# The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, Google
*ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

Presented by Stacy Patterson

# Outline

1. Background and Introduction
2. Design Considerations
3. System Description
4. “Evaluation” Results
5. Related Work
6. Conclusions

# Background on File Systems

- “A file system is the methods and data structures than an OS uses to store data.” [1]

- Divides data into logical units – files
  - Addressable by file names
  - Usually also supports hierarchical nesting (directories)

- Supports operations to access data:
  - Create file, write to file, read from file, delete file, etc.

# Distributed File Systems

- Data stored on different machines.
- Provides same (similar) interface as centralized file system.

- Should handle concurrent access to files in some way.

- May provide:
  - Replication for fault tolerance.
  - Caching at clients for performance.

# Objectives of Google File System

- A distributed file system that meets Google's needs.
- Standard distributed systems stuff:
  scalability, reliability, availability
- Google-specific needs:
  - Runs on hundreds/thousands of commodity machines.
  - Failures and errors are the norm.
  - Files are "huge" – Multi GB.
  - Most file mutations are appends, e.g., data streams from continuously running apps.

- Google can co-design file system API and applications.

# Design Assumptions

- System components often fail.

- Store modest number of large files.
  Few million files, each 100 MB or larger.

- Reads: primarily large streaming or small random.
- Writes: often large sequential appends.
  - Often concurrent – hundreds of simultaneous producers.

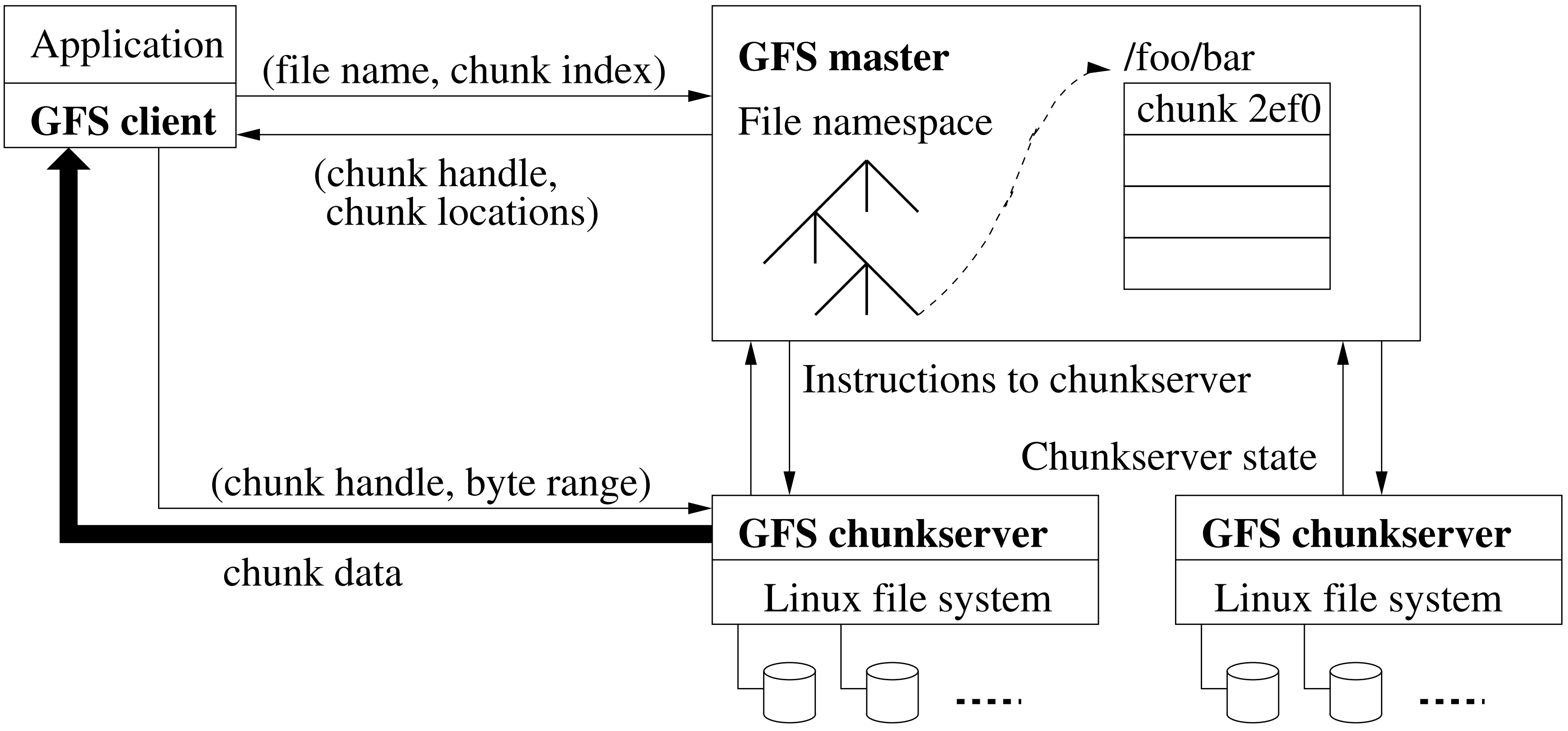
- High sustained bandwidth more important than low latency.

# Goals for Interface

- Provide familiar file system interface:
  - File operations: open, close, read, write.

- Also provide
  - Snapshot operation: create copy of directory tree
  - Record-append operation: atomic operation to append record to end of file.

# GFS Architecture



# File Chunks

- Files divided into fixed sized chunks – 64 MB
  - Each chunk has unique ID, assigned by master.
  - Client can translate file offset into chunk index.
- Chunkservers store chunks as Linux files.
  - Chunk replicated on multiple chunkservers (3 by default).

- Clients do not cache chunk data.

# Chunk Size

- Advantages of “large” chunk size:
    - Limits client interaction with master.
    - For sequential access, client can use persistent TCP connection to chunkserever.
    - Reduces amount of metadata at master.

- Disadvantages
    - Internal fragmentation – GFS uses lazy space allocation to minimize this.
    - Hot spots.

# The Master

- GFS has a **single** master.
- Master stores metadata: namespace and chunk information.
  - All metadata (e.g., namespace) changes must be done through the master.

- Master controls system-wide activities:
  - Chunk lease management
  - Garbage collection
  - Chunk migration

- Communicates with chunkservers in heartbeat messages.
  - Give instructions and collects chunkserver state

# More about the Master

- Master stores 3 types of metadata
  - (1) File and chunk namespaces
  - (2) Mapping from files to chunks
  - (3) Locations of chunk replicas.
- All stored in memory.
- (1) and (2) also persisted in **operation log**.
  - Stored on disk and replicated on other machines.

- Master learns (3) on startup by asking chunkservers what chunks they have

# Operation Log

- Record of metadata changes (add/delete files and directories).
- Defines order of concurrent operations.
- Write-ahead log
  - Master responds to client requests only after flushing records to local and remote disks.

- On startup, master recovers state by replaying log.
  - Small log = fast startup.

- Limit log size through non-blocking checkpointing.
  - Switch to new log file and create checkpoint in separate thread.

# System Interactions

- Mutation: an operation that changes the contents or metatdata of a chunk.
  - E.g., write or append.

- Each chunk has primary replica and several secondary replicas.
  - Master grants lease to primary (60 seconds)
  - Primary can request and receive lease extensions.
  - Master can revoke lease.

- Primary picks serial order for applying all mutations to a chunk.
  - All replicas follow this order.

# Read Operation

1. Application issues read request.
2. GFS client translates read request into file name and chunk index.
   Requests chunk locations from master.
3. Master responds with chunk name, primary and secondary replica locations.
   Client caches location info.

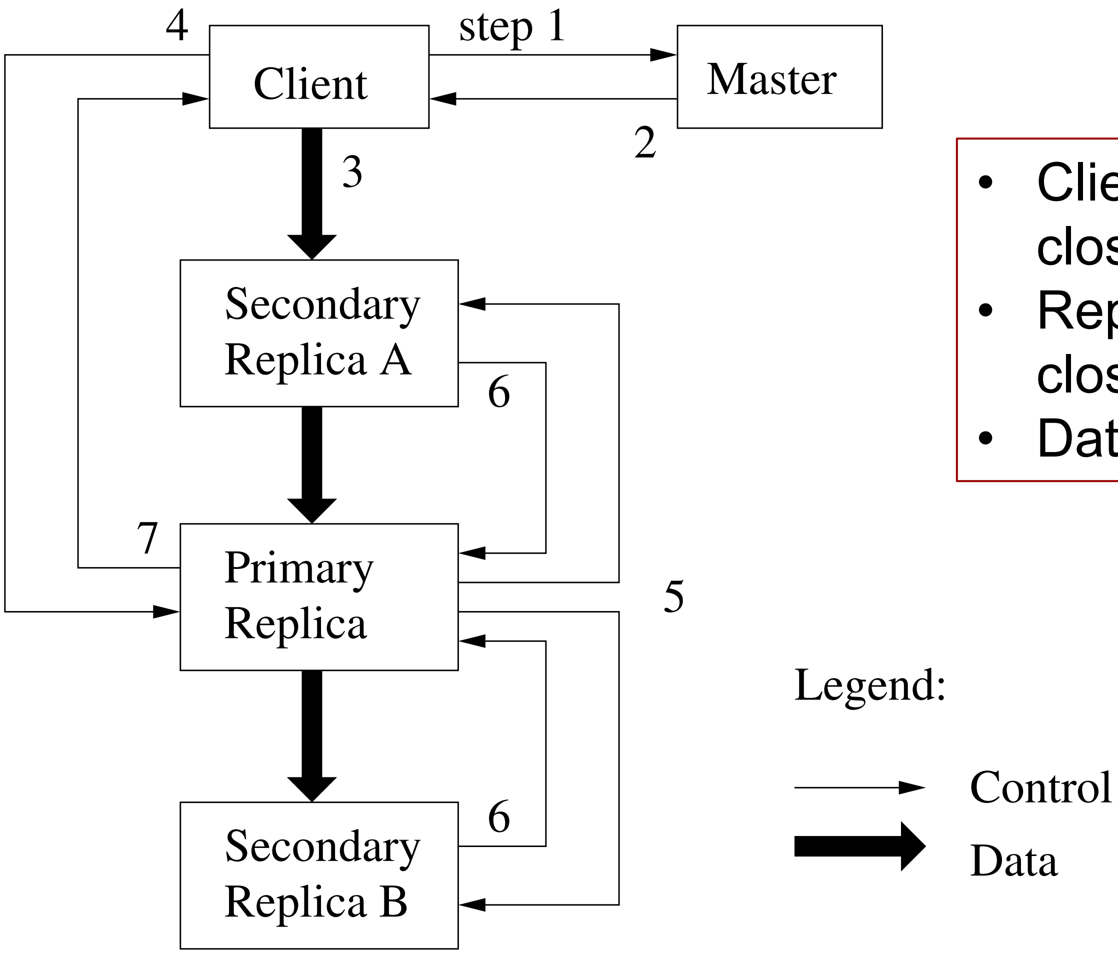
4. Client picks replica (usually closest) and sends read request.
5. Chunkserver responds with data.
6. Client forwards data to application.

# Write Operation

Application issues write request

1. GFS client translates request to filename, chunk index and sends request to master.
   If no chunkserver has lease, master grants one.
   Sends primary and secondary chunkserver locations to client.
2. Client caches this info.
   Only contacts master again if primary fails or primary loses lease.
3. Client sends data to replicas (pipelined).

# Data Flow



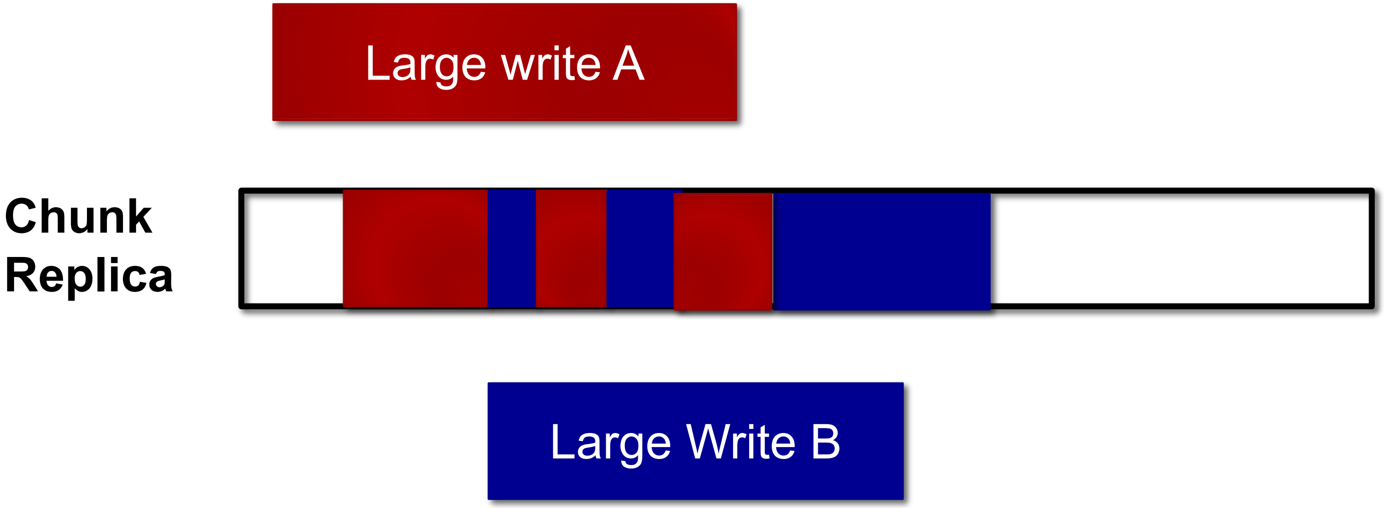
- Client forwards data to closest replica.
- Replica forwards to next closest replica, etc.
- Data pipelined over TCP

# Write Operation (2)

4. After all replicas have data, client sends write request to primary.
   Primary assigns serial number to mutation.
5. Primary forwards write request and serial number to all replicas.
   Secondaries apply mutations in serial number order.
6. Secondaries reply to primary.
7. Primary replies to client.
   Reports any errors at replicas so client can retry if needed.

# Write Operation (3)

- If write operation is large or involves multiple chunks, GFS client breaks it into several smaller write operations.
- These may be interleaved with concurrent writes from other clients.
- Replicas have same mutation order, but writes are not serialized.



Large write A

**Chunk Replica**

Large Write B

# Record-Append Operation

- Record-append is atomic.
- Client only specifies file. GFS determines the file offset.

- Similar to write operation, except:
  - Client sends data to all replicas that hold last file chunk (pipelined)
  - Client sends appendrequest to primary.
  - If record fits in chunk
    - Primary appends record. Sends offset to secondaries
    - Secondaries write data at same offset.
  - If record does not fit:
    - Primary pads chunk to max size. Tells secondaries to do same.
    - Tells client to retry on next chunk.

# Snapshot Operation

- Snapshot makes a copy of a file or directory tree.
- Uses standard copy-on-write.

- When master receives snapshot request, revokes leases.
  - All subsequent writes will require contacting master.

- On first write request for a chunk C:
  - Master defers replying to client.
  - Picks new chunk handle C'.
  - Asks all chunkservers that hold C to copy to C'.
  - Master grants replica lease on C' and replies to client.

# Consistency Model

- For file region:
  - Region is **consistent** if all clients will see same data, no matter which replica is read.
  - Region is **defined** if, after a mutation, it is consistent and all clients see the entire mutation.

- Non-concurrent successful mutation leaves region defined.
- Concurrent successful mutations leave region consistent, but maybe not defined.
  - Mingled fragments from multiple mutations.

- Failed mutation leaves region undefined.

# Consistency Model (2)

- Replicas may have defined regions interleaved with undefined regions (because of failed mutations).

- Record-append:
  - If record append fails at any replica, client retries.
  - But some replicas may have record before retry.
  - Retry gets new offset.
  - Record-append may have duplicates entries.

  - Applications must be able to deal with duplicates.

# Namespace Management

- File namespace mutations are atomic – all handled by master.
- Use read locks and write locks to manage concurrent operations.
  - Can create two files in same directory at same time.
    - 2 read locks on directory
    - 1 write lock for each new file
  - Can’t create a file in directory while it is being snapshotted.
    - Write lock required for both operations

# Replica Placement

- Master decides where to place replicas.

- Goals of placement policy
  - 1. Maximize reliability and availability.
  - 2. Maximize network bandwidth utilization.

- Not enough to put replicas on multiple machines.
- Also must put replicas on different racks.
  - Ensures chunks survive single rack failure.
  - Also means chunk access can use bandwidth on multiple racks.
  - Drawback: write must access multiple racks

# Replica Management

- Replica placement policy:
  - 1. Place new chunks on under-utilized servers.
  - 2. Limit number of new chunks on single server.
    - New chunks accessed more frequently.
  - 3. Place replicas on different racks

- Locations for new chunks and new replicas chosen according to this policy.

- Master periodically rebalances.
  - Moves replicas for better load distribution.

# Garbage Collection

- When file is deleted:
  - Master logs the delete operation.
  - Master renames file to hidden file.
- Periodically, master scans namespace
  - Deletes hidden files over 3 days old and removes metadata.
  - Identifies orphan chunks and erases metadata.
- In heartbeat messages:
  - Chunkservers tell master which chunks they have.
  - Master replies with list of chunks that can be deleted.

# Stale Replica Detection

- Chunk may become stale if replica misses mutation.
  - E.g., If it is down when mutation occurs.

- Master maintains chunk version number.
  - New version number assigned when lease is granted to primary.
  - All replicas record version number.

- When chunkserver restarts
  - Reports its list of chunks and versions numbers to master.
  - Master checks for stale replicas by checking version numbers.
  - If replica is stale, Master acts as if it does not exist.
- Stale replicas removed in regular garbage collection.

# Fault Tolerance Details

- Designed for high availability
  - Fast recovery of master and chunkservers (seconds).
  - Chunk replication.
  - Master replication.
  - Shadow masters provides read-only access.

- Data Integrity
  - Chunkservers use checksumming to detect data corruption.
  - Checksums over 64 KB blocks.
  - If data is corrupted, master makes another replica. Corrupted chunk replica is garbage collected.

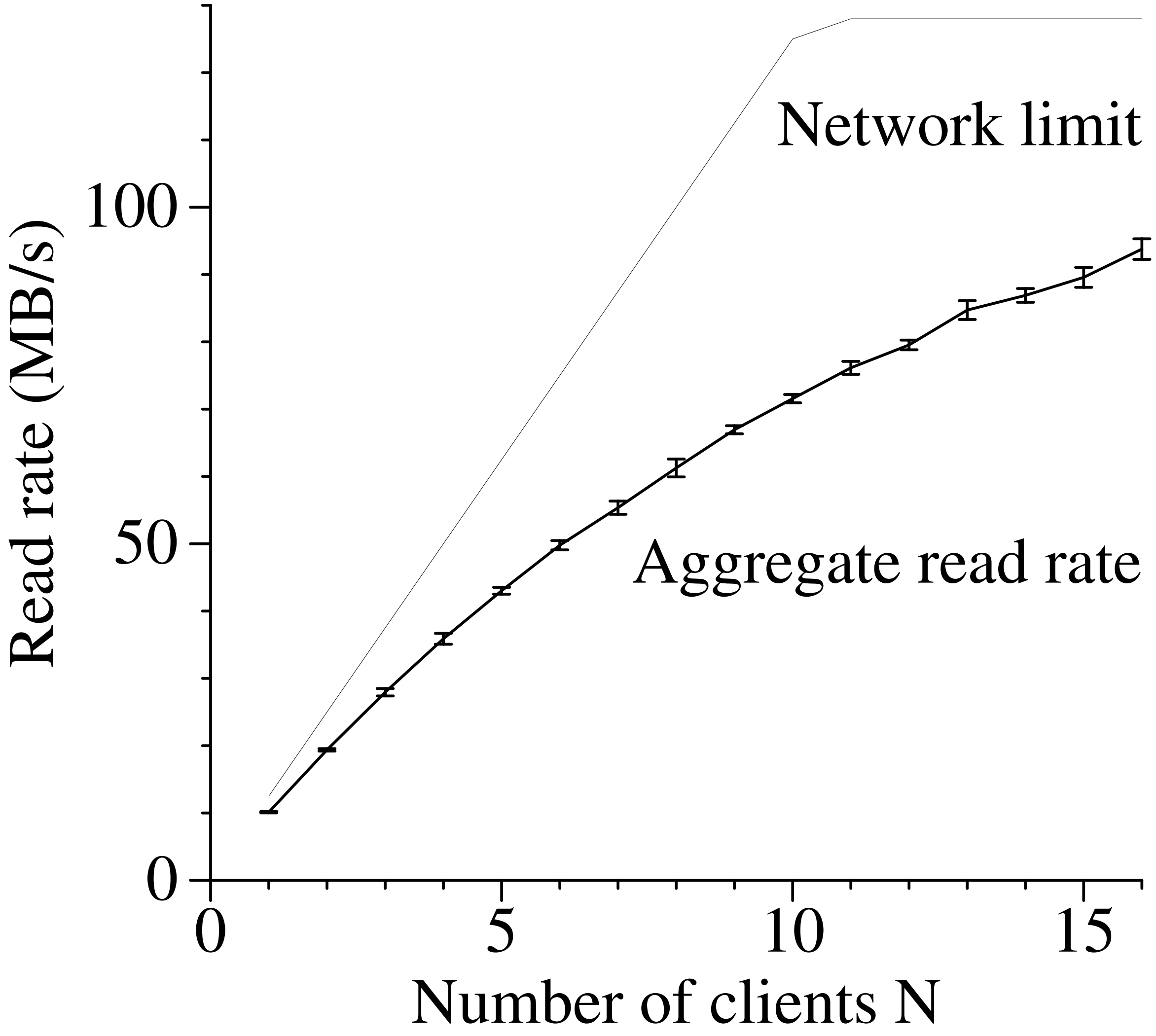
# Evaluation Setup

- Micro-benchmarks
  - One master, 2 master replicas
  - 16 chunkservers
  - 16 clients
  - All machines:  dual 1.4 GHz PIII processors, 2GB memory, 2 80 GB 5400 rpm disks, 100 Mbs full-duplex ethernet.
  - All GFS machines connected one switch.
    All clients connected to another.
    1Gbps link between them.

# Micro-Benchmark: Reads

- N clients reads simultaneously.
- Each client selects random 4MB region from 320 GB file set.

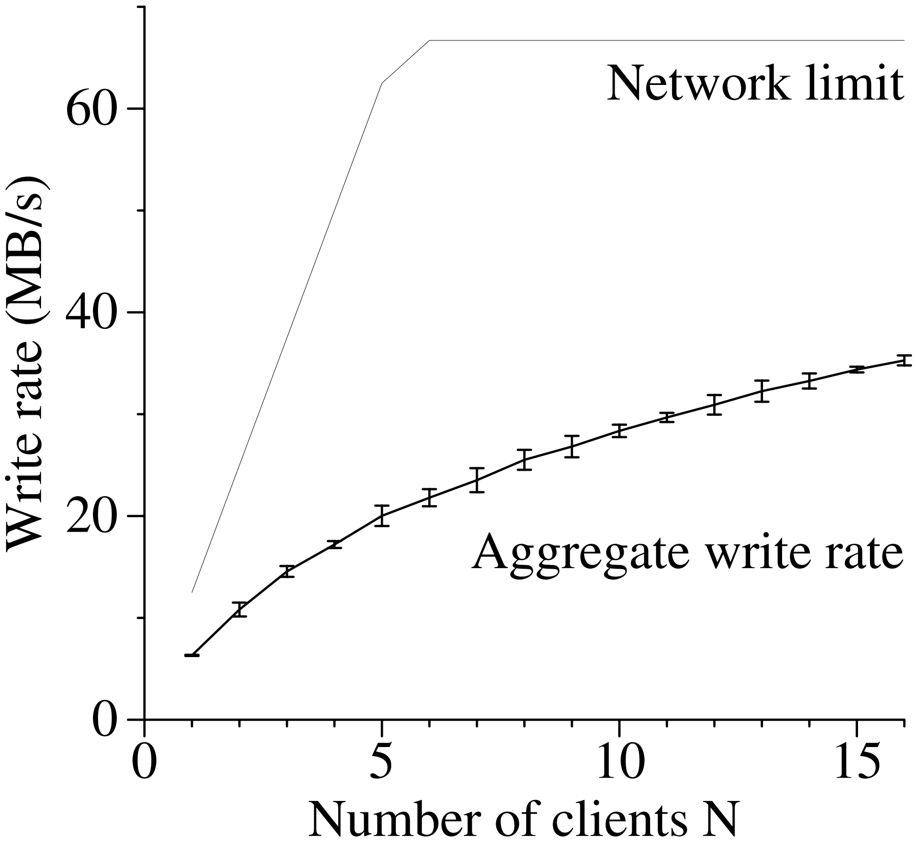
- Repeated 256 times, so each client reads 1GB data.


Network limit
Aggregate read rate
Read rate (MB/s)
0
50
100
0
5
10
15
Number of clients N

(a) Reads

# Micro-Benchmark: Writes

- N clients write simultaneously to N distinct files.
- Each writes 1GB data to new file, series of 1MB writes.
  - No concurrent writes to same chunk.

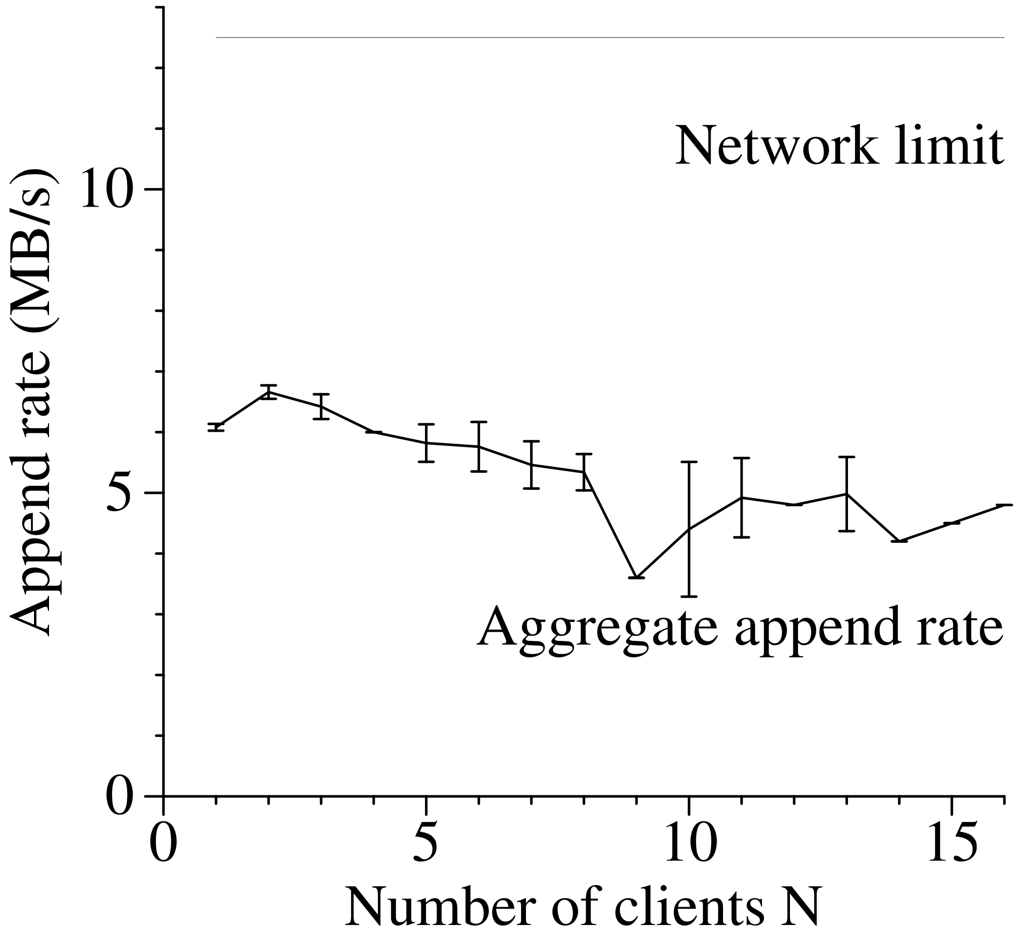

Network limit
Aggregate write rate
Write rate (MB/s)
0
20
40
60
0
5
10
15
Number of clients N

(b) Writes

# Micro-Benchmark: Record Appends

- N clients append to single file simultaneously.

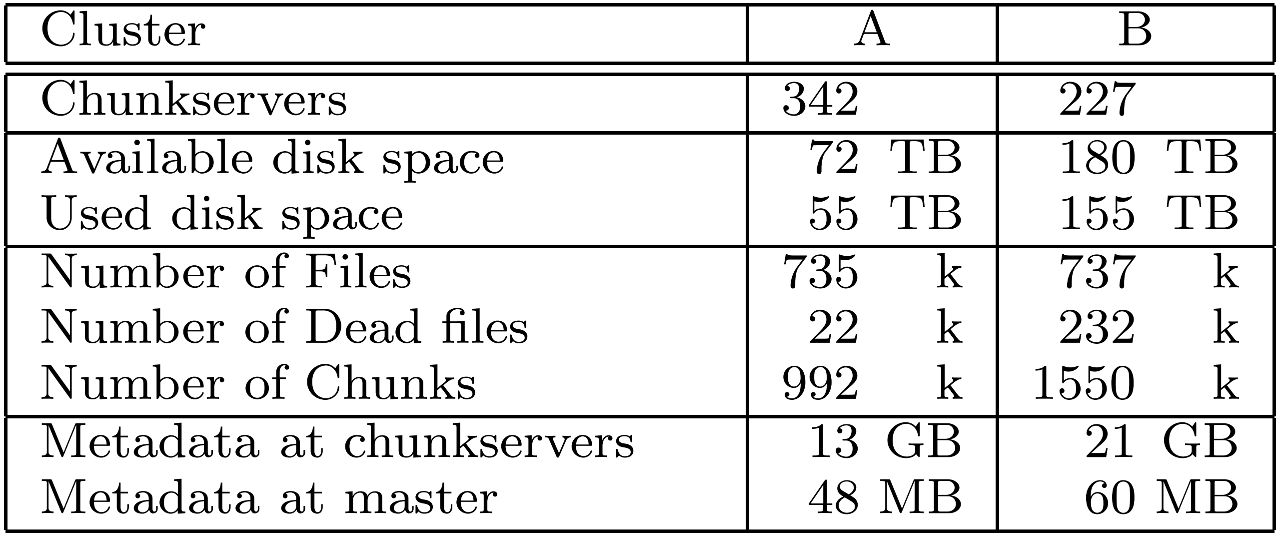
- What do they append?
- How many append invocations are there?



(c) Record appends

# Real World Clusters

- Cluster A: used for research and development
  - Typical task runs a few hours.
  - Reads a few MBs to few TBs, transform data, and writes back to cluster.
- Cluster B: used for production data processing
  - Tasks much longer.
  - Continuously generate and process multi-TB data sets.



| Cluster | A | B |
|---|---|---|
| Chunkservers | 342 | 227 |
| Available disk space | 72 TB | 180 TB |
| Used disk space | 55 TB | 155 TB |
| Number of Files | 735 k | 737 k |
| Number of Dead files | 22 k | 232 k |
| Number of Chunks | 992 k | 1550 k |
| Metadata at chunkservers | 13 GB | 21 GB |
| Metadata at master | 48 MB | 60 MB |

# Results

| Cluster | A | B |
|---|---|---|
| Read rate (last minute) | 583 MB/s | 380 MB/s |
| Read rate (last hour) | 562 MB/s | 384 MB/s |
| Read rate (since restart) | 589 MB/s | 49 MB/s |
| Write rate (last minute) | 1 MB/s | 101 MB/s |
| Write rate (last hour) | 2 MB/s | 117 MB/s |
| Write rate (since restart) | 25 MB/s | 13 MB/s |
| Master ops (last minute) | 325 Ops/s | 533 Ops/s |
| Master ops (last hour) | 381 Ops/s | 518 Ops/s |
| Master ops (since restart) | 202 Ops/s | 347 Ops/s |

# Recovery

- Killed single chunkserver in cluster B (15,000 chunks and 600GB of data)
  - All chunks restored 23.2 minutes.
- Killed two chunkservers – resulted in 266 chunks having single replica
  - Achieved 2x replication of these chunks in 2 minutes.

# Related Work (1)

- Andrew File System (AFS) [Howard 1998, CMU]
  - Provides weak consistency model.
    - Reads and writes done on local cached copy.
    - Writes committed to file on close.
    - Server informs clients of updates to cached files.
  - Not intended for large, shared data applications.
  - Like GFS, provides location-independent namespace.

- Simplified version available for Linux.
- Descendent of Coda file system.

# Related Work (2)

- Global File System (GFS) [Soltis 1996]
  - Journaled file system
  - Views storage as a Network Storage Pool.
  - Concurrency control using locks.
    - Can use a distributed lock manager
  - Location-dependent namespace.

  - GFS2 included in RedHat and CentOS.

# Related Work (3)

- Similarities with other distributed file systems
  - AFS also provides a location independent namespace.
  - GFS places data in manner similar to xFS [Anderson 1995] and Swift [Cabrera 1991] – for aggregate performance fault tolerance.

- Differences
  - Frangipani [Chandramohan 1997] and Intermezzo provide caching.
  - Frangipani, Intermezzo, Minnesota’s GFS, GPFS, use distributed algorithms for consistency.
  - Harp uses a primary copy scheme – gives stronger consistency guarantees.

# Related Work (3)

- GFS architecture resembles Network Attached Secured Disk Architecture.
    - Chunk servers act like network attached drives.

# Conclusions

- GFS designed specifically for Google's environment and applications.
- Unique features:
  - Large chunk size.
  - Optimized for concurrent record-appends and long sequential reads.
  - Online repair mechanism to replace lost replicas.

- Delivers high throughput to many concurrent readers and writers.
- Uses centralized master in scalable way.

- Works for Google … or does it?

# Collosus

- 2010 – Google stopped using GFS
- Instead uses Collosus.

- GFS built for batch operations (MapReduce).
- Collosus built for real-time services.
  - Chunk size is 1MB

- Includes multiple master nodes.

# HDFS vs GFS

- HDFS = Hadoop Distributed File System
  - Paper from 2010

- Built based on GFS
- Written in Java

- Similar architecture to GFS
  - Master → NameNode
  - Chunkserver → DataNode
  - Chunk → Block

# HDFS vs GFS (2)

Major differences

- Chunk size usually 64MB or 128MB by default
  - But can be specified by application.
- Synchronized access to files.
  - No concurrent writes.
  - Can read a block while it is being written.

- Still uses single master node
  - But also has a “backup master” – keeps copy of metadata in memory
  - Can be used as a read-only name node.

# HDFS (Hadoop 2.0, 2012)

- Added automatic failover when NameNode fails.

- Need
  - Mechanism to detect NameNode failure.
  - Mechanism to elect new NameNode.

  - These are provided by ZooKeeper.

# References

- [1] http://www.tldp.org/LDP/sag/html/filesystems.html