GFS

Google Stack

- GFS: large-scale storage for bulk data
- Chubby: Paxos storage for coordination
- BigTable: semi-structured data storage
- MapReduce: big data computation on key-value pairs
- MegaStore, Spanner: transactional storage with geo-replication

GFS

- Needed: distributed file system for storing results of web crawl and search index
- Why not use NFS? (That is, some existing distributed file system.)

GFS

- Needed: distributed file system for storing results of web crawl and search index
- Why not use NFS?
  - very different workload characteristics!
  - design GFS for Google apps, Google apps for GFS
- Requirements:
  - Fault tolerance, availability, throughput, scale
  - Concurrent streaming reads and writes

GFS Workload

- Producer/consumer
  - Hundreds of web crawling clients
  - Periodic batch analytic jobs like MapReduce
  - Throughput, not latency
- Big data sets (for the time):
  - 1000 servers, 300 TB of data stored
- Later: BigTable tablet log and SSTables
- Even later: Workload now?

GFS Workload

- Few million 100MB+ files
  - Many are huge
- Reads:
  - Mostly large streaming reads
  - Some sorted random reads
- Writes:
  - Most files written once, never updated
  - Most writes are appends, e.g., concurrent workers

GFS Interface

- app-level library
  - not a kernel file system
  - Not a POSIX file system
- create, delete, open, close, read, write, append
  - Metadata operations are linearizable
  - File data eventually consistent (stale reads)
- Inexpensive file, directory snapshots

Life without random writes

- Results of a previous crawl:

www.page1.com -> www.my.blogspot.com

www.page2.com -> www.my.blogspot.com

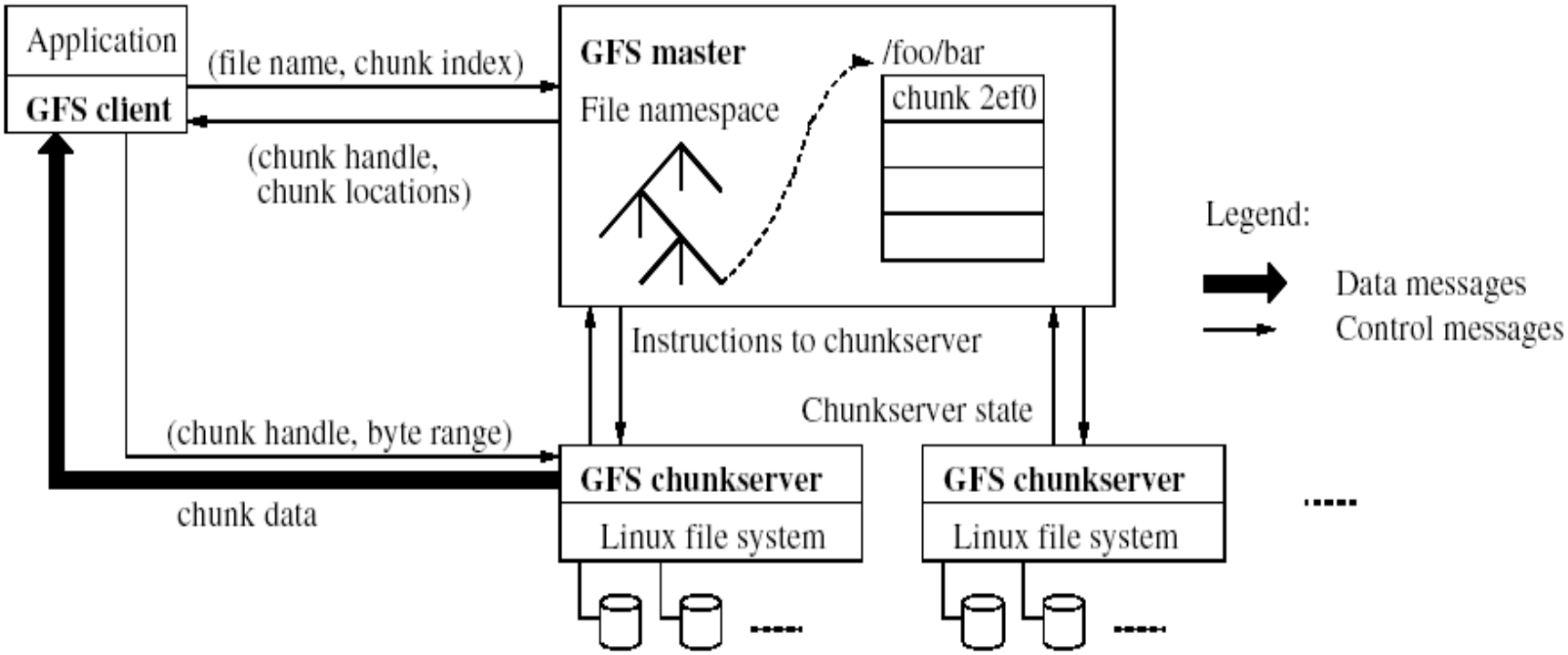
- New results: page2 no longer has the link, but there is a new page, page3:

www.page1.com -> www.my.blogspot.com

www.page3.com -> www.my.blogspot.com

- Option: delete old record (page2); insert new record (page3)
  - requires locking, hard to implement
- GFS: append new records to the file atomically

GFS Architecture



- each file stored as 64MB chunks
- each chunk on 3+ chunkservers
- single master stores metadata

“Single” Master Architecture

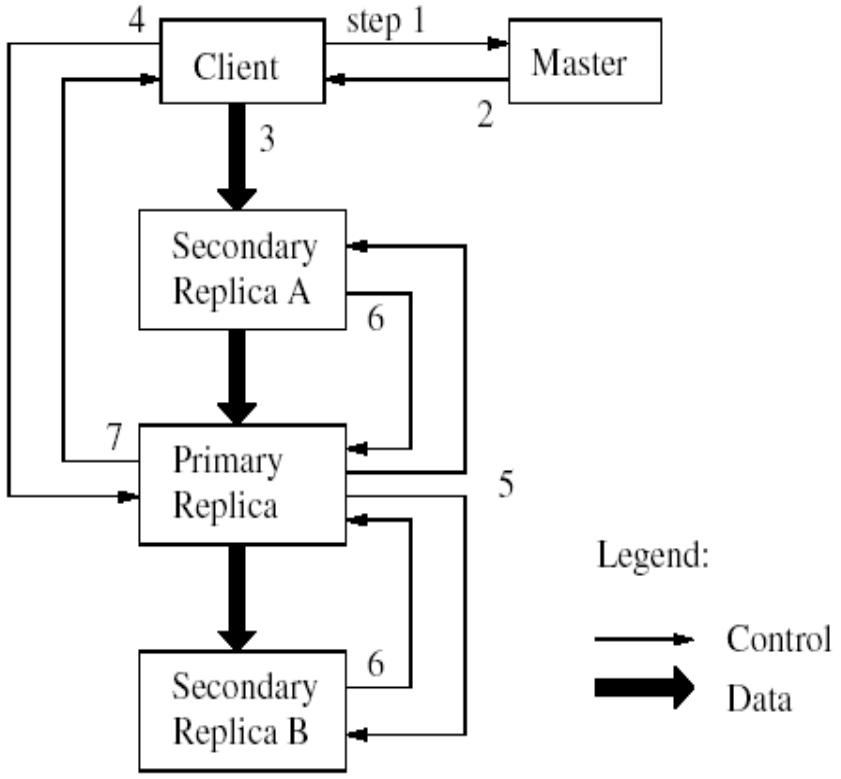
- Master stores metadata:
  - File name space, file name -> chunk list
  - chunk ID -> list of chunkservers holding it
  - Metadata stored in memory (~64B/chunk)
- Master does not store file contents
  - All requests for file data go directly to chunkservers
- Hot standby replication using shadow masters
  - Fast recovery
- All metadata operations are linearizable

Master Fault Tolerance

- One master, set of replicas
  - Master chosen by Chubby
- Master logs (some) metadata operations
  - Changes to namespace, ACLs, file -> chunk IDs
  - Not chunk ID -> chunkserver; why not?
- Replicate operations at shadow masters and log to disk, then execute op
- Periodic checkpoint of master in-memory data
  - Allows master to truncate log, speed recovery
  - Checkpoint proceeds in parallel with new ops

Handling Write Operations

- Mutation is write or append
- Goal: minimize master involvement
- Lease mechanism
  - Master picks one replica as primary; gives it a lease
  - Primary defines a serial order of mutations
- Data flow decoupled from control flow



Write Operations

- Application originates write request
- GFS client translates request from (fname, data) --> (fname, chunk-index) sends it to master
- Master responds with chunk handle and (primary+secondary) replica locations
- Client pushes write data to all locations; data is stored in chunkservers' internal buffers
- Client sends write command to primary

Write Operations (contd.)

- Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk
- Primary sends serial order to the secondaries and tells them to perform the write
- Secondaries respond to the primary
- Primary responds back to client
- If write fails at one of the chunkservers, client is informed and retries the write/append, but another client may read stale data from chunkserver

At Least Once Append

- If failure at primary or any replica, retry append (at new offset)
  - Append will eventually succeed!
  - May succeed multiple times!
- App client library responsible for
  - Detecting corrupted copies of appended records
  - Ignoring extra copies (during streaming reads)
- Why not append exactly once?

Caching

- GFS caches file metadata on clients
  - Ex: chunk ID -> chunkservers
  - Used as a hint: invalidate on use
  - TB file => 16K chunks
- GFS does not cache file data on clients

Garbage Collection

- File delete => rename to a hidden file
- Background task at master
  - Deletes hidden files
  - Deletes any unreferenced chunks
- Simpler than foreground deletion
  - What if chunk server is partitioned during delete?
- Need background GC anyway
  - Stale/orphan chunks

Data Corruption

- Files stored on Linux, and Linux has bugs
  - Sometimes silent corruptions
- Files stored on disk, and disks are not fail-stop
  - Stored blocks can become corrupted over time
  - Ex: writes to sectors on nearby tracks
  - Rare events become common at scale
- Chunkservers maintain per-chunk CRCs (64KB)
  - Local log of CRC updates
  - Verify CRCs before returning read data
  - Periodic revalidation to detect background failures

Discussion

- Is this a good design?
- Can we improve on it?
- Will it scale to even larger workloads?

~15 years later

- Scale is much bigger:
  - now 10K servers instead of 1K
  - now 100 PB instead of 100 TB
- Bigger workload change: updates to small files!
- Around 2010: incremental updates of the Google search index

GFS -> Colossus

- GFS scaled to ~50 million files, ~10 PB
- Developers had to organize their apps around large append-only files (see BigTable)
- Latency-sensitive applications suffered
- GFS eventually replaced with a new design, Colossus

Metadata scalability

- Main scalability limit: single master stores all metadata
- HDFS has same problem (single NameNode)
- Approach: partition the metadata among multiple masters
- New system supports ~100M files per master and smaller chunk sizes: 1MB instead of 64MB

Reducing Storage Overhead

- Replication: 3x storage to handle two copies
- Erasure coding more flexible: m pieces, n check pieces
  - e.g., RAID-5: 2 disks, 1 parity disk (XOR of other two) => 1 failure w/ only 1.5 storage
- Sub-chunk writes more expensive (read-modify-write)
- After a failure: get all the other pieces, generate missing one

Erasure Coding

- 3-way replication:
  3x overhead, 2 failures tolerated, easy recovery
- Google Colossus: (6,3) Reed-Solomon code
  1.5x overhead, 3 failures
- Facebook HDFS: (10,4) Reed-Solomon
  1.4x overhead, 4 failures, expensive recovery
- Azure: more advanced code (12, 4)
  1.33x, 4 failures, same recovery cost as Colossus