The Google File System

By Sanjay Ghemawat, Howard Gobioff, and
Shun-Tak Leung

(Presented at the 19th ACM Symposium on
Operating Systems Principles)

Introduction

- Google – a well known search engine.
- In-house applications process large amounts of data.
- Google needs a distributed file system that matches its applications needs.
- This paper discusses the solution: Google File System (GFS).

Topics

- GFS Design Motivations
- GFS Architecture
- Distributed Read/Write Algorithms
- System Interactions
- Fault-Tolerance Techniques
- Performance Results

Google Architecture Facts

- More than 15,000 commodity-class PC's.
- Fault-tolerance provided in software.
- More cost-effective solution.
- Multiple clusters distributed worldwide.
- One query reads 100's of MB of data.
- One query consumes 10's of billions of CPU cycles.
- Thousands of queries served per second.
- Google stores dozens of copies of the entire Web!

Conclusion: Need large, distributed, highly fault-tolerant file system.

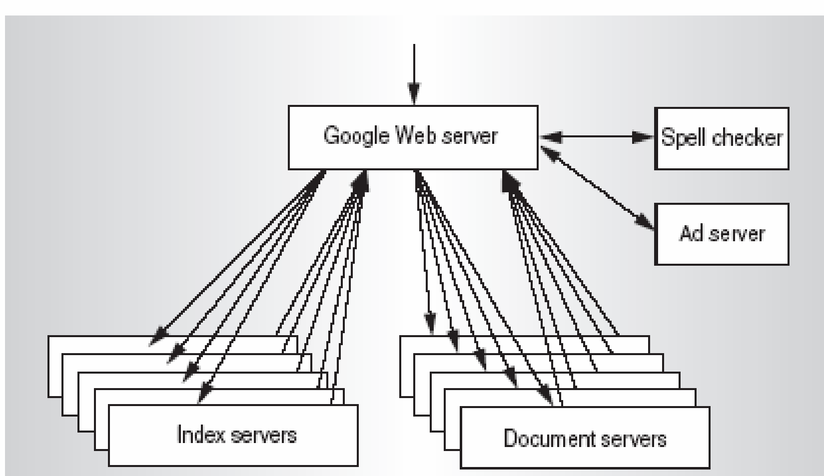
Example: Serving a Query

Overview

- http://www.google.com/search?q=google+file+system
- Browser gets load-balanced I.P. address from DNS.
- Browser sends HTTP request to Google cluster.
- Google Web Server (GWS) is chosen to coordinate the response.
- Response generation is parallelized over multiple machines.
- Google Web Server sends results formatted as HTML back to browser.

Example: Serving a Query

Response Generation



Example: Serving a Query

Response Generation

- For each word in the query, GWS gets a set of docID's from the Index Servers.
- GWS intersects the docID sets and orders them using the PageRank algorithm.
- GWS gets title, link, and paragraph information for each docID from the "Document Servers".
- GWS also communicates with the Ad and Spell-checking Servers.
- Results are formatted as HTML and sent back.

Example: Serving a Query

Response Generation

- Note how *one* response generation is distributed over *many* machines:
    - *Spell checker*
    - *Index servers*
    - *Document servers*
    - *Ad servers*

Design Motivations

1. GFS runs on a large number of machines. Failures occur regularly, so fault-tolerance and auto-recovery need to be built in.
2. File sizes are ***much*** larger. Standard I/O assumptions (e.g. block size) have to be re-examined.
3. Record appends are the prevalent form of writing. Need good semantics for concurrent appends to the same file by multiple clients.
4. Google applications and GFS are both designed in-house - so they can and should be co-designed.

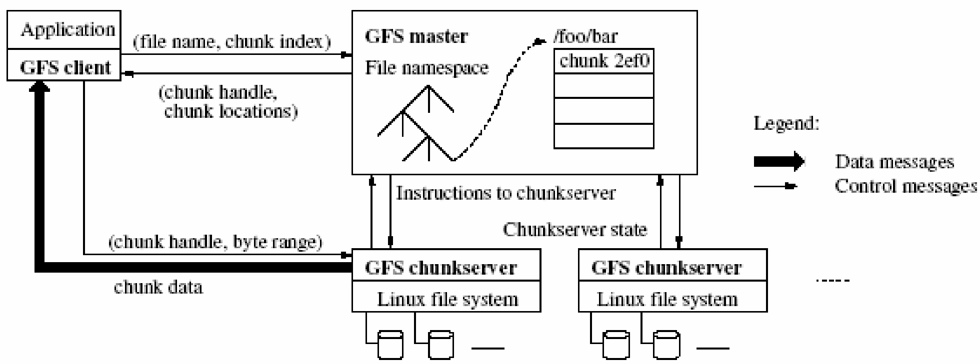
GFS Architecture

Analogy

- On a single-machine FS:
  - An upper layer maintains the metadata.
  - A lower layer (i.e. disk) stores the data in units called “blocks”.
- In the GFS:
  - A master process maintains the metadata.
  - A lower layer (i.e. a set of chunkservers) stores the data in units called “chunks”.

GFS Architecture

Diagram



GFS Architecture

Overview

1. **Master:**
   - Single process running on a separate machine.
   - Maintains metadata (file namespace, access control information, file to chunk mappings, chunk replica locations).

2. **Chunkserver:**
   - A machine with Linux FS.
   - Stores chunks as files. Chunk size – 64 MB!
   - Each chunk is identified by a globally unique handle.
   - Chunks are replicated over multiple machines for fault-tolerance (Default: 3).
   - There are hundreds of chunkservers in a GFS cluster distributed over multiple racks.

3. **Client (many of them):**
   - Google application with GFS client linked in.
   - Gets necessary metadata for read/write operations from master.
   - Actual read/write only involves data flowing between client and chunkserver.

Distributed Read Algorithm

1. Client application originates read request (file, byte range).
2. Linked-in GFS client translates to (file, chunk index) and sends to master.
3. Master responds with chunk handle and locations of all chunk replicas.
4. Client chooses chunkserver and reads replica from it.

Distributed Write Algorithm

1. Client application originates write request.
2. Linked-in GFS client translates to (file, chunk index) and sends to master.
3. Master replies with primary/secondary replica locations.
4. Client pushes write data to all replica locations. Note: data is stored in buffers, but not yet written.
5. Primary looks at all instances of write data for the chunk in its buffer and chooses serial order.

Distributed Write Algorithm

6. Primary performs the writes in that order.
7. Primary communicates the serial order to the secondaries and they also write to disk.
8. Secondaries report back to primary. Primary reports to client.
9. If error occurs at any of replicas, client retries write.

Note: This algorithm allows multiple clients to write the same chunk in parallel, because of the serial order mechanism.

Record Append Algorithm

Important operation for Google applications:

- Merging results from multiple machines.
- File used as producer - consumer queue.

1. Client sends (file, chunk index) request to master.
2. Master replies with primary/secondary replica locations.
3. Client pushes write data to all replica locations.
4. Primary checks if write extends beyond chunk boundary.

Record Append Algorithm

1. If yes:
    1. Primary pads its chunk.
    2. Primary tells secondaries to pad their chunks.
    3. Primary informs client and tells him to retry on next chunk.
2. If no:
    1. Primary appends to its chunk.
    2. Primary tells secondaries to append to their chunks.
    3. Finally, primary reports success to client.
3. If error occurs at any of replicas, client retries write.

Note: This algorithm allows multiple clients to append to the same file in parallel.

Master <-> Chunkservers

- Master communicates with chunkservers regularly to obtain state:
    - Is chunkserver up or down?
    - Are there any disk failures on chunkserver?
    - Are any replicas corrupted?
    - Are there any stale replicas (chunkserver was down during a write to one of its replicas)?
    - Which chunk replicas does this chunkserver store?
- … and to send instructions.

Master Operations

- Master manages file namespace.
- Master periodically scans through its entire state in the background during which it performs the following operations:
  - Re-replicating chunks when:
    - Replica quota for a chunk is increased.
    - When a chunkserver with a replica is down.
    - When a replica becomes corrupted..
  - Rebalancing chunks regularly for load balancing.
  - Managing garbage collection (deleting file and chunk metadata for deleted files).

Fault Tolerance Mechanisms

- Fast Recovery: master and chunkservers are designed to restart and restore state in a few seconds.
- Chunk Replication: across multiple machines, across multiple racks.
- Master Replication:
  - Master state is replicated on multiple machines.
  - Master operation log and checkpoints also replicated on multiple machines.
  - Monitoring process automatically starts up new master process if the old one fails.
  - "Shadow" masters for reading data if "real" master is down.
- Data integrity verification accomplished through checksums.

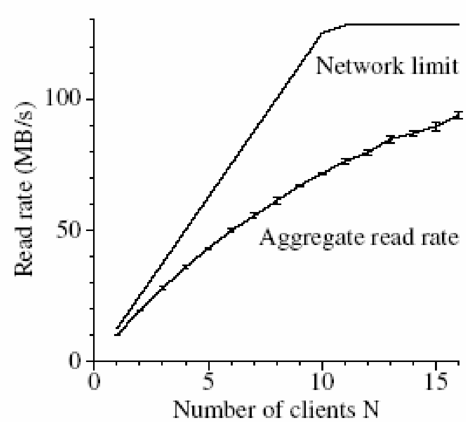
Performance Results

Test Cluster

- Performance measured on cluster with:
  - 1 master
  - 16 chunkservers
  - 16 clients
- Each server machine connected to central switch by 100 Mbps Ethernet.
- Each client machine also connected to central switch by 100 Mbps Ethernet.
- Both switches connected with a 1 Gbps link.

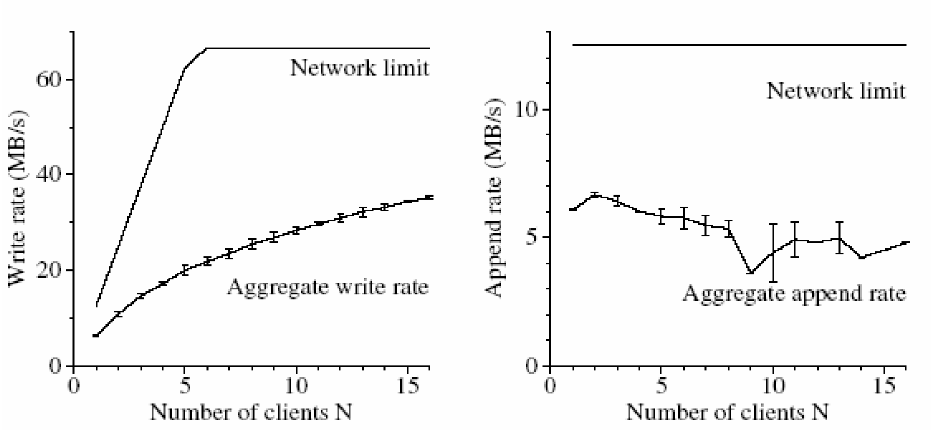
Performance Results

Test Cluster



Performance Results

Test Cluster



Performance Results

Real-world Clusters

- Cluster A:
  - Used for research and development.
  - Used by over a hundred engineers.
  - Typical task initiated by user and runs for a few hours.
  - Task reads MB's-TB's of data, transforms/analyzes the data, and writes results back.
- Cluster B:
  - Used for production data processing.
  - Typical task runs much longer than a Cluster A task.
  - Continuously generate and process multi-TB data sets
  - Human users rarely involved.
- Clusters have been running for about a week when measurements were taken.

Performance Results

Real-world Clusters

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

Cluster Characteristics

Performance Results

Real-world Clusters

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

Performance Metrics for Clusters

Performance Results

Real-world Clusters

An experiment in recovery time:

- One chunkserver in Cluster B killed.
- Chunkserver has 15,000 chunks containing 600 GB of data.
- Limits imposed:
  - Cluster can only perform 91 concurrent clonings (40% of number of chunkservers).
  - Each clone operation can consume at most 6.25 MB/s.
- Took 23.2 minutes to restore all the chunks.
- This is 440 MB/s.

Conclusion

- Implementation of GFS was motivated by needs specific to Google's workload.
- GFS is a successful distributed file system running on a large amount of commodity machines.
- Distributed Read / Write / Record Append algorithms were discussed.
- Fault-tolerance is built into the system which automatically monitors itself.
- Performance results indicate that the system is a viable solution to Google's needs.