The Google File System
SOSP 2003

Alan Sussman

CMSC 818S

April 12, 2007

Notes

- Project interim report due Wed., 4/18
- Final dates?
- Next sets of papers?
  - Timur - Bloom filters

Overview

- Lots of interesting things here
- Usual distributed file system concerns
  - performance, scalability, reliability, availability
- But note the close ties to their applications
  - web crawling, web search, and more (Google Earth, Google Maps, Froogle, …)
  - distributed file system design is driven by performance needs and special characteristics of the apps

Assumptions

- Application workload and technology environment
  - component failures are normal occurrences, so need to monitor, detect errors, tolerate faults, and automatically recover
    - basically large PC clusters, each with lots of disk
    - app bugs, OS bugs, human errors, and hardware component failures (disks, memory, connectors, networking, power supplies)
  - files are very large – multi-GB common, each containing many app objects (e.g., web documents)
    - and total dataset sizes many TB, with billions of objects
    - so standard file system parameter choices may not work – e.g., block sizes

Assumptions (cont.)

- most files mutate by **appending** new data, not overwriting
  - and random writes are rare
  - once written, files only read, and often only sequentially
  - so optimize performance and guarantee atomicity for append writes, and don't bother with client caching (no temporal locality)
- co-design apps and file system API
  - relaxed consistency model
  - atomic append operation to allow multiple concurrent appends by different clients w/o synchronization
  - want high sustained bandwidth, low latency not as important, for bulk data processing

Interface (API)

- File system interface
  - but not standard POSIX API
  - most usual file operations – *create*, *delete*, *open*, *close*, *read*, *write*
  - *snapshot* copies a file or directory tree cheaply (using copy-on-write)
  - *record append* allows multiple concurrent appends to the same file, with guaranteed atomicity of each append
    - useful for multi-way merging of results, and producer-consumer queries (multiple producers, 1 or more consumers)

Architecture

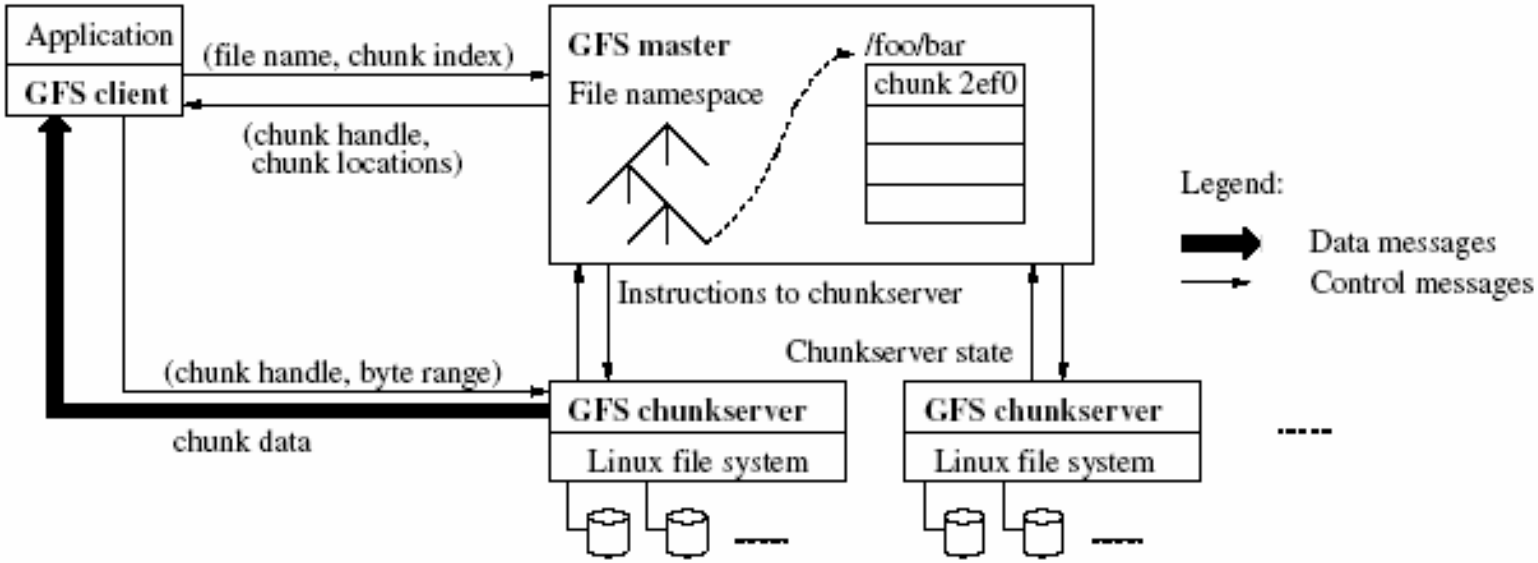

Figure 1: GFS Architecture

Architecture (cont.)

- Files divided into fixed-size *chunks*
  - and each chunk gets an immutable, globally unique 64 bit *chunk handle* from the master at chunk creation time
- One *master*
  - maintains all file system metadata
    - namespace, access control info, mapping from files to chunks, current chunk locations
    - chunk lease management, garbage collection of orphaned chunks, chunk migration between chunkservers
    - talks to each chunkserver periodically in *heartbeat* messages, to send instructions and collect chunkserver state
- Multiple *chunkservers*
  - store chunks on local disks as Linux files
  - read/write data specified by chunk handle and a byte range (or append, for write)
  - chunk replicated on multiple chunkservers for reliability (3 by default)
  - no caching of file data, since Linux file system buffer cache does it
- Multiple *clients*
  - code for client API linked into apps, talks to master for metadata operations and to chunkservers to read/write data for app
  - no client caching of file data, since apps either stream data or have too large working sets for a cache to help

The master

- One master (not really, *shadow masters* later)
  - to use global info for chunk placement and replication
  - only accessed for metadata - which chunkserver(s) to access
    - this is cached for a limited time in clients
- Chunk size is 64MB
  - chunk replica is a Linux file on a chunkserver
    - extended only as needed to minimize internal fragmentation
  - large size is good because
    - client requires fewer metadata accesses to master for chunk location info (and location info can be cached in client)
    - reduces network overhead since client likely to perform multiple operations on a chunk, so can keep persistent TCP connection to chunkserver
    - reduces size of metadata on master, so can keep metadata in memory
  - disadvantage is that small files can cause hot spots for many clients accessing the same small file
    - solution is more replication of small files, and re-working apps

Master metadata

- File and chunk namespaces, mappings from files to chunks, locations of each chunk's replicas
  - all stored in memory, so fast access
- Namespaces and mappings must be *persistent*
  - done by logging mutations to an *operation log* stored on master's disk and replicated on other machines for reliability
- Chunk location information not stored persistently
  - acquired from chunkservers at master startup or when a chunkserver joins the cluster
  - so the chunkserver is completely responsible for its own chunk data
- Periodically scans entire state in background
  - for chunk garbage collection, replication for data in failed chunkservers, chunk migration to balance load and disk space use
- Size of metadata not a problem
  - less than 64 bytes per 64MB chunk, and most chunks are full
  - and same for file namespace
- Operation log contains record of critical metadata changes
  - also provides a time line to define the order of concurrent operations (from multiple clients)
  - stored reliably, and changes not made visible until changes are persistent (like a database *transaction log*)
  - master recovers by replaying log from last checkpoint, which is done periodically to keep the log small
  - checkpoint creation is done in the background, and when completed written to disk locally and remotely
  - recovery only needs a checkpoint file and subsequent log files

Consistency

- File namespace mutations (e.g., file creation) are atomic, and handled in master with locking – operation log defines a total ordering of the mutations
- Let's not get into the details, but the model essentially provides well-defined, relaxed consistency guarantees that work well for the target applications
- And failure of a chunkserver, after detection from lost heartbeats and/or checksumming, is handled by making more replicas
  - so a chunk is lost only if all replicas lost before further replication
- Apps deal with the relaxed consistency model by using appends instead of overwrites, checkpointing, and writing self-validating, self-identifying records (so can detect inconsistencies if they occur)

Write Control and Data Flow

Mutation performed at all chunk replicas, with leases to maintain consistent order across replicas

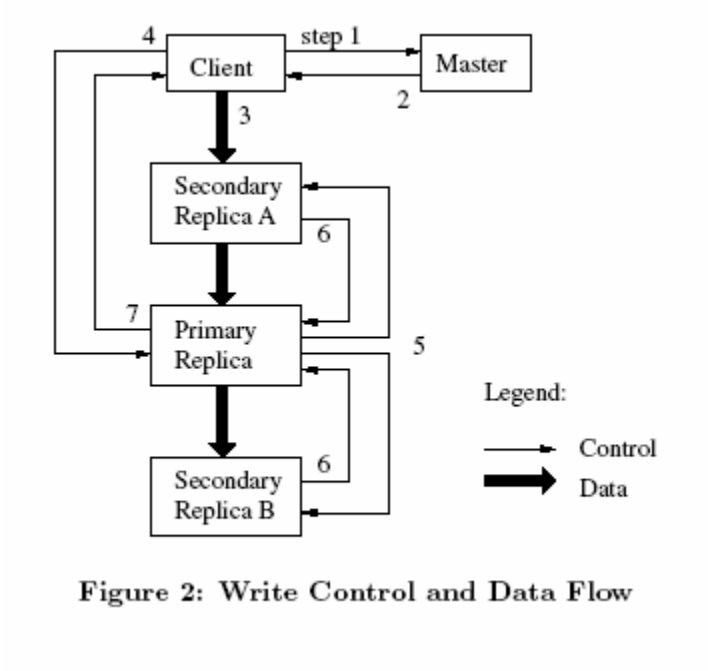

Figure 2: Write Control and Data Flow

1. Client asks master for primary replica
2. Master replies with primary and secondaries, cached in client
3. Client pushes data to all replicas, in any order, stored in buffer cache
4. Once all replicas ack, client sends write request to primary, which assigns serial number and applies mutation
5. Primary forwards write request to all secondaries, secondaries apply mutations in serial number order
6. Secondaries reply to primary on completion of mutation
7. Primary replies to client, reporting errors at any replicas (had to succeed at least at primary), so client can retry from step 3, or eventually from step 1

Atomic record appends

- Client specifies data, GFS guarantees record appended to file at least once atomically, at offset chosen by GFS and returned to client
  - works for multiple concurrent client appends
  - works like previous slide, with client pushing data to all replicas of last chunk of file
    - primary checks to see if data causes chunk to be greater than 64MB,
    - if so pads chunk to 64MB and tells secondaries to do same, then tells client to retry on next chunk
    - if data fits, primary appends data to its replica, tells secondaries to write at the offset it did, and replies success to client

Snapshots

- Make a copy of a file, or a directory tree, almost instantaneously, with little interruption of ongoing mutations
  - good for checkpoints of current state before large changes (to undo), or for branch copies of large datasets
  - uses standard copy-on-write techniques to only make a real copy when a chunk changes
  - optimized to only do local copies on chunkservers, by placing copied chunks on same chunkserver as original one (can be migrated later)

Master namespace management

- Use locks to allow multiple concurrent operations on part of the namespace
- Namespace is logically a lookup table mapping full pathnames to metadata (no inodes/directories)
- Each node in namespace tree has a read-write lock, and each master operation acquires locks before it runs
- For operation on **/d1/d2/…/dn/leaf** acquire read-write locks on **/d1**, **/d1/d2**, …, **/d1/d2/…/dn** and a read or write lock on **/d1/d2/…/dn/leaf**
- Locks acquired in order listed, to prevent deadlock (two-phase locking)
- This strategy correctly serializes concurrent operations in the same part of the tree
- Also allows concurrent mutations in the same directory

Replicas

- Placement done to ensure spread across multiple racks, not just multiple machines, in a cluster
  - at the expense of more network bandwidth between racks
- Replicas created at chunk creation, for re-replication and for rebalancing
  - goal is maximize availability, balance load across chunkservers, balance available disk space across chunkservers
  - master re-replicates when number of available replicas is less than what client requested, and done lazily in background with obvious priorities (to not lose last replica, to satisfy a client request directly)
  - lazy rebalancing also used to fill up a new chunkserver gradually

Garbage collection

- Storage for deleted files not reclaimed immediately, but lazily via garbage collection for files and chunks
- Master logs the delete, but just renames file to be a hidden name with the deletion timestamp
  - during master's regular scan of namespace, remove hidden files if older than 3 days, removing the in-memory metadata
  - so can undelete files until garbage collection
  - orphaned chunks (not reachable from any file) identified and their metadata erased during scan
  - chunkservers report subset of chunks they own in HeartBeat message, and master replies with identity of all chunks no longer in master's metadata, so chunkserver can delete its replicas of those chunks
- This mechanism is reliable under component failures, and is done in batches by the master, amortizing overhead
- Main disadvantage is when space is tight, or when lots of files are created and deleted rapidly
  - users get some control over replication and reclamation policy (e.g., don't replicate this temp file)

Master replication

- Master state replicated for reliability
  - operation logs and checkpoints replicated on multiple machines
  - mutation to state committed only after log record flushed to disk locally and on all master replicas
  - but one master process is in charge of all mutations and background activities that change system internal state
  - monitor process can restart the master from checkpoint and logs, obtained from some “shadow” master
  - even while primary master is down, shadow masters provide read-only access to file system
    - shadows update themselves from the operation log, and poll chunkservers at startup to locate chunk replicas and then monitor chunkserver status

Data integrity

- Chunkservers use checksumming to detect corruption of stored data
  - disks themselves do this too for sectors
  - chunks broken into 64KB blocks, each with a 32 bit checksum, stored in memory and stored persistently with logging, separate from user data
  - checksum verified on read for each block, and if error detected an error is returned to the requestor (client or another chunkserver)
    - master will create another replica of the failed chunk, then tell the chunkserver with the error to delete its chunk replica
- Checksumming is low overhead for reads, since they're stored in memory and are small compared to the data
- Write checksumming optimized for appends, since only need to compute new checksum for one partial block (maybe) and any new blocks appended
- Overwrites more expensive – read, checksum, write, compute new checksum

Performance measurements

- Overall, high aggregate throughputs for reading and writing, and seems to scale with size of cluster
  - no obvious bottlenecks

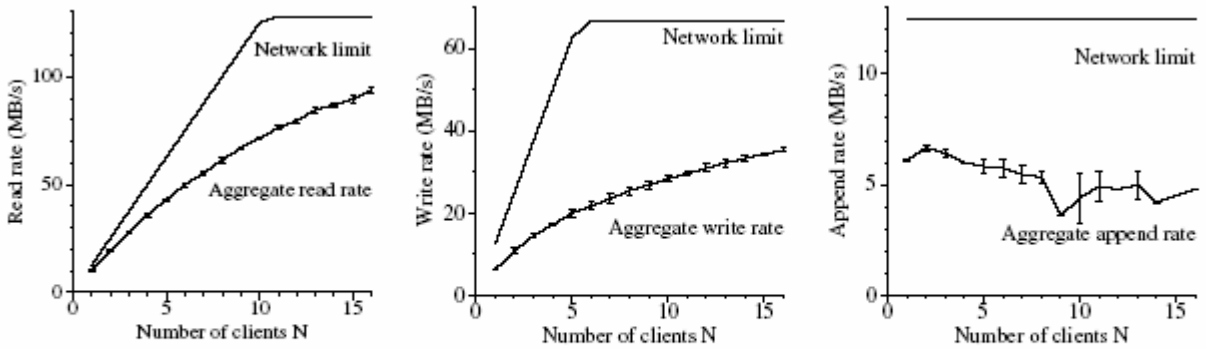

Figure 3: Aggregate Throughputs. Top curves show theoretical limits imposed by our network topology. Bottom curves show measured throughputs. They have error bars that show 95% confidence intervals, which are illegible in some cases because of low variance in measurements.