GFS: The Google File System

Brad Karp
UCL Computer Science

CS GZ03 / M030
3rd November, 2010

Motivating Application: Google

- Crawl the whole web
- Store it all on "one big disk"
- Process users' searches on "one big CPU"
- More storage, CPU required than one PC can offer
- Custom parallel supercomputer: expensive (so much so, not really available today)

Cluster of PCs as Supercomputer

- Lots of cheap PCs, each with disk and CPU
  - High aggregate storage capacity
  - Spread search processing across many CPUs
- How to share data among PCs?
- Ivy: shared virtual memory
  - Fine-grained, relatively strong consistency at load/ store level
  - Fault tolerance?
- NFS: share fs from one server, many clients
  - Goal: mimic original UNIX local fs semantics
  - Compromise: close-to-open consistency (performance)
  - Fault tolerance?

Cluster of PCs as Supercomputer

GFS: File system for sharing data on clusters, designed with Google's application workload specifically in mind

- Ivy: shared virtual memory
  - Fine-grained, relatively strong consistency at load/ store level
  - Fault tolerance?
- NFS: share fs from one server, many clients
  - Goal: mimic original UNIX local fs semantics
  - Compromise: close-to-open consistency (performance)
  - Fault tolerance?

Google Platform Characteristics

- 100s to 1000s of PCs in cluster
- Cheap, commodity parts in PCs
- Many modes of failure for each PC:
  - App bugs, OS bugs
  - Human error
  - Disk failure, memory failure, net failure, power supply failure
  - Connector failure
- Monitoring, fault tolerance, auto-recovery essential

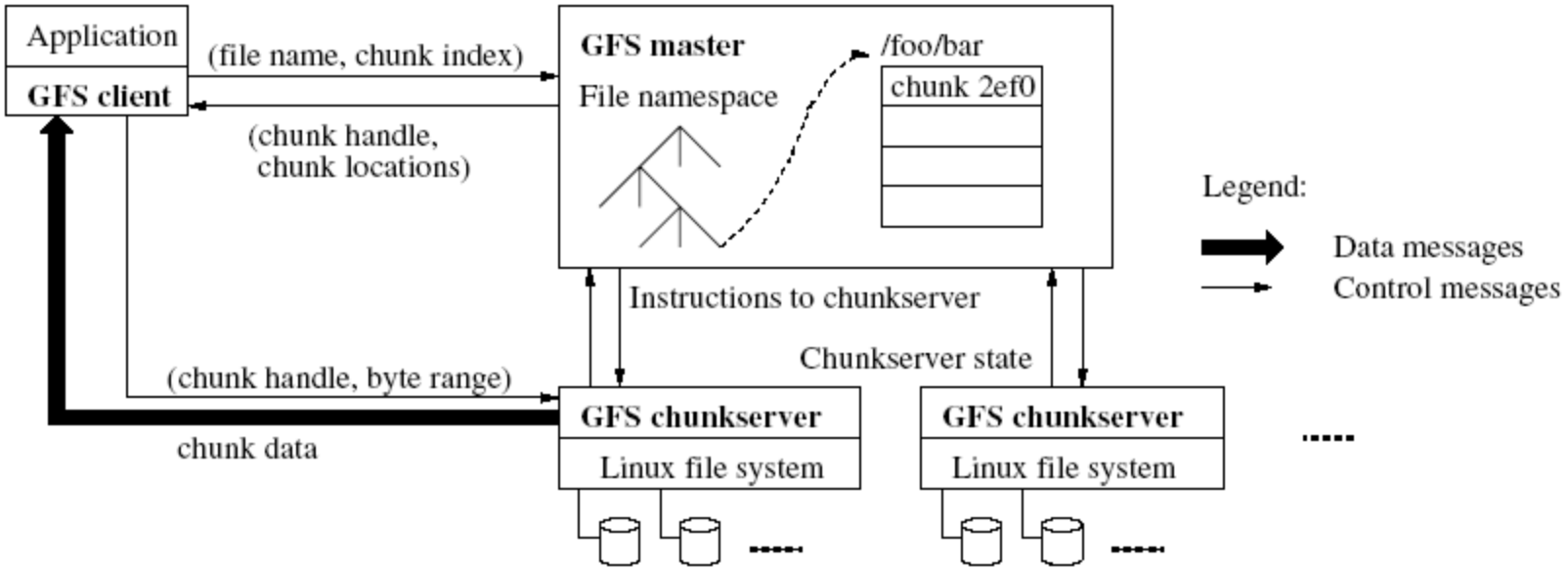
Google File System: Design Criteria

- Detect, tolerate, recover from failures automatically
- Large files, >= 100 MB in size
- Large, streaming reads (>= 1 MB in size)
  - Read once
- Large, sequential writes that append
  - Write once
- Concurrent appends by multiple clients (e.g., producer-consumer queues)
  - Want atomicity for appends without synchronization overhead among clients

GFS: Architecture

- One master server (state replicated on backups)
- Many chunk servers (100s – 1000s)
  - Spread across racks; intra-rack b/w greater than inter-rack
  - Chunk: 64 MB portion of file, identified by 64-bit, globally unique ID
- Many clients accessing same and different files stored on same cluster

GFS: Architecture (2)



Master Server

- Holds all metadata:
  - Namespace (directory hierarchy)
  - Access control information (per-file)
  - Mapping from files to chunks
  - Current locations of chunks (chunkservers)
- Manages chunk leases to chunkservers
- Garbage collects orphaned chunks
- Migrates chunks between chunkservers

Master Server

- Holds all metadata:
  - Namespace (directory hierarchy)

Holds all metadata in RAM; very fast operations on file system metadata

  - Current locations of chunks (chunkservers)
- Manages chunk leases to chunkservers
- Garbage collects orphaned chunks
- Migrates chunks between chunkservers

Chunkserver

- Stores 64 MB file chunks on local disk using standard Linux filesystem, each with version number and checksum
- Read/write requests specify chunk handle and byte range
- Chunks replicated on configurable number of chunkservers (default: 3)
- No caching of file data (beyond standard Linux buffer cache)

Client

- Issues control (metadata) requests to master server
- Issues data requests directly to chunkservers
- Caches metadata
- Does no caching of data
  - No consistency difficulties among clients
  - Streaming reads (read once) and append writes (write once) don't benefit much from caching at client

Client API

- Is GFS a filesystem in traditional sense?
  - Implemented in kernel, under vnode layer?
  - Mimics UNIX semantics?
- No; a library apps can link in for storage access
- API:
  - open, delete, read, write (as expected)
  - snapshot: quickly create copy of file
  - append: **at least once, possibly with gaps and/or inconsistencies among clients**

Client Read

- Client sends master:
  - read(file name, chunk index)
- Master's reply:
  - chunk ID, chunk version number, locations of replicas
- Client sends "closest" chunkserver w/replica:
  - read(chunk ID, byte range)
  - "Closest" determined by IP address on simple rack-based network topology
- Chunkserver replies with data

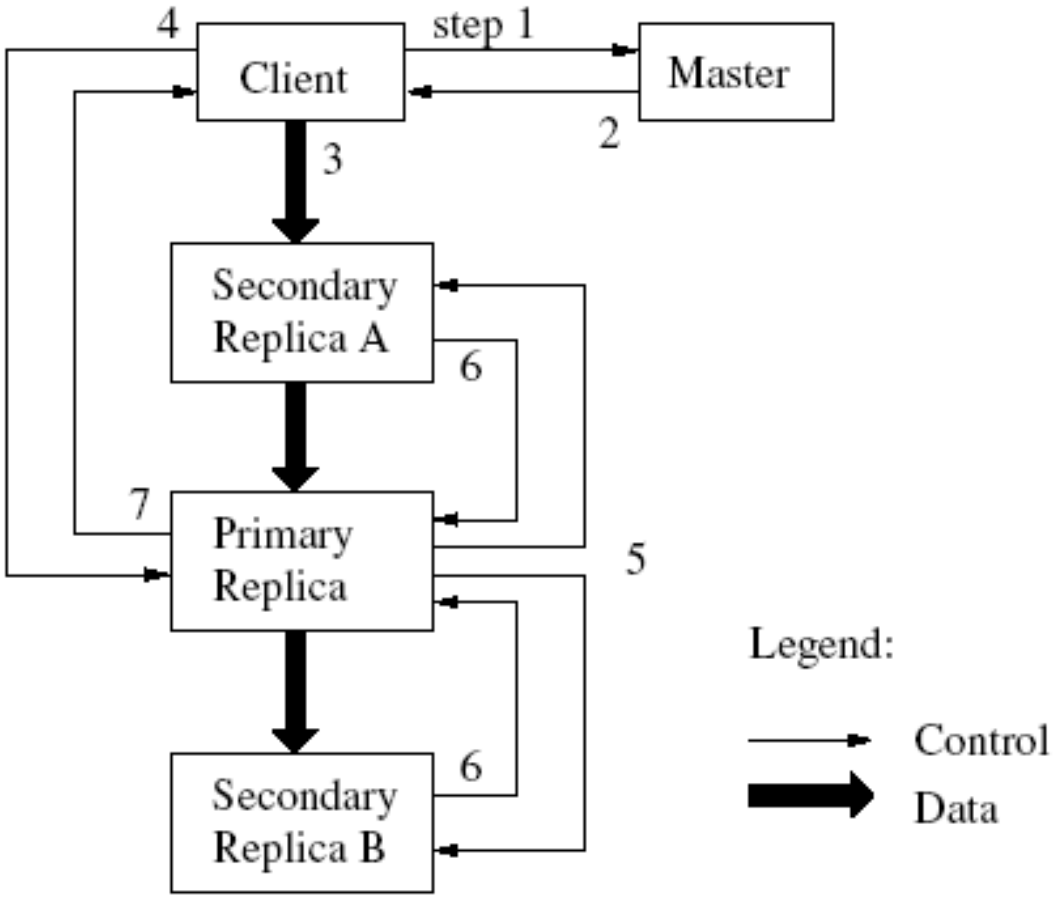
Client Write

- Some chunkserver is primary for each chunk
  - Master grants lease to primary (typically for 60 sec.)
  - Leases renewed using periodic heartbeat messages between master and chunkservers
- Client asks server for primary and secondary replicas for each chunk
- Client sends data to replicas in daisy chain
  - Pipelined: each replica forwards as it receives
  - Takes advantage of full-duplex Ethernet links

Client Write (2)

- All replicas acknowledge data write to client
- Client sends write request to primary
- Primary assigns serial number to write request, providing ordering
- Primary forwards write request with same serial number to secondaries
- Secondaries all reply to primary after completing write
- Primary replies to client

Client Write (3)



Client Record Append

- Google uses large files as queues between multiple producers and consumers
- Same control flow as for writes, except…
- Client pushes data to replicas of last chunk of file
- Client sends request to primary
- Common case: request fits in current last chunk:
  - Primary appends data to own replica
  - Primary tells secondaries to do same at same byte offset in theirs
  - Primary replies with success to client

Client Record Append (2)

- When data won't fit in last chunk:
  - Primary fills current chunk with padding
  - Primary instructs other replicas to do same
  - Primary replies to client, "retry on next chunk"
- If record append fails at any replica, client retries operation
  - So replicas of same chunk may contain different data —even duplicates of all or part of record data
- **What guarantee does GFS provide on success?**
  - Data written at least once in atomic unit

GFS: Consistency Model

- Changes to namespace (i.e., metadata) are atomic
  - Done by single master server!
  - Master uses log to define global total order of namespace-changing operations
- Data changes more complicated
- Consistent: file region all clients see as same, regardless of replicas they read from
- Defined: after data mutation, file region that is consistent, and all clients see that entire mutation

GFS: Data Mutation Consistency

	Write	Record Append
serial success	defined	defined interspersed with inconsistent
concurrent successes	consistent but undefined	
failure	inconsistent	

- Record append **completes at least once, at offset of GFS' choosing**
- **Apps must cope with Record Append semantics**

Applications and Record Append Semantics

- Applications should include checksums in records they write using Record Append
  - Reader can identify padding / record fragments using checksums
- If application cannot tolerate duplicated records, should include unique ID in record
  - Reader can use unique IDs to filter duplicates

Logging at Master

- Master has all metadata information
  - Lose it, and you've lost the filesystem!
- Master logs all client requests to disk sequentially
- Replicates log entries to remote backup servers
- Only replies to client after log entries safe on disk on self and backups!

Chunk Leases and Version Numbers

- If no outstanding lease when client requests write, master grants new one
- Chunks have version numbers
  - Stored on disk at master and chunkservers
  - Each time master grants new lease, increments version, informs all replicas
- Master can revoke leases
  - e.g., when client requests rename or snapshot of file

What If the Master Reboots?

- Replays log from disk
  - Recovers namespace (directory) information
  - Recovers file-to-chunk-ID mapping
- Asks chunkservers which chunks they hold
  - Recovers chunk-ID-to-chunkserver mapping
- If chunk server has older chunk, it's stale
  - Chunk server down at lease renewal
- If chunk server has newer chunk, adopt its version number
  - Master may have failed while granting lease

What if Chunkserver Fails?

- Master notices missing heartbeats
- Master decrements count of replicas for all chunks on dead chunkserver
- Master re-replicates chunks missing replicas in background
  - Highest priority for chunks missing greatest number of replicas

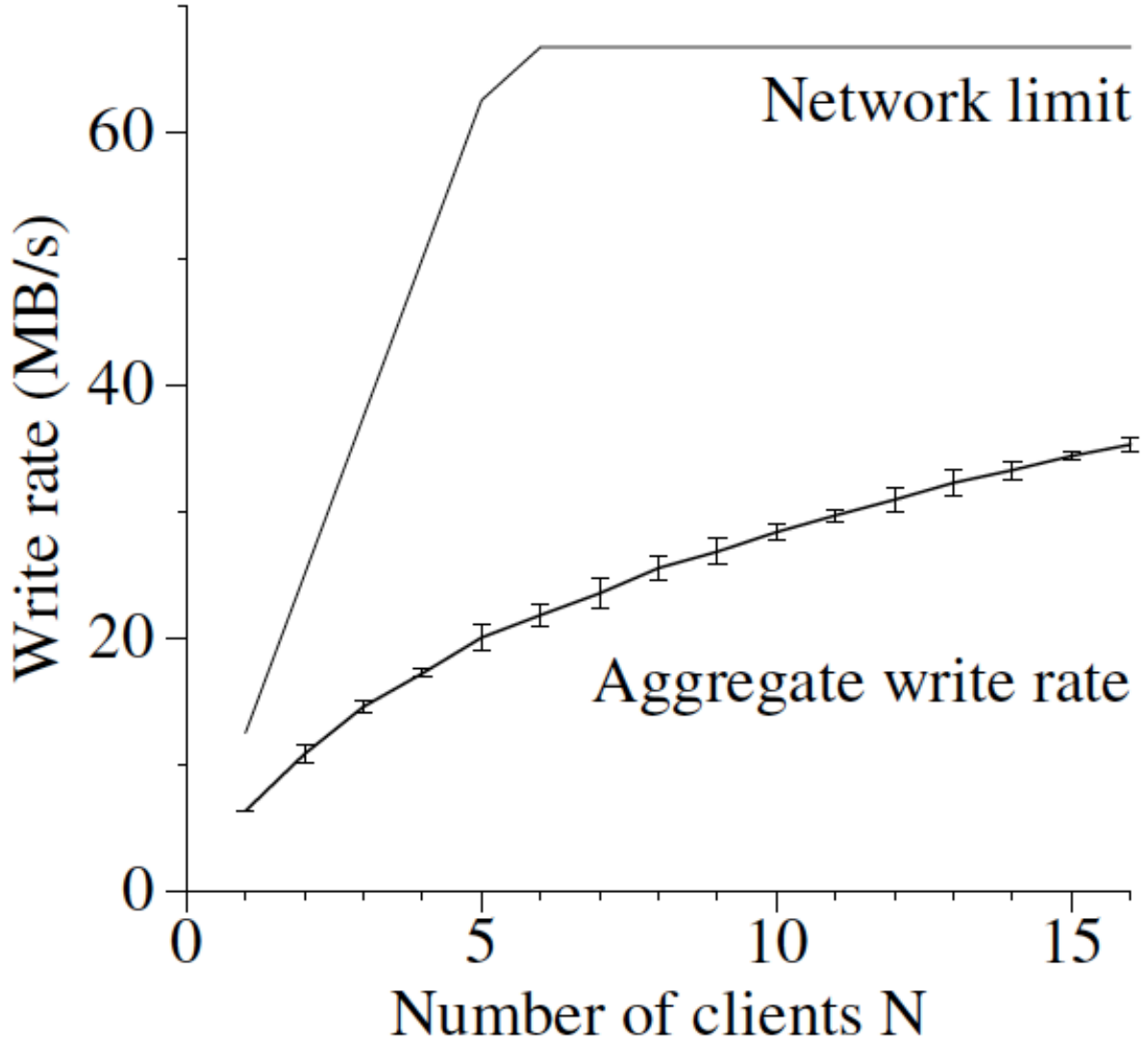
File Deletion

- 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

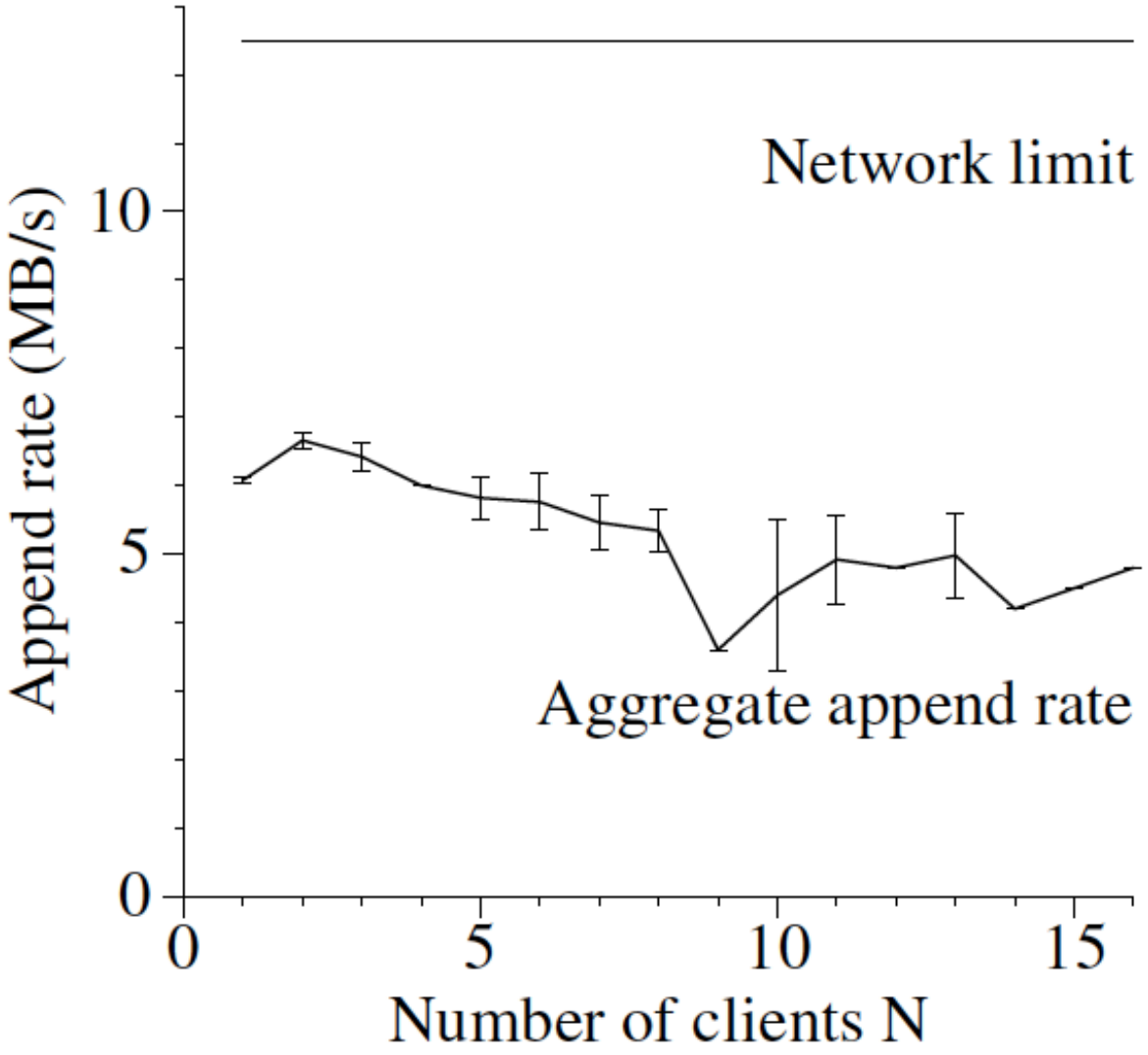
What About Small Files?

- Most files stored in GFS are multi-GB; a few are shorter
- Instructive case: storing a short executable in GFS, executing on many clients simultaneously
  - 3 chunkservers storing executable overwhelmed by many clients' concurrent requests
  - App-specific fix: replicate such files on more chunkservers; stagger app start times

Write Performance (Distinct Files)



Record Append Performance (Same File)



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!)