The Google File System (GFS) + HDFS

CS4230 – Distributed and Cloud Computing

Jay Urbain, PhD

- "Google's Colossus Makes Search Real-Time by Dumping MapReduce", High Scalability (World Wide Web log), 2010-09-11.
- Carr, David F (2006-07-06), "How Google Works."
- Ghemawat, S.; Gobioff, H.; Leung, S. T. (2003). "The Google file system". Proceedings of the nineteenth ACM Symposium on Operating Systems Principles - SOSP '03.

Introduction

- Google's search engine and related applications process a *lot* of data.

Google's first production server rack, circa 1998

Motivating Facts

Google processes its data in computing clusters

- Google makes their own servers – use commodity-class CPUs running customized versions of Linux.
- Seek to maximize performance per dollar, not absolute performance.
- How this is specifically measured is not public information – includes system running costs and power consumption
- Novel switching power supply design to reduce costs and improve power efficiency.

Motivating Facts

- Google processes 3.5 billion requests per day (2015)
- Stores 10 exabytes of data (10 billion gigabytes!)

Multiples of bytes				
SI decimal prefixes		Binary usage	IEC binary prefixes	
Name (Symbol)	Value		Name (Symbol)	Value
kilobyte (kB)	10^3	2^{10}	kibibyte (KiB)	2^{10}
megabyte (MB)	10^6	2^{20}	mebibyte (MiB)	2^{20}
gigabyte (GB)	10^9	2^{30}	gibibyte (GiB)	2^{30}
terabyte (TB)	10^{12}	2^{40}	tebibyte (TiB)	2^{40}
petabyte (PB)	10^{15}	2^{50}	pebibyte (PiB)	2^{50}
exabyte (EB)	10^{18}	2^{60}	exbibyte (EiB)	2^{60}
zettabyte (ZB)	10^{21}	2^{70}	zebibyte (ZiB)	2^{70}
yottabyte (YB)	10^{24}	2^{80}	yobibyte (YiB)	2^{80}
See also: Multiples of bits · Orders of magnitude of data				

Motivating Facts

- Thousands of queries served per second.
- One query can read 100's of *MB* of data.
- One query can consume 10's of billions of CPU cycles.
- Google stores *dozens* of copies of their Web index!
- Reliability is achieved through *software* not individual hardware systems – they *assume hardware failure*.

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

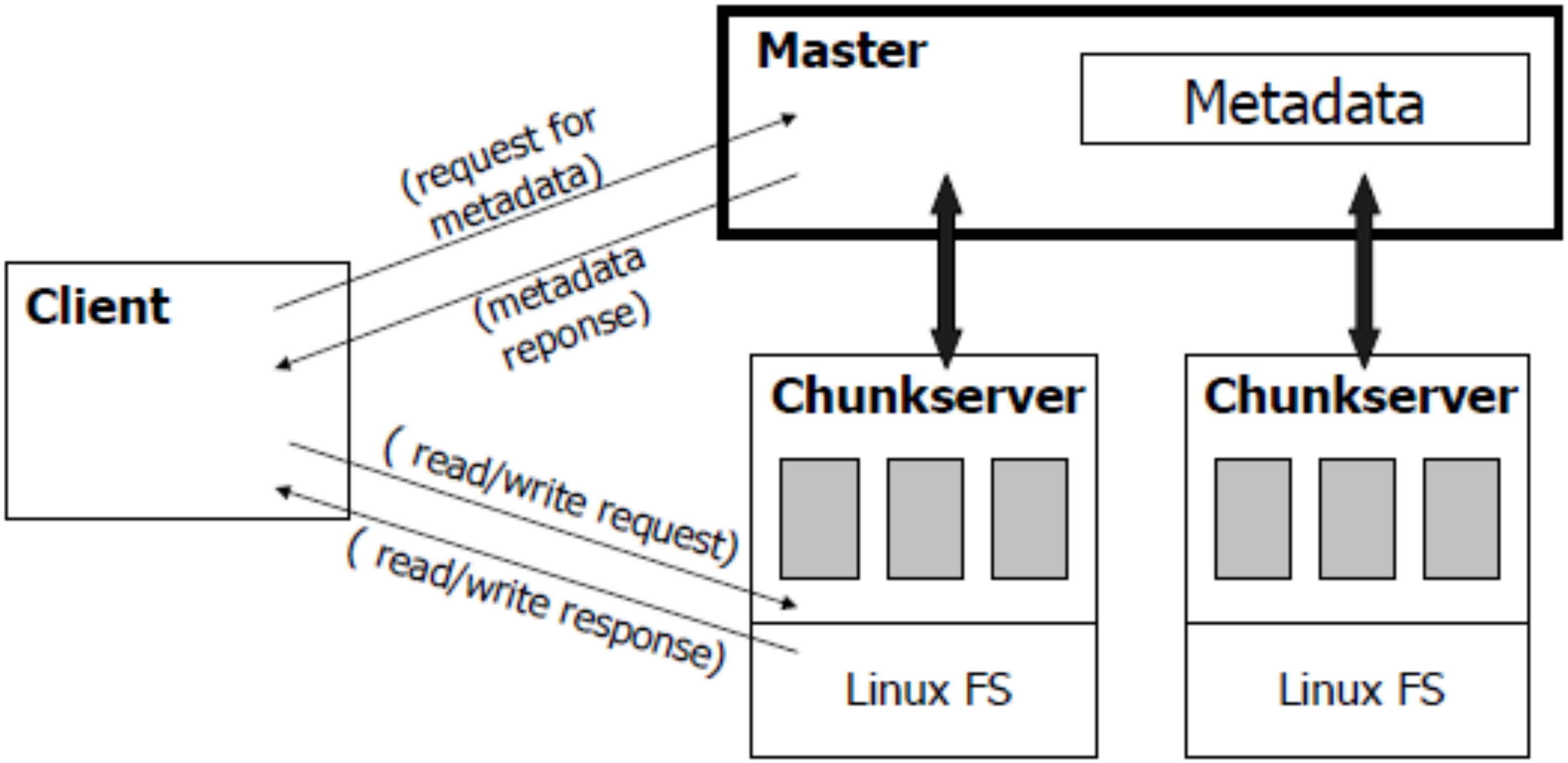
Topic Outline

- Design Motivations
- Architecture
- Read/Write/Record Append
- Fault-Tolerance
- Performance Results

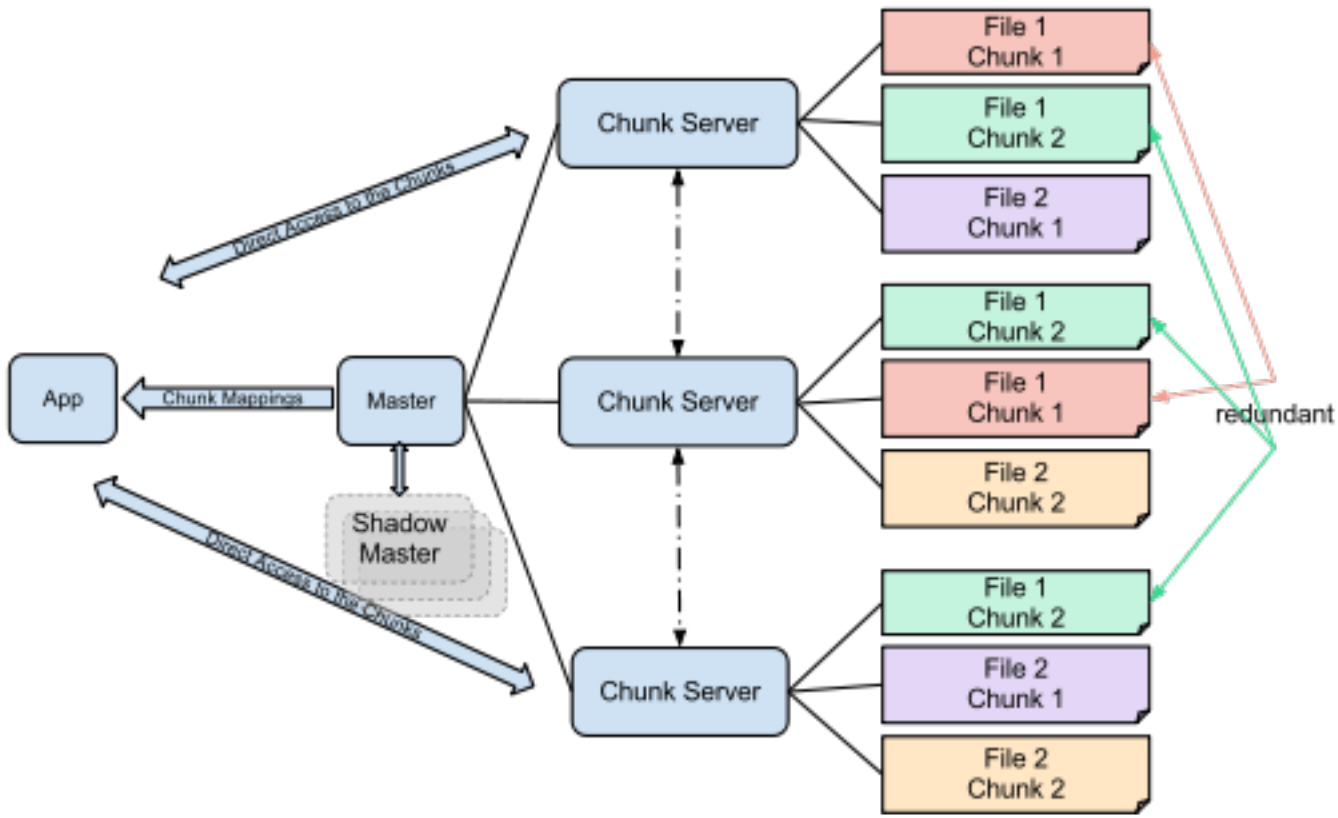
Design Motivations

1. **Fault-tolerance and auto-recovery** needs to be built into the system.
   – Thousands of commodity components
2. Standard **I/O assumptions** (e.g. block size) have to be **re-examined**.
   – Files are huge – 100MB typical, multi-GB common.
3. **Record appends** are the prevalent form of writing.
   – Dominant access pattern is streaming read; append, overwrite of existing data supported, but not optimized.
4. Google **applications** and **GFS** should be **co-designed**.
   – Relax concurrency model to simplify file system.

GFS Architecture



Files are divided into fixed-size “chunks”



Chunks are stored redundantly across Chunk Servers

GFS Architecture

What is a chunk?

- Analogous to a block, except *larger*.
- Size: 64 MB! *Note: 4K is typical block size.*
- Stored on *chunkserver* as a file.
- *Chunk handle* (~ chunk file name) is used to reference a chunk.
- Chunk replicated across multiple *chunkservers*
- *Note: There are hundreds of chunkservers in a GFS cluster distributed over multiple racks.*

GFS Architecture

What is a master?

- A *single* process running on a separate machine.
- Stores all metadata:
  - File namespace
  - File to chunk mappings
  - Chunk location information
  - Access control information
  - Chunk version numbers
  - Etc.

GFS Architecture

Master <-> Chunkserver Communication:

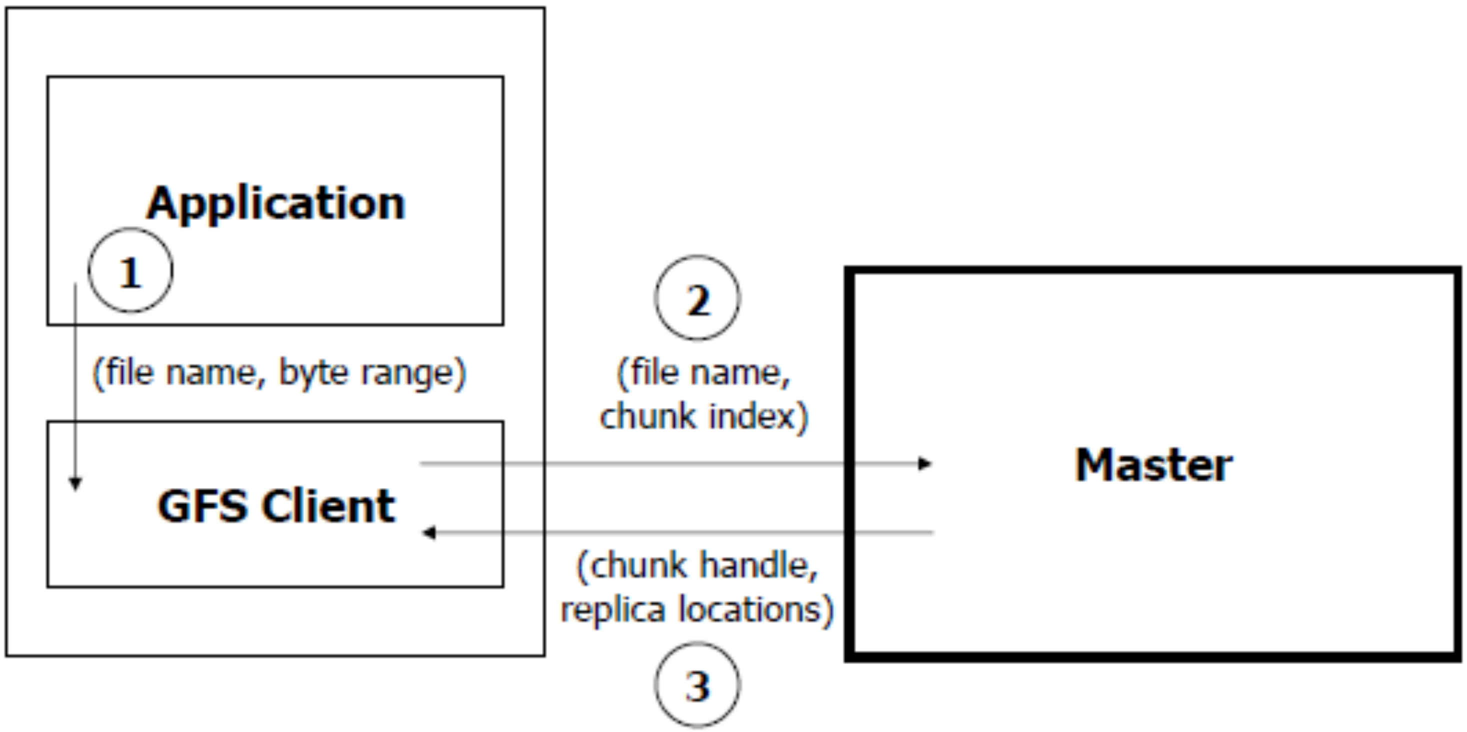
- Master and chunkserver communicate regularly (monitor) to obtain chunkserver state:
  - Ping/keep alive from chunkserver to master: Is chunkserver down?
  - Are there disk failures on chunkserver?
  - Are any replicas corrupted?
  - Which chunk replicas does chunkserver store?
- Master sends instructions to chunkserver:
  - Delete existing chunk.
  - Create new chunk.

GFS Architecture

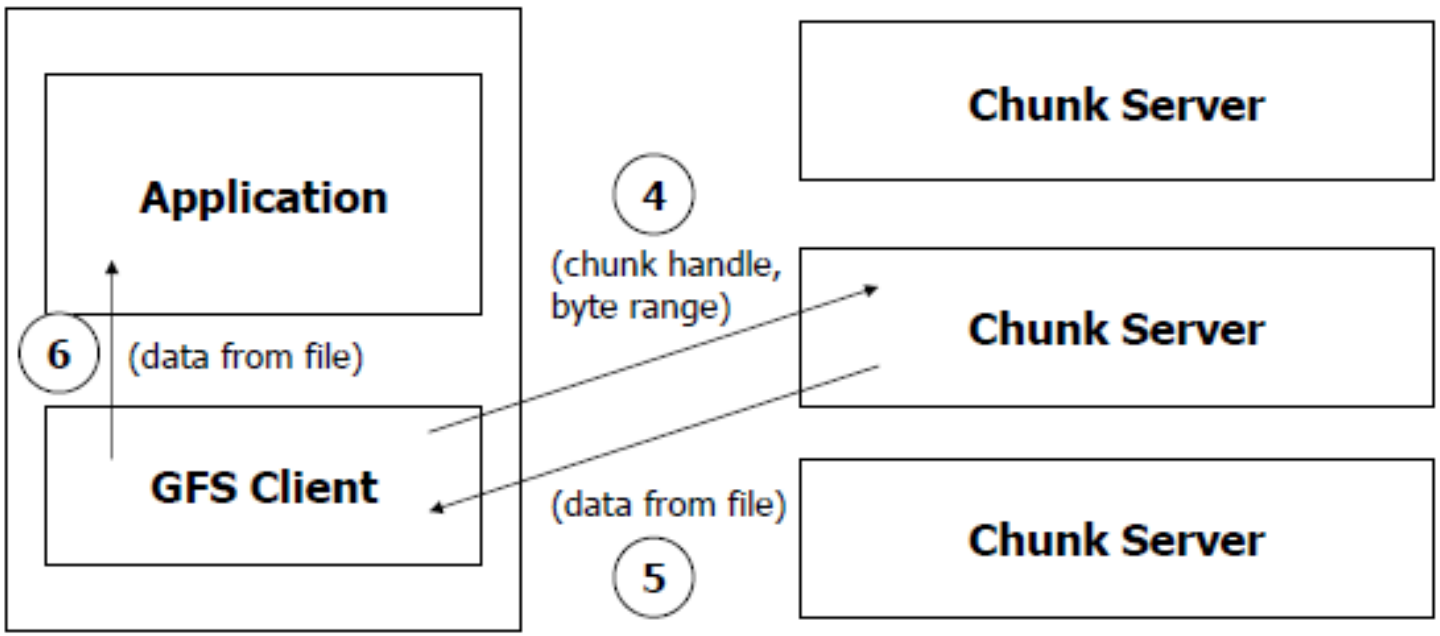
Serving Requests:

- Client sends the master a request containing the *file name* and *chunk index*.
- Client retrieves *metadata* for operation from master.
- Read/Write data flows between *client* and *chunkserver*.
- Single master is not bottleneck, because its involvement with read/write operations is minimized.

Read Algorithm - 1



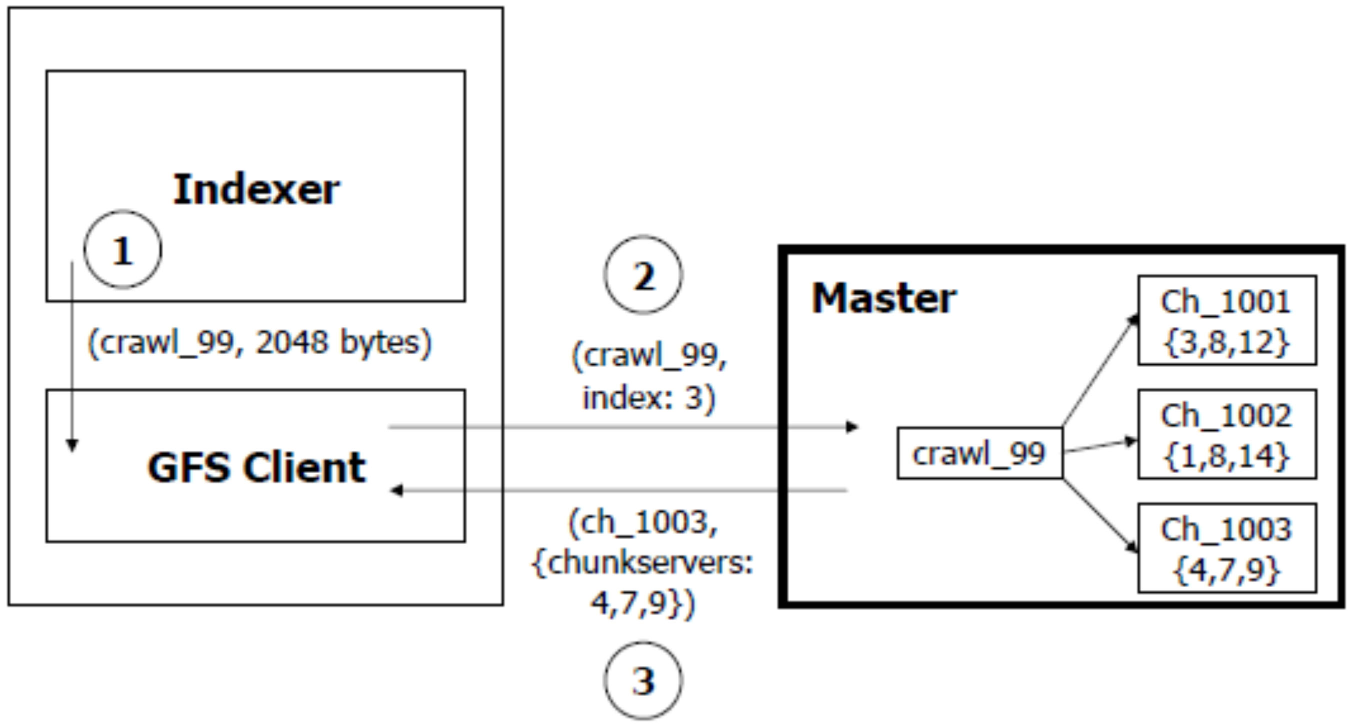
Read Algorithm - 2



Read Algorithm Steps

1. Application originates the read request.
2. GFS client translates the request from (filename, byte range) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and replica locations (i.e. chunkservers where the replicas are stored).
4. Client picks a location (usually the closest) and sends the (chunk handle, byte range) request to that location.
5. Chunkserver sends requested data to the client.
6. Client forwards the data to the application.

Read Algorithm – Example



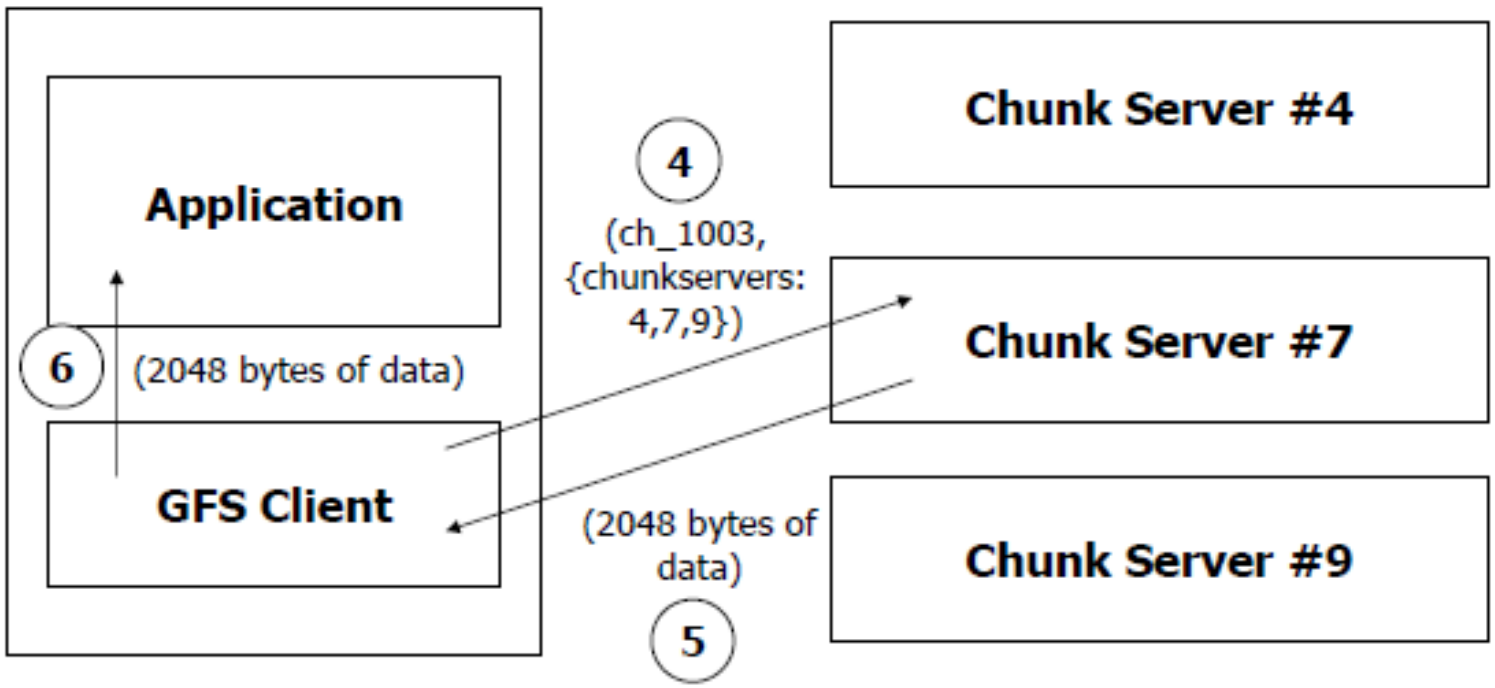
Read Algorithm – Chunk Index

Calculating chunk index from byte range:

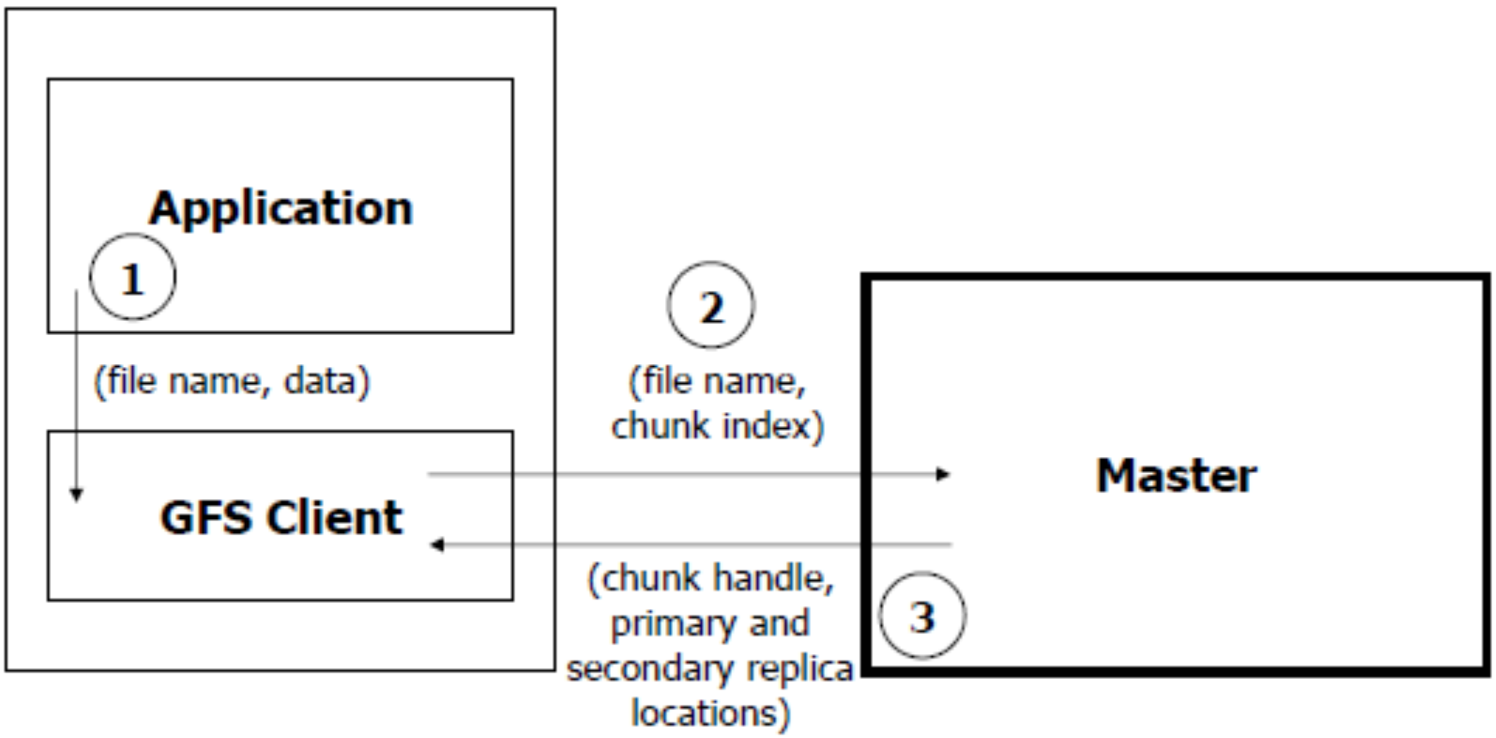
(Assumption: File position is 201,359,161 bytes)

- Chunk size = 64 MB.
- 64 MB = 1024 *1024 * 64 bytes = 67,108,864 bytes.
- 201,359,161 bytes = 67,108,864 * **3** + 32,569 bytes.
- Chunk index is 3.

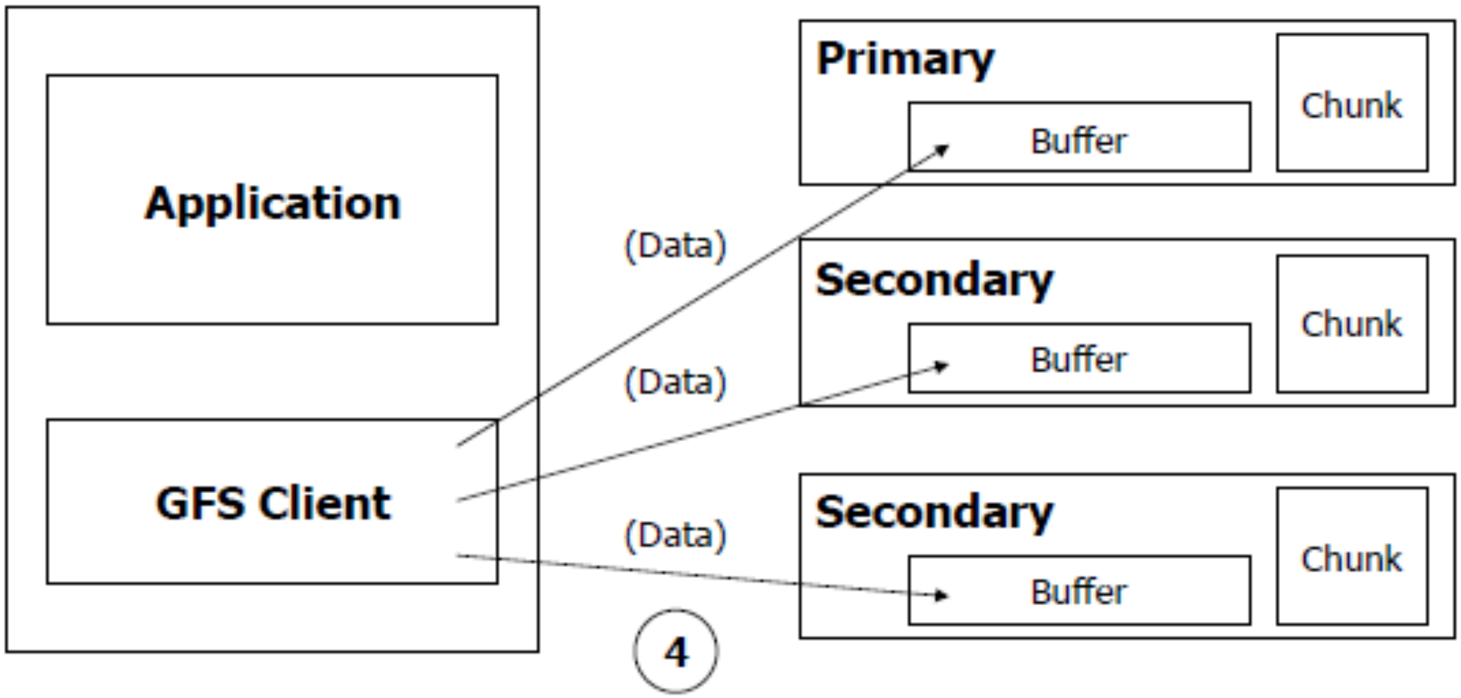
Read Algorithm – Chunk Index



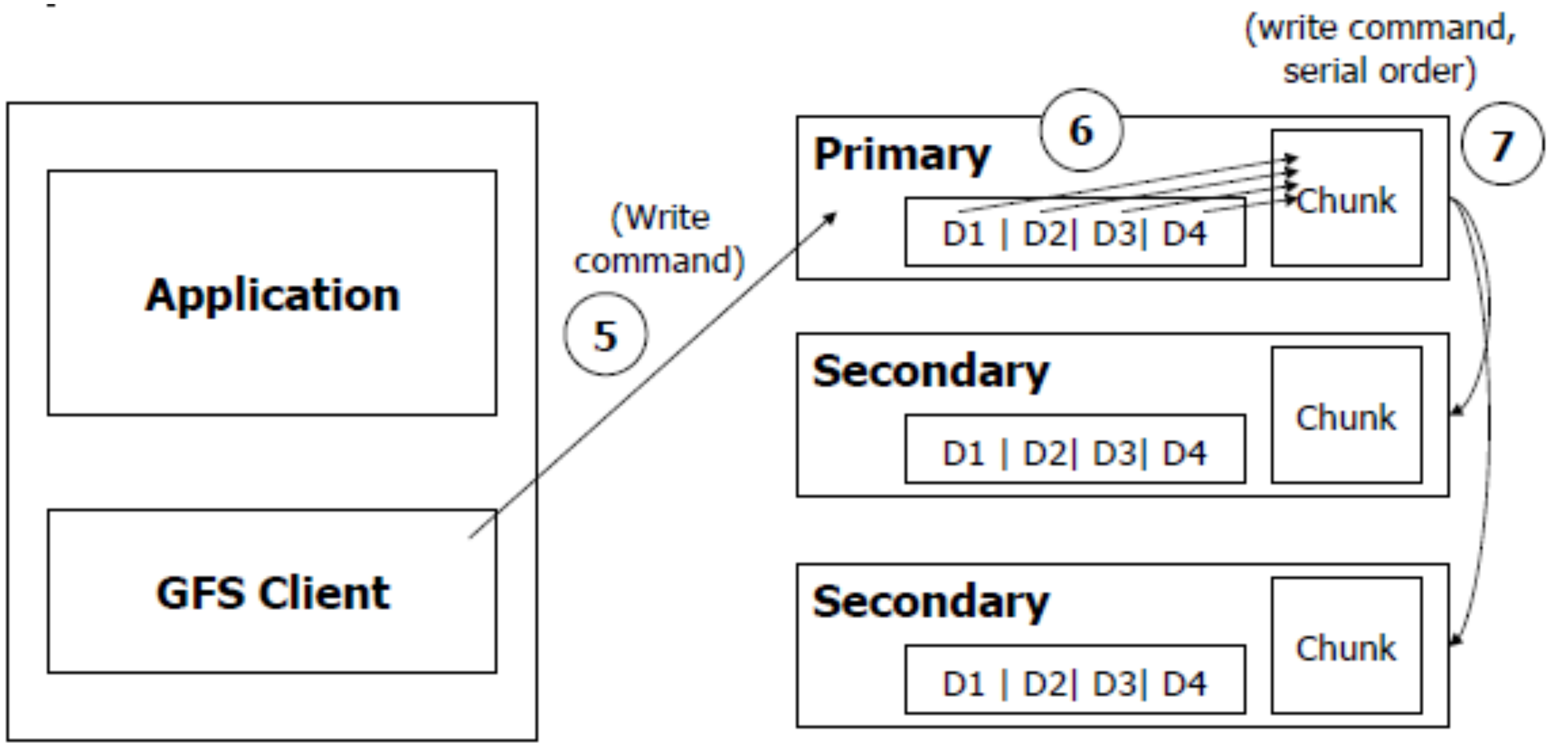
Write Algorithm - 1



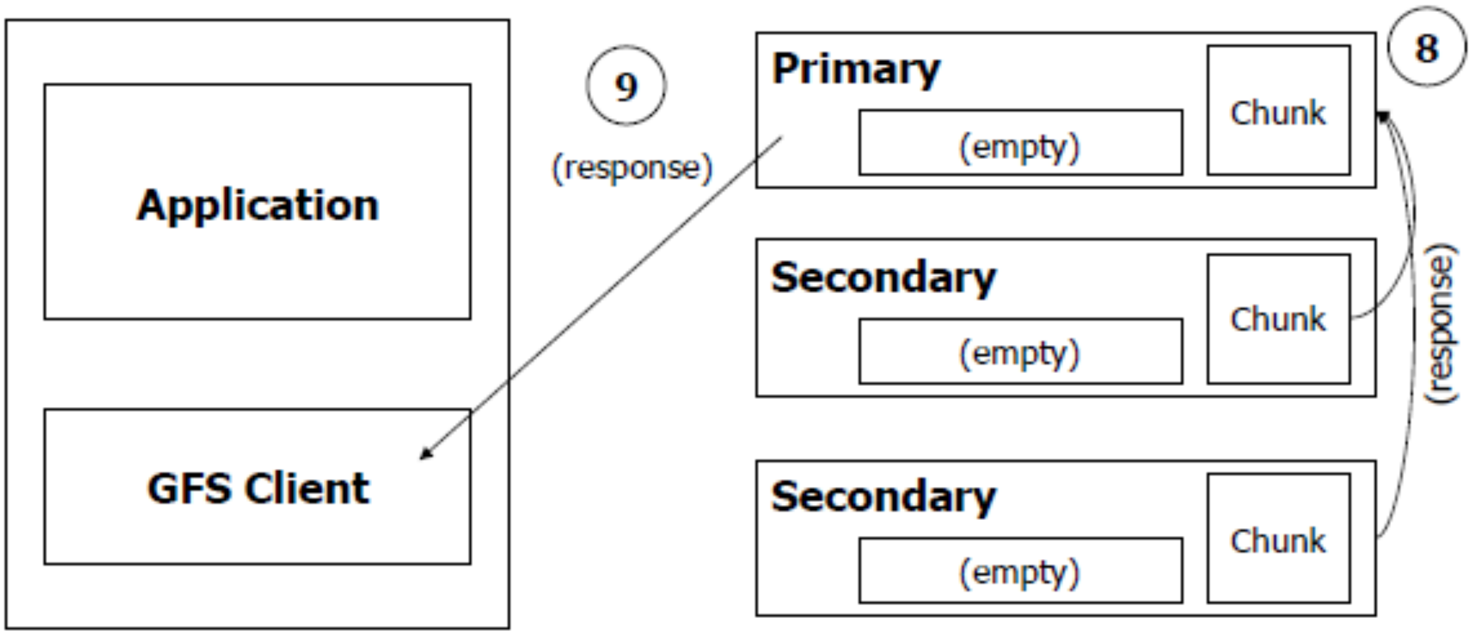
Write Algorithm - 2



Write Algorithm - 3



Write Algorithm - Steps



Write Algorithm Steps

1. Application originates write request.
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to *all* locations. Data is stored in chunkservers' internal buffers.
5. Client sends write command to primary.

Write Algorithm Steps

6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk.
7. Primary sends serial order to the secondaries and tells them to perform the write.
8. Secondaries respond to the primary.
9. Primary responds back to client.

Note: If write fails at one of chunkservers, client is informed and retries the write.

Record Append Algorithm

Important operation at Google:

- Merging results from multiple machines in one file.
- Using file as *producer-consumer* queue.

1. Application originates record append request.
2. GFS client translates request and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations.

Record Append Algorithm

5. Primary checks if record fits in specified chunk.
6. If record does not fit, then the primary:
   – pads the chunk,
   – tells secondaries to do the same, and informs the client.
   – Client then retries the append with the next chunk.
7. If record fits, then the primary:
   – appends the record,
   – tells secondaries to do the same,
   – receives responses from secondaries, and sends final response to the client.

Observations

- Clients can read in parallel.
- Clients can write in parallel.
- Clients can append records in parallel.

Fault Tolerance

- 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 Mechanisms:
  - Log of all changes made to metadata.
  - Periodic checkpoints of the log.
    - Log and checkpoints replicated on multiple machines.
    - Master state is replicated on multiple machines.
    - "Shadow" masters for reading data if "real" master is down.
- Data integrity:
  - Each chunk has an associated checksum.

Performance

- When used with relatively small number of servers (15), the file system achieves reading performance comparable to that of a single disk (80–100 MB/s), but has a reduced write performance (30 MB/s), and is relatively slow (5 MB/s) in appending data to existing files.
- The authors present no results on random seek time.
- As the master node is not directly involved in data reading, the read rate increases significantly with the number of chunk servers, achieving 583 MB/s for 342 nodes.
- Aggregating a large number of servers also allows big capacity, while it is somewhat reduced by storing data in three independent locations (to provide redundancy).

Hadoop File System (HDFS)

- Java-based distributed file system that provides scalable and reliable data storage.
- Modeled after GFS.
- Open source.
- Designed to span large clusters of commodity servers.
- HDFS has demonstrated production scalability of up to 200 PB of storage and a single cluster of 4500 servers, supporting close to a billion files and blocks.
- HDFS is a scalable, fault-tolerant, distributed storage system that works closely with a wide variety of concurrent data access applications, coordinated by YARN.

How HDFS Works

- An HDFS cluster is comprised of a NameNode, which manages the cluster metadata, and DataNodes that store the data.
- Files and directories are represented on the NameNode by inodes.
- Inodes record attributes like permissions, modification and access times, or namespace and disk space quotas.

How HDFS Works

- The file content is split into large blocks (128 Mbytes).
- Each block of the file is independently replicated at multiple DataNodes.
- The blocks are stored on the local file system on the DataNodes.
- The Namenode actively monitors the number of replicas of a block.
- When a replica of a block is lost due to a DataNode failure or disk failure, the NameNode creates another replica of the block.

How HDFS Works

- The NameNode maintains the namespace tree and the mapping of blocks to DataNodes, holding the entire namespace image in RAM.
- The NameNode does not directly send requests to DataNodes. It sends instructions to the DataNodes by replying to heartbeats sent by those DataNodes.
- The instructions include commands to:
  - replicate blocks to other nodes,
  - remove local block replicas,
  - re-register and send an immediate block report, or
  - shut down the node.

HDFS Architecture