CS535 BIG DATA

PART 1. BATCH COMPUTING MODELS FOR BIG DATA ANALYTICS

1. DISTRIBUTED MODEL FOR SCALABLE BATCH COMPUTING – DISTRIBUTED FILE SYSTEMS

Sangmi Lee Pallickara
Computer Science, Colorado State University
http://www.cs.colostate.edu/~cs535

FAQs

- Questions about PA1
  - Send an email to cs535@cs.colostate.edu
- FAQ for PA1 page is available at:
  - http://www.cs.colostate.edu/~cs535/FAQ_PA1.html
- Term Project
  - Google computing cluster credit is available

Objectives

- Distributed File System
  - GFS, GFS2 (Colossus)

Distributed File Systems

Google File System

This material is built based on

- Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system. Proceedings of SOSP 2003: 29-43
- Andrew Fikes, Storage Architecture and Challenges, Faculty Summit, 2010
  - http://static.googleusercontent.com/external_content/untrusted_dlcp/research.google.reverse-proxy.org/en/us/university/relations/facultysummit2010/storage_architecture_and_challenges.pdf
- Jeff Dean's SOCC keynote, Building Large-Scale Internet Services
  - http://static.googleusercontent.com/media/research.google.com/en//people/jeff/SOCC2010-keynote-slides.pdf
- http://sysmagazine.com/posts/206986/
- Erasure Coding: Backblaze Open sources Reed-Solomon
  - https://www.backblaze.com/blog/reed-solomon/
- An introduction to Reed-Solomon codes
  - http://www.cs.cmu.edu/~guyb/realworld/reedsolomon/reed_solomon_codes.html

Storage Software: GFS

- Google's first cluster-level file system (2003)
  - Designed for batch applications with large files Single master for metadata and chunk management Chunks are typically replicated 3x for reliability
- Lessons
  - Scaled to approximately 50M files, and 10PB
  - Large files increased application complexity
  - Not appropriate for latency sensitive applications
  - Scaling limits added management overhead

Storage Software: Colossus (GFS2)

- Next-generation cluster-level file system
- Automatically sharded metadata layer
  - Data typically written using Reed-Solomon (1.5x)
  - Client-driven replication, encoding and replication
  - Metadata space has enabled availability
- Why Reed-Solomon?
  - Cost. Especially with cross cluster replication
  - More flexible cost vs. availability choice

Storage Landscape

- Early Google:
  - US-centric traffic
  - Batch, latency-insensitive indexing processes
  - Document "snippets" serving (single seek)
- Current day:
  - World-wide traffic
  - Continuous crawl and indexing processes (Caffeine)
  - Seek-heavy, latency-sensitive apps (Gmail)
  - Person-to-person, person-to-group sharing (Docs)

Storage Landscape: Flash (SSDs)

- Important future directions:
  - More workloads that are increasingly seek heavy
  - 50-150x less expensive than disk per random read
  - Best usage is still being explored
- Concerns:
  - Availability of devices
  - 17-32x more expensive per GB than disk
  - Endurance not yet proven in the field

Google File System

Demand pulls in GFS (1/2)

- Files are huge by traditional standards
- File mutations predominantly through **appends**
  - Not overwrites
- Component failures are the **norm**
- Applications and File system API designed in **lock-step**

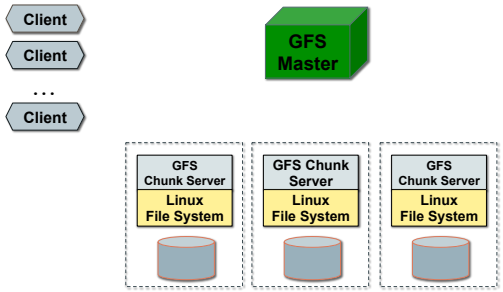
Demand pulls in GFS n (2/2)

- Hundreds of producers will **concurrently append** to a file
  - Many-way merging
- High **sustained bandwidth** is more important
  - than low latency

The file system interface

- Does not implement standard APIs such as POSIX
- Supports `create, delete, open, close, read` and `write`
- **`snapshot`**
  - Create a fast copy of file and directory tree
- **`record append`**
  - Multiple files can concurrently append records to the same file
    - Without additional locking

Architecture of GFS



Chunks

- Obvious reason
  - The file is too big
- **Set the stage** for computations that operate on this data
  - Parallel I/O
  - I/O seek times are 14 x 10^6 slower than CPU access times

Chunk size

- This is fixed at **64 MB**
  - Much larger than typical FS block sizes (512 bytes)
- **Lazy space allocation (delayed space allocation)**
  - Stored as plain Linux file
  - Physical allocation of disk space is delayed as long as possible
    - Until data at the size of the chunk size
  - Extended only as needed
  - Avoiding **internal fragmentation**

Chunk size: But why this big?-Advantage

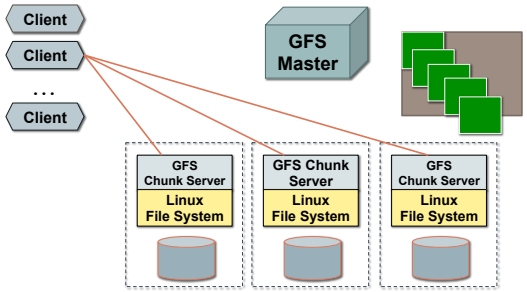
- **Reduces client interaction** with the master
  - Can cache info for a multi-TB working set
- Reduce network **overhead**
  - With a large chunk, client performs more operations
  - Persistent connections
- Reduce **size of metadata** stored in the master
  - 64 bytes of metadata per 64 MB chunk

Large chunk size: Disadvantage

- Small files (with small number of chunks)
  - May become hot spots
  - e.g. popular executable files
- Solution
  - Assigning a higher replication factor

Master Operations in GFS

Architecture of GFS



Master operations

- Single master
- Manage system **metadata**
- **Leasing** of chunks
- **Garbage collection** of orphaned chunks
- Chunk **migrations**

Why have a single Master?

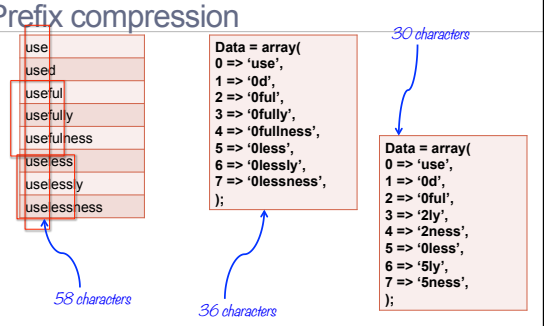
- Vastly **simplifies** design
- Easy to use global knowledge to **reason about**
  - Chunk placements
  - Replication decisions

ALL system metadata is managed by the Master and stored in Main Memory

① File and chunk namespaces
② Mapping from files to chunks
③ Location of chunks

Logs mutations into a permanent log

Prefix compression



Why keep the entire metadata in memory?

- **Speed**
- Master can **scan** its **state** in the background
  - Implement chunk garbage collection
  - Re-replicate if there are failures
  - Chunk migration to balance load and space
- **Add** extra memory to increase file system size

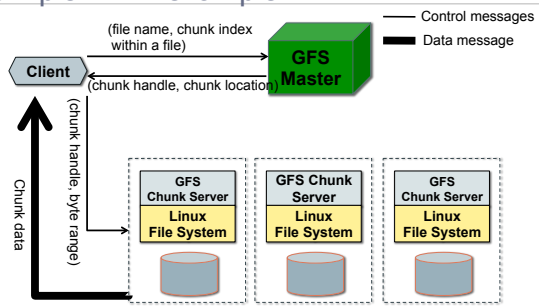
Size of the file system with 1 TB of RAM: Assume file sizes are exact multiples of chunk sizes

- Number of entries = $2^{40}/2^{6}$
- MAXIMUM SIZE of the file system
  = Number of entries x Chunk size
  $= \frac{2^{40}}{2^{6}} \times 2^{6} \times 2^{20}$
  $= 2^{60} = 1\ EB$

Tracking the chunk servers

- Master **does not** keep a persistent copy of the location of chunk servers
- List maintained via **heart-beats**
  - Allows list to be in *sync* with reality despite failures
  - Chunk server has final word on chunks it holds

Simple `read` example



Caching at the client/chunk servers

- Clients **do not cache** file data
  - At client the working set may be *too large*
  - Simplify client; eliminate *cache-coherency* problems
- Chunk servers **do not cache** file data either
  - Chunks are stored as local files
  - Linux's buffer cache *already keeps* frequently accessed data in memory

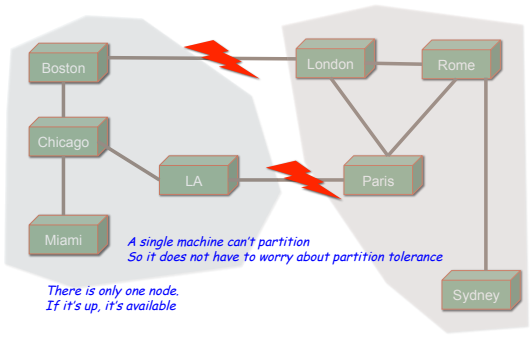
Consistency in GFS:

Relaxed Consistency

Relaxed consistency

- The CAP theorem
  - Eric Brewer, 2000
  - Seth Gilbert and Nancy Lynch, 2002
    - Formal proof
- Given the three properties of **Consistency, Availability**, and **Partition tolerance**, you can only get **two**
- Availability
  - If you can talk to a node in the cluster, it can read and write data.
- Partition tolerance
  - The cluster can survive communication failures that separate the cluster into multiple partitions unable to communicate with each other

Two breaks in the communication lines



Eventually consistent

- At any time nodes may have replication inconsistencies
- If there are no more updates (or updates can be ordered), eventually all nodes will be updated to the same value

In GFS the state of file region after mutation depends on …

- **Type** of the mutation
- **Success/Failure** of the mutation
- Whether there were **concurrent** mutations

Mutations in GFS

- **Mutation** changes the content or metadata of a chunk
  - Write, and append
- Each mutation is performed at **all** chunk replicas
- **Append** is done based on a *record*. (1/4 of a chunk)
  - Offset is specified by **GFS**

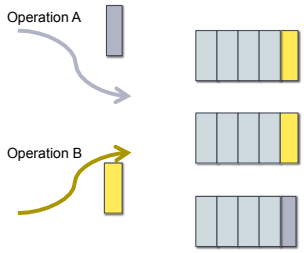
Data mutation

- Write
  - Data to be written at an application-specified file offset
- Record appends (in GFS)
  - Data ("record") to be appended atomically at least once even in the presence of concurrent mutations
  - GFS chooses the offset
  - No need for a distributed lock manger

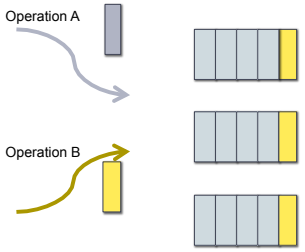
GFS has a relaxed consistency model

- **Consistent**: See the same data
  - On all replicas
- **Defined**: If it is consistent AND
  - Clients **see mutation** writes in its **entirety**

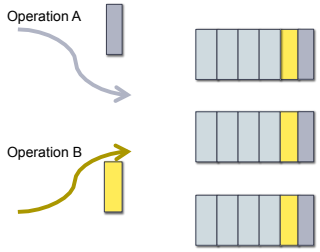
Inconsistent and undefined



Consistent but undefined



Defined



File state region after a mutation

	Write	Record Append
Serial success	**Defined**	**defined** interspersed with *inconsistent*
Concurrent success	Consistent but *undefined*	
Failure	Inconsistent	

GFS guarantees the mutated file to be defined and to contain data written by the last mutation

- Applying mutations to a chunk in the same order on all its replicas
- Using chunk version numbers to detect any replica that has become stale
  - If the chunkserver was down

Clients cache chunk locations…

- What if the chunk location points to a stale replica before that information is refreshed?
- The window is limited by:
  - Cache entry's timeout
  - Next open to the file
    - It will purge all chunk information for that file from the cache
- Append-only
  - Stale replica usually returns a premature end of chunk
    - rather than outdated data
  - When a reader retries and contact master
    - It will get current chuck locations

Implications for applications

- Rely on **appends** instead of overwrites
- Checkpoint
- Write **records** that are
  - Self-validating
  - Self-identifying

Managing Mutations:

Handling writes and appends to a file

GFS uses leases to maintain consistent mutation order across replicas

- Master grants **lease** to one of the replicas
  - **Primary**
- Primary picks **serial-order**
  - For all mutations to the chunk
  - Other replicas *follow* this order
    - When applying mutations

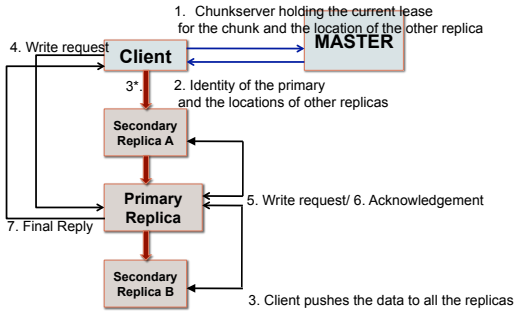
Lease mechanism designed to minimize communications with the master

- Lease has initial *timeout* of 60 seconds
- As long as chunk is being mutated
  - Primary can request and receive **extensions**
- Extension requests/grants piggybacked over heart-beat messages

Revocation and transfer of leases

- Master may **revoke** a lease before it expires
- If communications lost with primary
  - Master can safely give lease to another replica
    - **Only After** the lease period for old primary *elapses*

How a `write` is actually performed



Client pushes data to all the replicas (I)

- Each chunk server stores data in an **LRU buffer** until
  - Data is used
  - Aged out

Client pushes data to all the replicas (II)

- When chunk servers acknowledge receipt of data
  - Client sends a *write request to primary*
- Primary assigns *consecutive serial numbers* to mutations
  - Forwards to replicas

Data flow is decoupled from the control flow to utilize network efficiently

- Utilize each machine's network bandwidth
- Avoid network bottlenecks
- Avoid high-latency links
- **Leverage** network topology
  - Estimate distances from IP addresses
- Pipeline the data transfer
  - Once a chunkserver receives some data, it starts forwarding immediately.
  - For transferring B bytes to R replicas
    - Ideal elapsed time will be $\approx B/T+RL$ where:
      - T is the network throughput
      - L is latency to transfer bytes between two machines