: corrupt, possessed, etc.
- A chunkserver sends chunk IDs that it has, and get orphaned chunks in reply
- A chunkserver sends corrupt chunk ID

Single Master

- Single master
  - Global knowledge
  - Better placement / replication
  - Simplifies design
- Problems:
  - Single point of failure
  - Scalability bottleneck
- How to deal with the problems ?

Single Master

- GFS solutions:
  - Master log & checkpoints replicated
  - Outside monitor watches master livelihood
    - Starts new master process as needed
  - Shadow master
    - provide read-only access when primary is down
  - Minimize master involvement
    - never move data through it, use only for metadata
    - large chunk size
    - master delegates authority to primary replicas in data mutations (chunk leases)

Metadata

- Global metadata is stored on the master
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- All in memory (64 bytes / chunk)
  - Fast
  - Easily accessible

Metadata

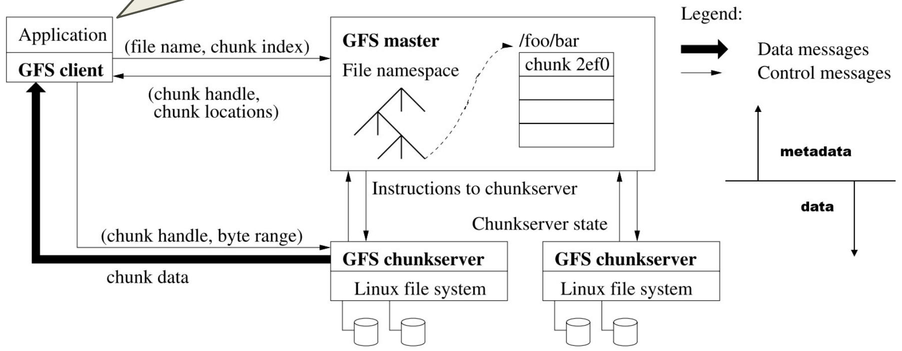
- Master has an operation log for persistent logging of critical metadata updates
  - Persistent on local disk
  - Replicated
  - Checkpoints for faster recovery

Chunkservers

- Stores 64 MB file chunks on local disk using standard Linux file system, each with version number and checksum
- Read/write requests specify chunk handle and byte range
- Chunks replicated on configurable number of chunkservers (default: 3)

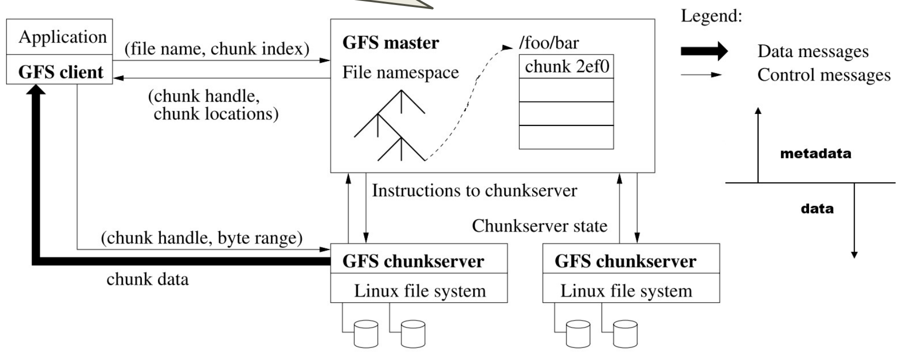
A Simple Read Example (1)

Using fixed chunk size, translate filename & byte offset to chunk index.



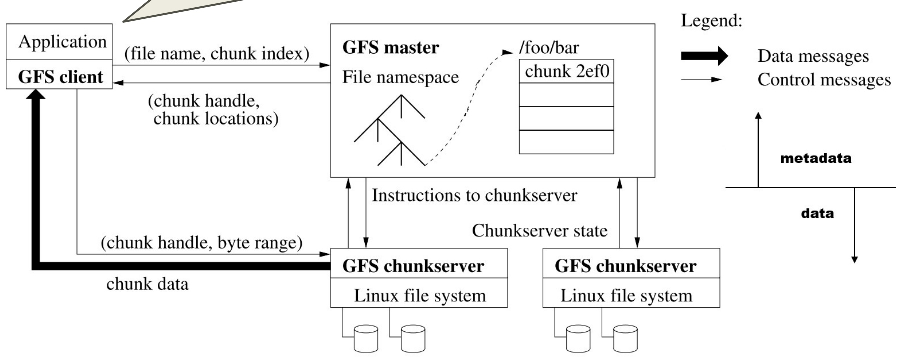
A Simple Read Example (2)

Replies with corresponding chunk handle & locations of replicas (including which is 'primary')

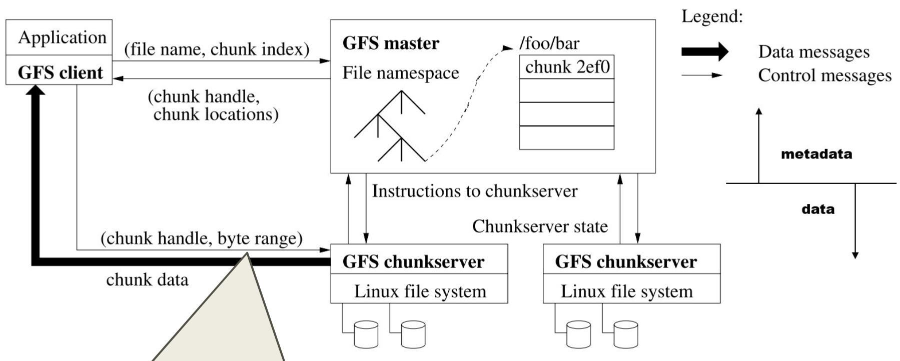


A Simple Read Example (3)

Cache information
using filename & chunk index as key



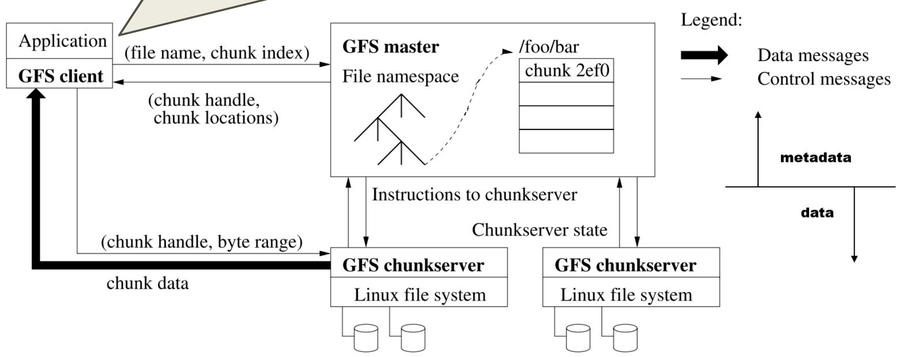
A Simple Read Example (4)



Request data from nearest chunkserver

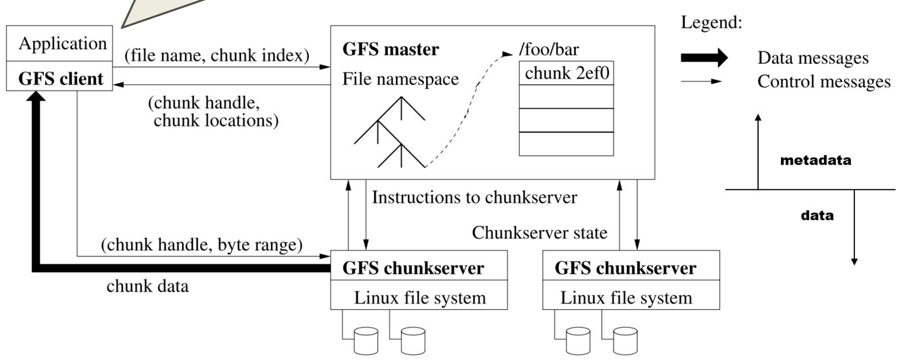
A Simple Read Example (5)

Further reads of the same chunk require no more client-master interaction until the cached information expires or the file is reopened.



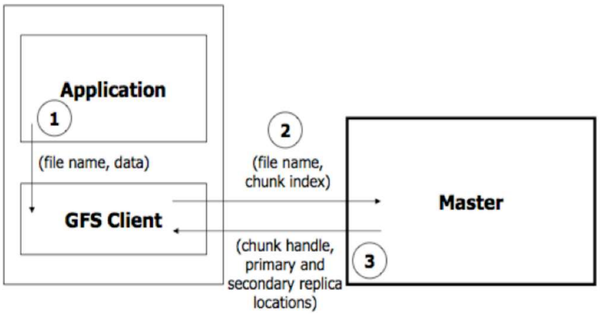
A Simple Read Example (6)

Typically asks for multiple chunks in the same request



Write Operation (1)

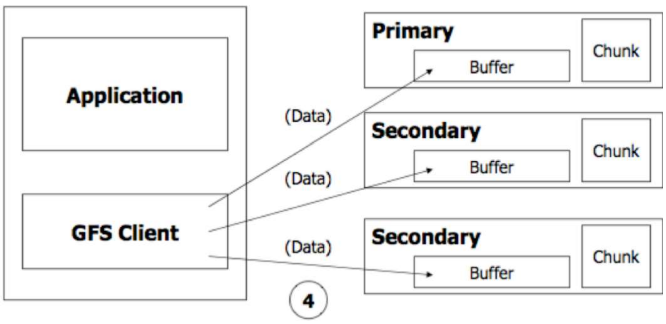
1. Application originates the request.
2. GFS client translates request and sends it to master.
3. Master responds with chunk handle and replica locations.



Write Operation (2)

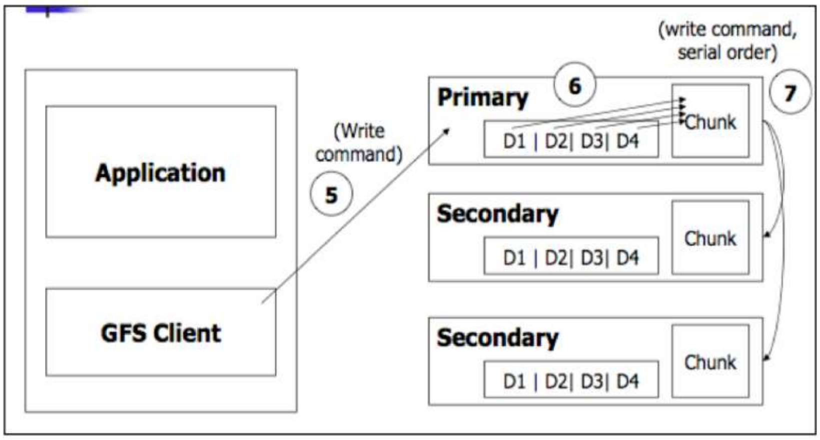
4. GFS Client pushes write data to all locations.

Data is stored in chunkserver's internal buffers.



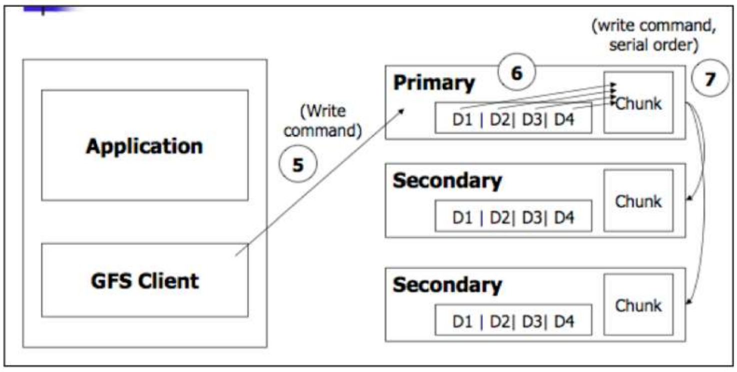
Write Operation (3)

5. Client sends write command to primary.



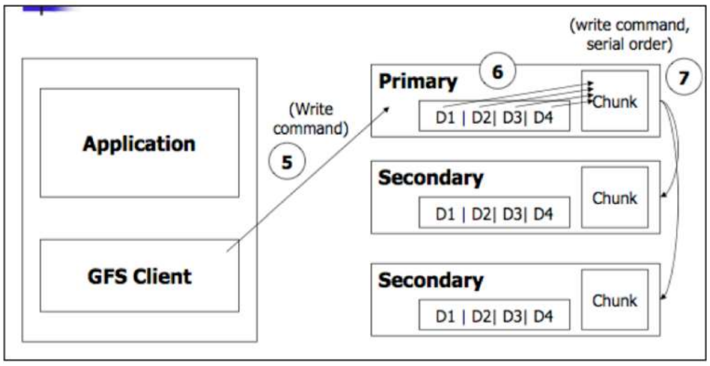
Write Operation (4)

6. Primary determines **serial order** for data instances in its buffer and writes the instances in that order to the chunk.



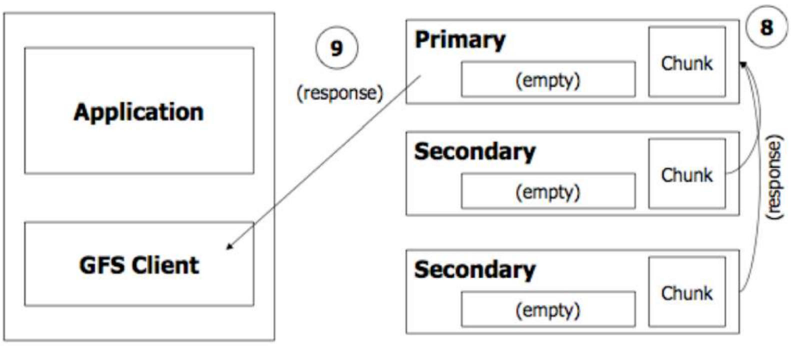
Write Operation (5)

7. Primary sends the serial order to the secondaries and tells them to perform the write in the same order. (replica consistency)



Write Operation (6)

8. Secondaries respond back to primary.
9. Primary responds back to the client.



Atomic Record Append Operation

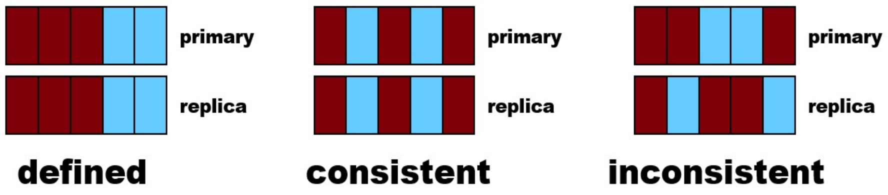
- Performed atomically (as a single byte sequence)
- At-least-once semantics
- Append offset is chosen by GFS and returned to client
- Same as write, extension to step 7:
  - If record fits in current chunk: write record and tell replicas the offset
  - If record exceeds chunk: pad the chunk, reply to client to use next chunk

File Deletion

- When client deletes file:
  - Master records deletion in its log
  - File renamed to hidden name including deletion timestamp
- Master scans file namespace in background:
  - Removes files with such names if deleted for longer than 3 days (configurable)
  - In-memory metadata erased
- Master scans chunk namespace in background:
  - Removes unreferenced chunks from chunkservers

Relaxed Consistency Model

- Consistent = all replicas have the same value
- Defined = consistent, all clients will always see what the mutation writes in its entirety (all replicas process chunk-mutation requests in the same order)
- Undefined: consistent + but it may not reflect what any one mutation has written
- Inconsistent: clients see different data at different times



Relaxed Consistency Model

- **Write**
  - Concurrent writes may be consistent but undefined
  - Write operations that are large or cross chunk boundaries are subdivided by client into individual writes
  - Concurrent writes may become interleaved

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Relaxed Consistency Model

- **Record Append**
  - Atomically, at-least-once semantics
  - Client retries failed operation
  - After successful retry, replicas are defined in region of append but may have intervening undefined regions

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Relaxed Consistency Model

- **Application safeguards**
  - Application safeguards (e.g., self-validating, self-identifying records)
  - Insert checksums in record headers to detect fragments
  - Insert sequence numbers to detect duplicates

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

Fault Tolerance (1)

- **High availability**
  - Fast recovery
    - Master and chunkservers are designed to restore their states and start in seconds
    - Do not distinguish between normal & abnormal termination
  - Chunk replication
    - 3 replicas (default)
  - Master replication
    - Master log & checkpoints replication
    - Shadow masters provide read-only access when the primary master is down

CSIE59830/CSIEM0410/AIIA50050 Big Data Systems   Big Data Storage I – DFS & GFS 49

Fault Tolerance (2)

- **Data integrity**
  - Checksum every 64KB block in each chunk
  - Verified at read & write times
  - Background scans for rarely used data
- **Limitations**
  - Security
    - Trusted environment, trusted users
    - That doesn't stop users from interfering with each other…
  - Does not mask all forms of data corruption: requires application-level checksum

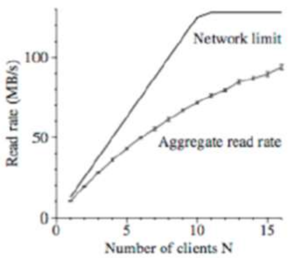
CSIE59830/CSIEM0410/AIIA50050 Big Data Systems   Big Data Storage I – DFS & GFS 50

Performance Test

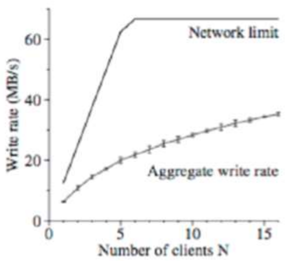
- **Cluster setup:**
  - 1 master
  - 16 chunkservers
  - 16 clients
- Server machines connected to central switch by 100 Mbps Ethernet
- Switches connected with 1 Gbps link

Performance Test

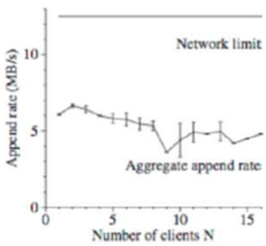
Reads



Write



Record Append



- 1 client:
  - 10 MB/s, 80% limit
- 16 clients:
  - 6 MB/s, 75% limit

- 1 client:
  - 6.3 MB/s, 50% limit
- 16 clients:
  - 35 MB/s, 50% limit
  - 2.2 MB/s per client

- 1 client:
  - 6 MB/s
- 16 clients:
  - 4.8 MB/s per client

GFS Summary

- Success: used actively by Google to support search service and other applications
  - Availability and recoverability on cheap hardware
  - High throughput by decoupling control and data
  - Supports massive data sets and concurrent appends
- Semantics not transparent to apps
  - Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics)
- Performance not good for all apps
  - Assumes read-once, write-once workload (no client caching!)

GFS Conclusions

- Many GFS clusters
  - Hundreds/thousands of storage nodes each
  - Managing petabytes of data
- GFS is under BigTable, and many other services.
- GFS demonstrates how to support large-scale processing workloads on commodity hardware
  - design to tolerate frequent component failures
  - optimize for huge files that are mostly appended and read
  - feel free to relax and extend FS interface as required
  - go for simple solutions (e.g., single master)
- GFS has met Google's storage needs, therefore good enough for them.

Transition From GFS to Colossus File System(CFS)

- Typical cluster now:
  - 10s of thousands of machines
  - PB of distributed HDD
  - Optional multi-TB local SSD
  - 10 GB/s bisection bandwidth
- Can a 2003 design meet the new demand?

GFS Architectural Problems

- GFS master
  - One machine not large enough for large FS
  - Single bottleneck for metadata operations
  - Fault tolerant, not High Availability(HA) enough
- Performance
  - No guarantees of latency
  - Not predicable performance

GFS2 Design Goals

- Bigger! Faster! More predictable latency
- Enhanced storage scalability and improved availability
- A distributed metadata model for a more scalable and highly available metadata system
- GFS master replaced by **Colossus**
- GFS chunkserver replaced by **D**
- **Unlike GFS, we only have very limited info about Colossus!**

Colossus (GFS2 or CFS)

- Next-generation cluster-level file system
- Automatically sharded(split) metadata layer
- Data typically written using Reed-Solomon (block-based error-correcting codes)
- Client-driven replication, encoding and replication
- Metadata space has enabled availability analyses
- Why Reed-Solomon?
  - Cost. Especially w/ cross cluster replication.
  - Field data and simulations show improved MTTF
  - More flexible cost vs. availability choices

Colossus (GFS2)

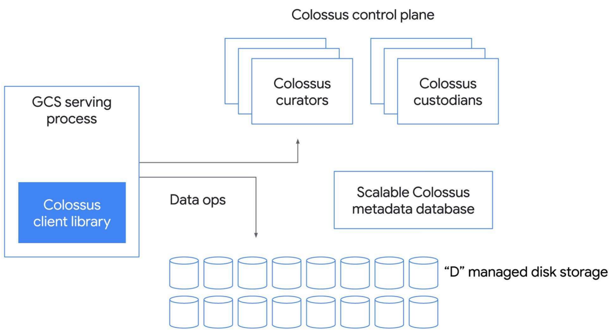
- A "multi-cell" approach, which put multiple GFS masters on top of a pool of chunkservers
- Also have Name Spaces, which are a static way of partitioning a namespace that people can use to hide all of this from the actual application a namespace file describes
- The distributed master allows you to grow file counts, in line with the number of machines you're willing to throw at it

Colossus (GFS2)

- The distributed master system is essentially a whole new design.
- Can aim for something on the order of 100 million files per master.
- You can also have hundreds of masters
- BigTable "as one of the major adaptations made along the way to help keep GFS viable in the face of rapid and widespread change."

Colossus Architecture

- Key components of Colossus:
  - Client library
  - Control plane (curators and custodians)
  - Metadata database
  - D file server
  - (more details to follow)



Colussus Client Library

- How an app or service interacts with Colossus.
- Many functionality: distributed file services, data operations, software RAID, …
- Applications can use a variety of encodings to fine-tune performance and cost trade-offs for different workloads.
- Clients talk directly to curators for control operations. (more later)

Colossus Control Plane

- Foundation of Colossus
- Provide scalable metadata service through many curators and custodians (more later)
- Scale horizontally to overcome the limits of GFS
- With Colossus, a single cluster is scalable to exabytes of storage and tens of thousands of machines.

Colossus Metadata DB

- Provide highly scalable metadata service
- Curators store file system metadata in Google BigTable (high-performance NoSQL database).
- Scale over 100x over the largest GFS clusters
- Custodians provide storage management through D file servers.
- Flexible design also improves availability.

Colossus Custodians

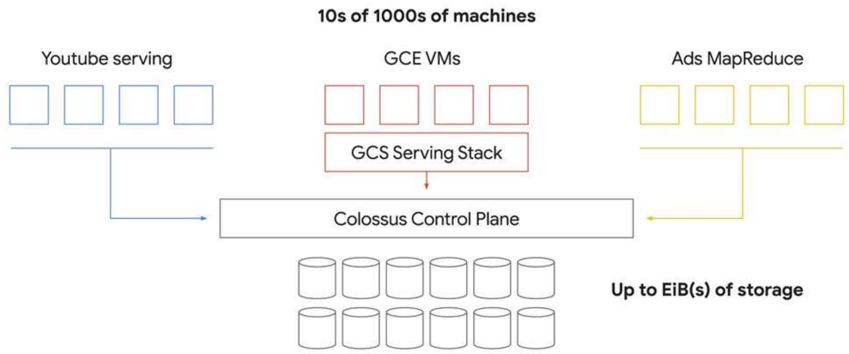
- The background storage managers.
- Play a key role in maintaining the durability and availability of data.
- Handling many tasks like disk space balancing and RAID reconstruction.
- Improve overall efficiency over GFS

D File Servers

- D file servers replace the chunkservers of GFS.
- D servers manage the network attached disk storage.
- Minimize the hops for data on the network by allowing data to flow directly between clients and the D file servers.

Typical Cluster with Colossus

- A single cluster with Colossus is scalable to exabytes of storage and tens of thousands of machines.



Colossus Advantages

- Cloud storage with Colossus can support a wide range of use cases.
- The sharded storage pool managed by Colossus provides the illusion that each instance of application has its own isolated file system.
- Disaggregation of resources drives more efficient use of valuable resources and lowers costs across all workloads.
- Support both batch analytic and low latency workloads.

Colossus Advantages

- Colossus abstracts away physical hardware complexity to simplify storage-intensive applications.
- Colossus provides a range of service tiers. Applications use them by specifying I/O, availability, and durability requirements, and then provisioning resources (bytes and I/O) as abstract, undifferentiated units.
- Colossus steers IO around HW failures and does fast background recovery to provide highly durable and available storage.

Colossus Advantages

- Colossus uses a mix of flash and disk storage to meet a wide variety of access patterns and frequencies.
- With the right mix, Colossus can maximize storage efficiency and avoid wasteful overprovisioning.
- Colossus uses intelligent disk management to get as much value as possible from available disk IOPS (I/O operations per second, pronounced eye-ops).
- Newly written data (i.e. hotter data) is evenly distributed across all the drives.
- Data is rebalanced and moved to larger capacity drives as it ages and becomes colder.

Colossus Impact

- **Colossus** has been extremely useful for optimizing storage efficiency
- Metadata scaling enables declustering of resources
- Ability to combine disks of various sizes and workloads of varying types is very powerful
- Looking forward, I/O cost trends will require both applications and storage systems to evolve