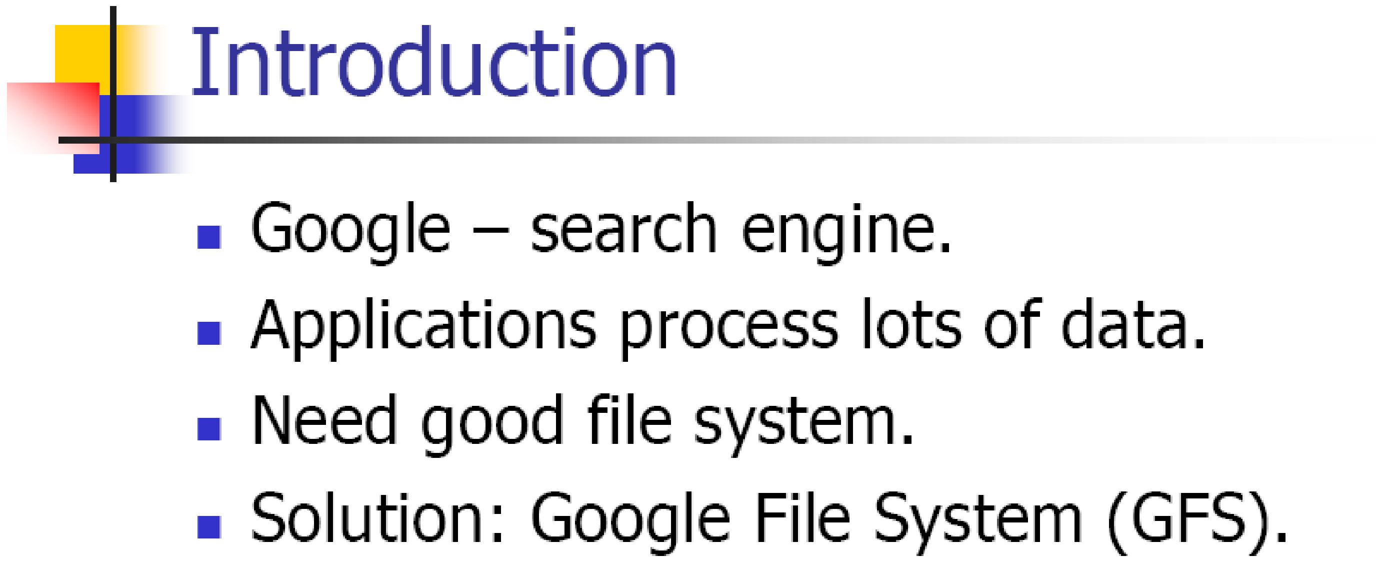
Google file System



Introduction

- Google – search engine.
- Applications process lots of data.
- Need good file system.
- Solution: Google File System (GFS).

Google file System

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

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

Google file system (GFS)

- *Google File System, a scalable distributed file system for large distributed data-intensive applications.*
- *Google File System (GFS) to meet the rapidly growing demands of Google's data processing needs.*
- *GFS shares many of the same goals as other distributed file systems such as performance, scalability, reliability, and availability.*
- *GFS provides a familiar file system interface.*
- *Files are organized hierarchically in directories and identified by pathnames.*
- *Support the usual operations to* ***create, delete, open, close, read, and write*** *files.*

GFS

- *Small as well as multi-GB files are common.*
- *Each file typically contains many application objects such as web documents.*
- *GFS provides an atomic append operation called record append. In a traditional write, the client specifies the offset at which data is to be written.*
- *Concurrent writes to the same region are not serializable.*
- *GFS has snapshot and record append operations.*

GFS (snapshot and record append)

- *The snapshot operation makes a copy of a file or a directory.*
- *Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append.*
- *It is useful for implementing multi-way merge results.*
- *GFS consist of two kinds of reads: large streaming reads and small random reads.*
- *In large streaming reads, individual operations typically read hundreds of KBs, more commonly 1 MB or more.*
- *A small random read typically reads a few KBs at some arbitrary offset.*

Common Goals of GFS and most Distributed File Systems

- *Performance*
- *Reliability*
- *Scalability*
- *Availability*

Other GFS Concepts

- **Component failures are the norm rather than the exception.**
  - *File System consists of hundreds or even thousands of storage machines built from inexpensive commodity parts.*
- **Files are Huge. Multi-GB Files are common**.
  - *Each file typically contains many application objects such as web documents.*
- **Append, Append, Append**.
  - *Most files are mutated by appending new data rather than overwriting*

Other GFS Concepts

- ***Why assume hardware failure is the norm?***
  - *It is cheaper to assume common failure on poor hardware and account for it, rather than invest in expensive hardware and still experience occasional failure.*
  - *The amount of layers in a distributed system (network, disk, memory, physical connections, power, OS, application) mean failure on any could contribute to data corruption.*

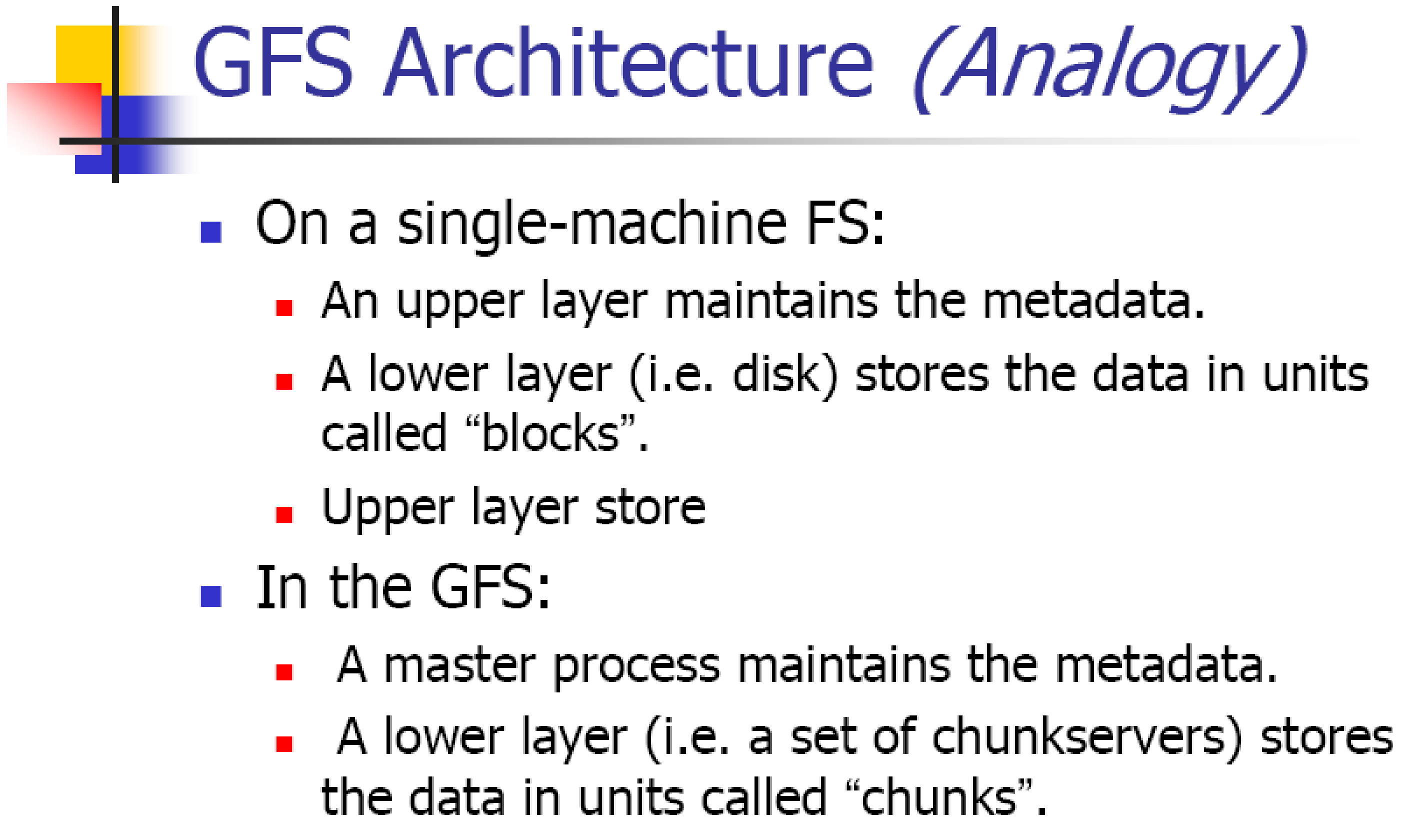
GFS Assumptions

- **System built from inexpensive commodity components that fail**
  - Modest number of files – expect few million and > 100MB size. Did not optimize for smaller files.
- **2 kinds of reads** – :
  - large streaming read (1MB)
  - small random reads (batch and sort)
- High sustained bandwidth chosen over low latency

GFS Interface

- *GFS – familiar file system interface*
- *Files organized hierarchically in directories, path names*
- *Create, delete, open, close, read, write operations*
- *Snapshot and record append (allows multiple clients to append simultaneously - atomic)*

GFS Architecture

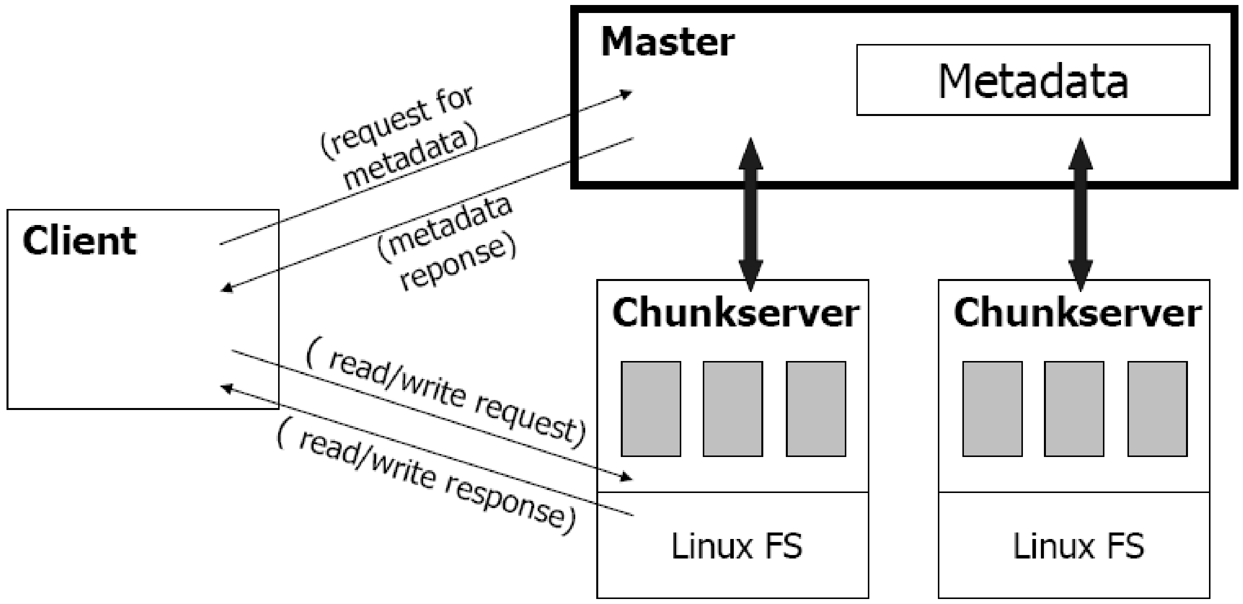


GFS Architecture *(Analogy)*

- On a single-machine FS:
  - An upper layer maintains the metadata.
  - A lower layer (i.e. disk) stores the data in units called “blocks”.
  - Upper layer store
- 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

GFS Architecture



GFS Architecture

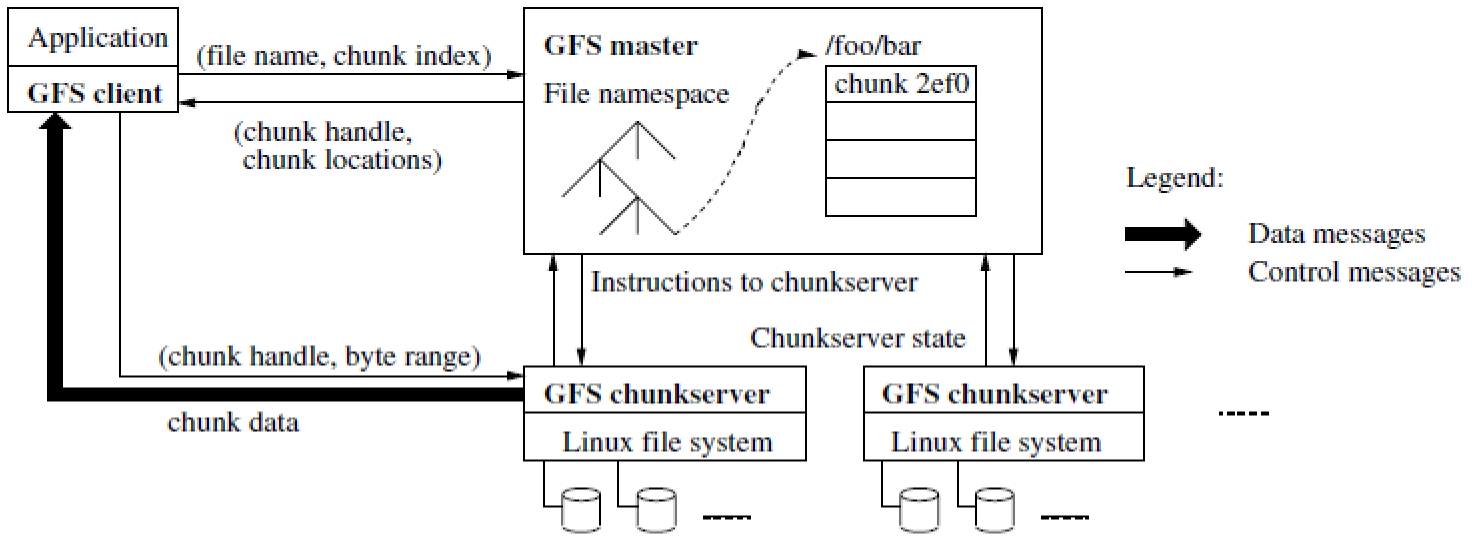
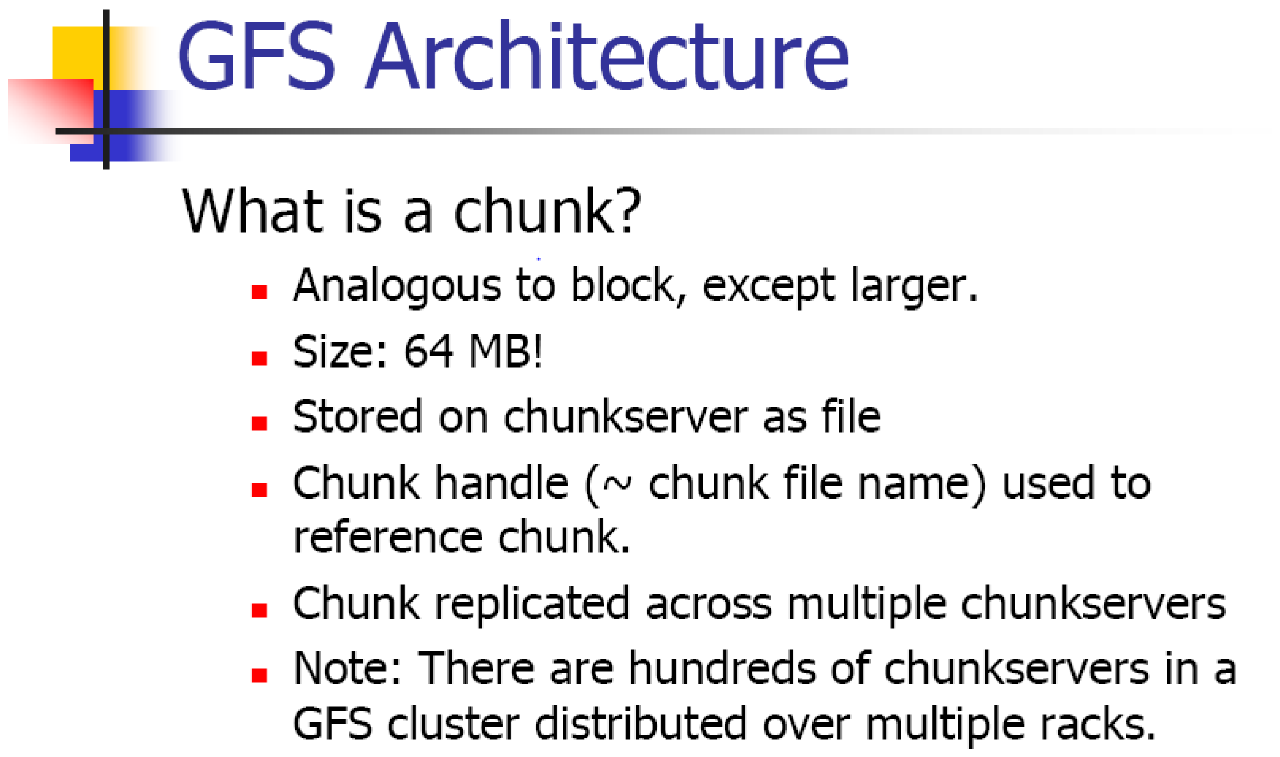


Figure 1: GFS Architecture

Chunk



GFS Architecture

What is a chunk?

- Analogous to block, except larger.
- Size: 64 MB!
- Stored on chunkserver as file
- Chunk handle (~ chunk file name) used to reference 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

- *A GFS cluster consists of a single master and multiple chunk-servers and is accessed by multiple clients, Each of these is typically a commodity Linux machine.*

- *It is easy to run both a chunk-server and a client on the same machine.*

- *As long as machine resources permit, it is possible to run flaky application code is acceptable.*

GFS Architecture

- Files are divided into fixed-size *chunks.*
- *Each chunk is* identified by an immutable and globally unique 64 bit *chunk assigned by the master at the time of chunk creation.*
- Chunk-servers store chunks on local disks as Linux files, each chunk is replicated on multiple chunk-servers.
- The master maintains all file system metadata. This includes the namespace, access control

GFS Architecture

Master <-> Chunkserver Communication:

- Master and chunkserver communicate regularly to obtain state:
  - 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.

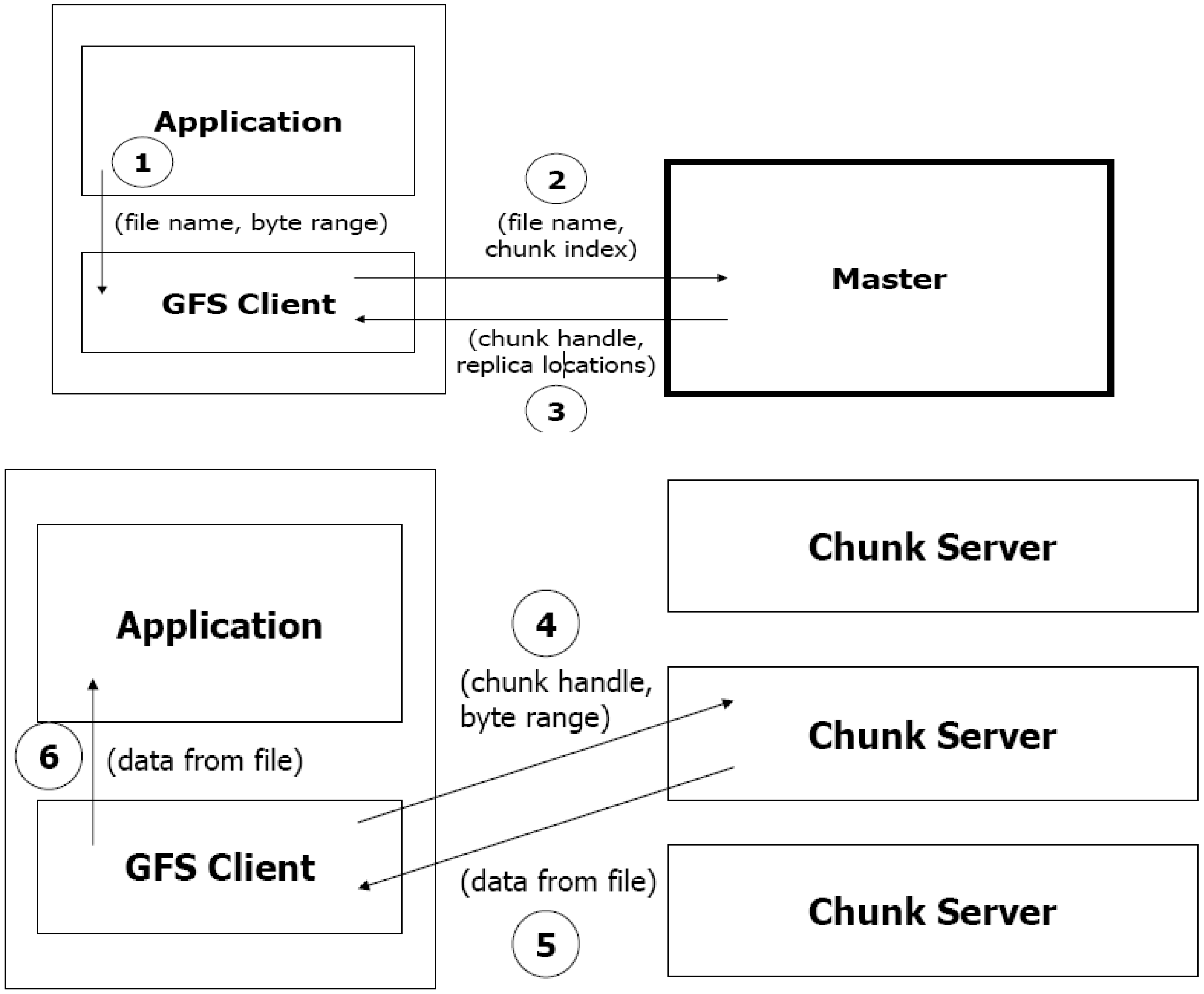
GFS Architecture

Serving Requests:

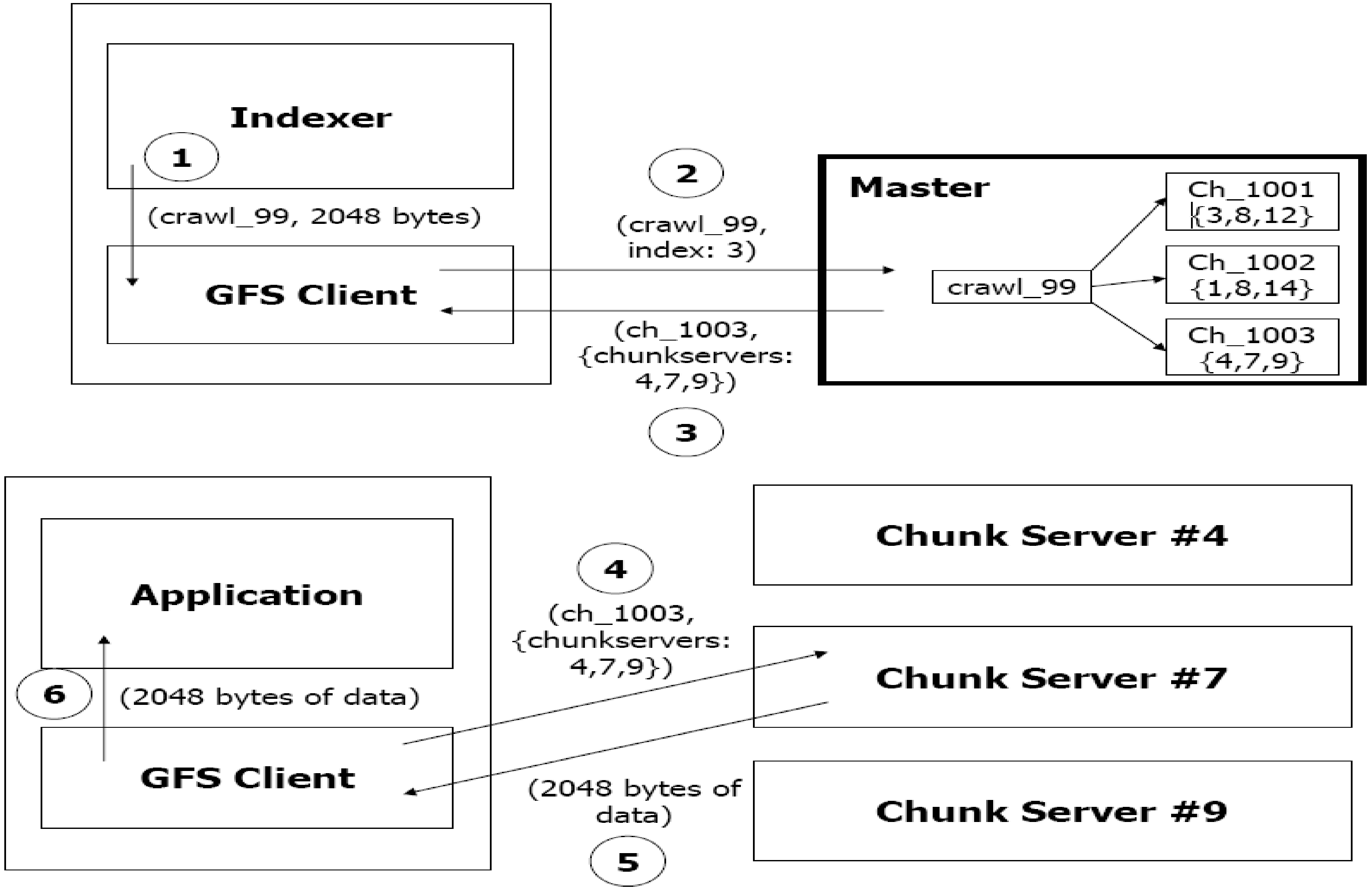
- Client retrieves metadata for operation from master.
- Read/Write data flows between client and chunkserver.
- Single master is not bottleneck, because its involvement with read/write operations is minimized.

Read Algorithm

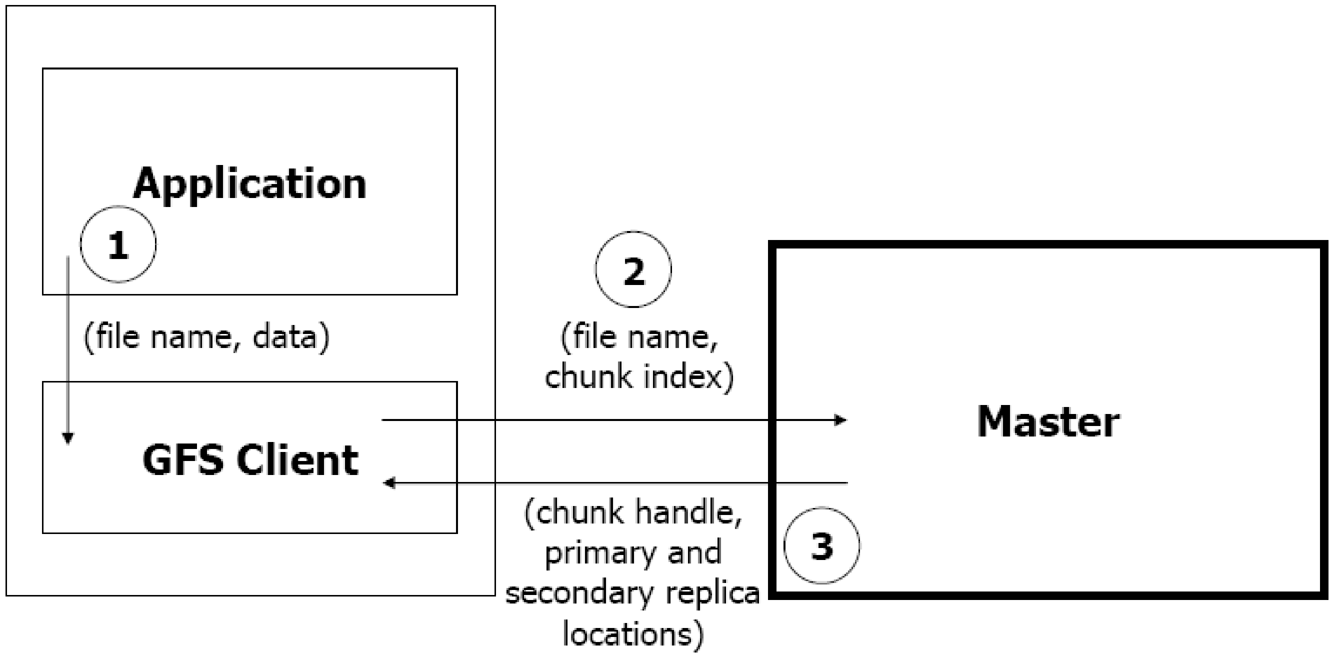
1. Application originates the read request.
2. GFS client translates the request from (filename, byte range) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and replica locations (i.e. chunkservers where the replicas are stored).
4. Client picks a location and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


Application
1
(file name, byte range)
GFS Client
2
(file name, chunk index)
Master
(chunk handle, replica locations)
3
Application
6
(data from file)
GFS Client
4
(chunk handle, byte range)
Chunk Server
Chunk Server
Chunk Server
(data from file)
5

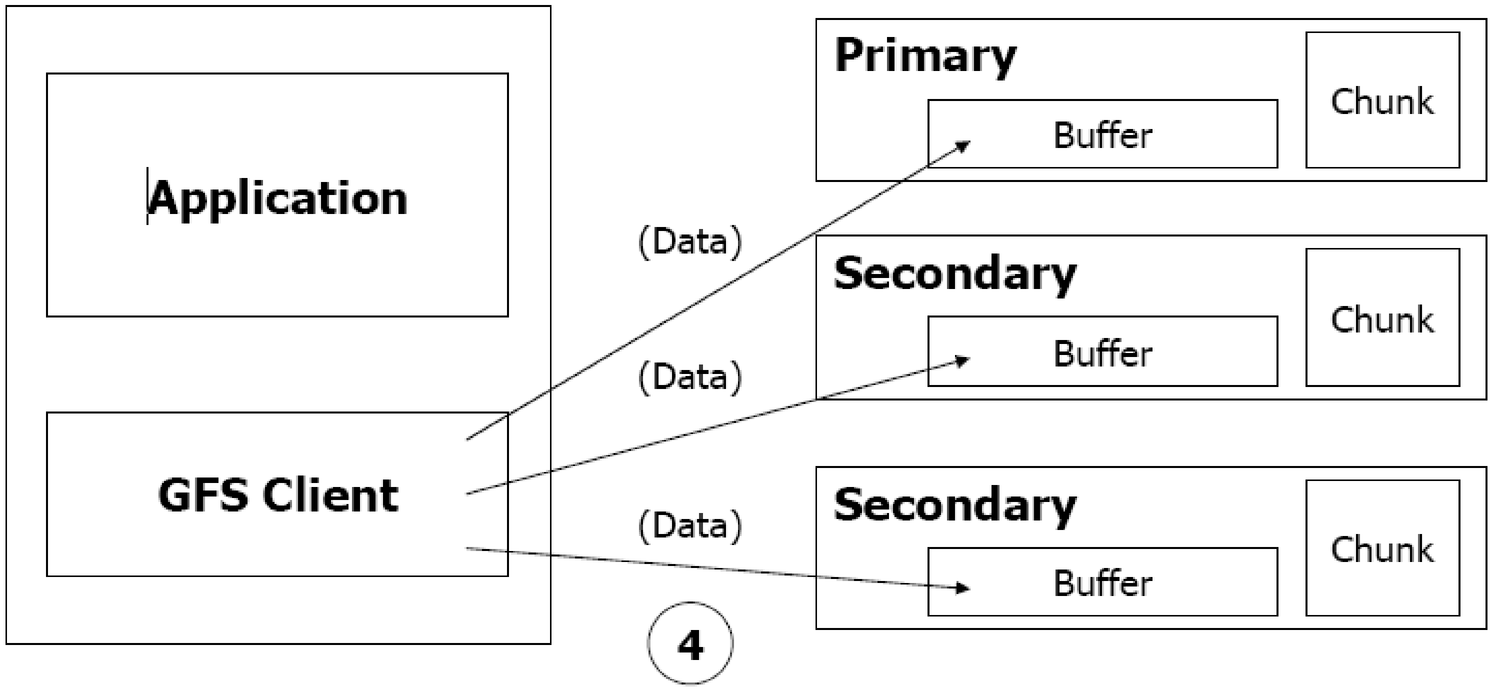
Read Algorithm (*Example*)



Write Algorithm



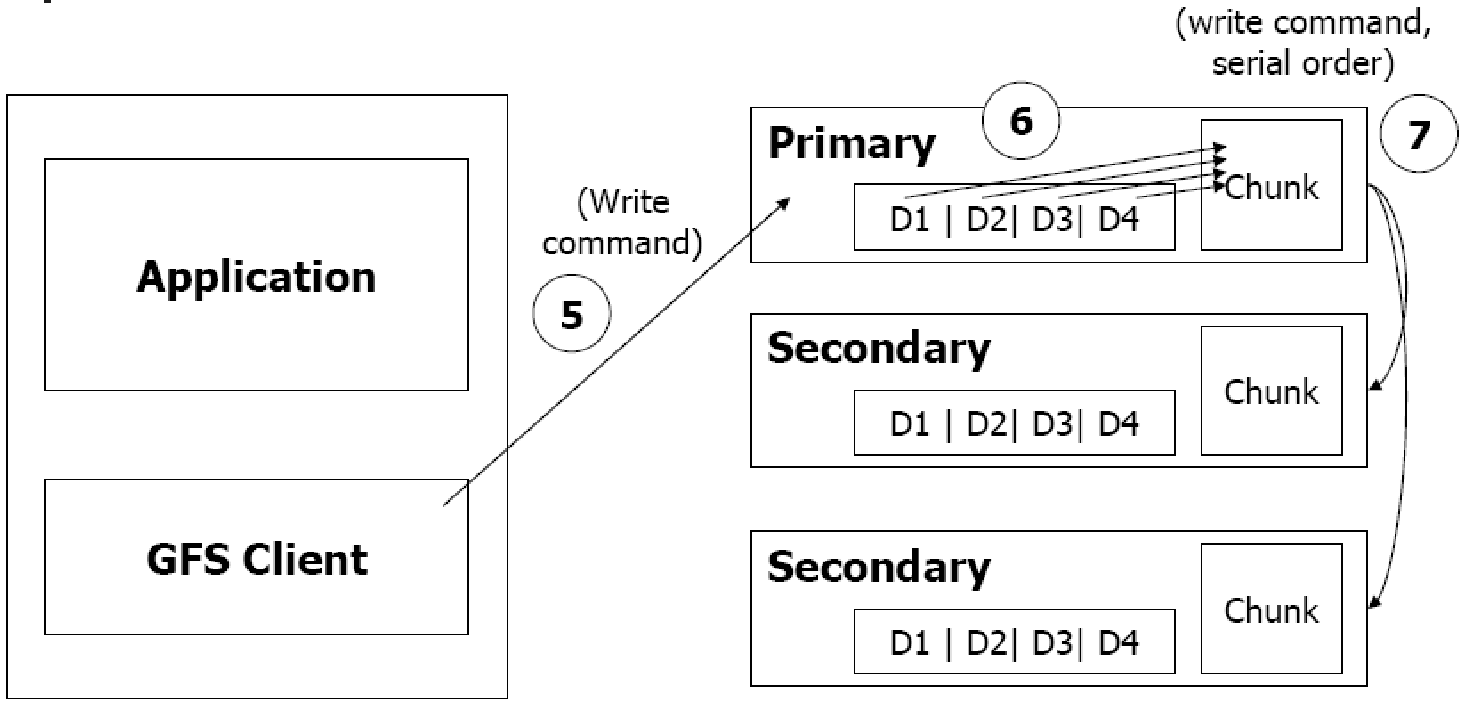
Write Algorithm



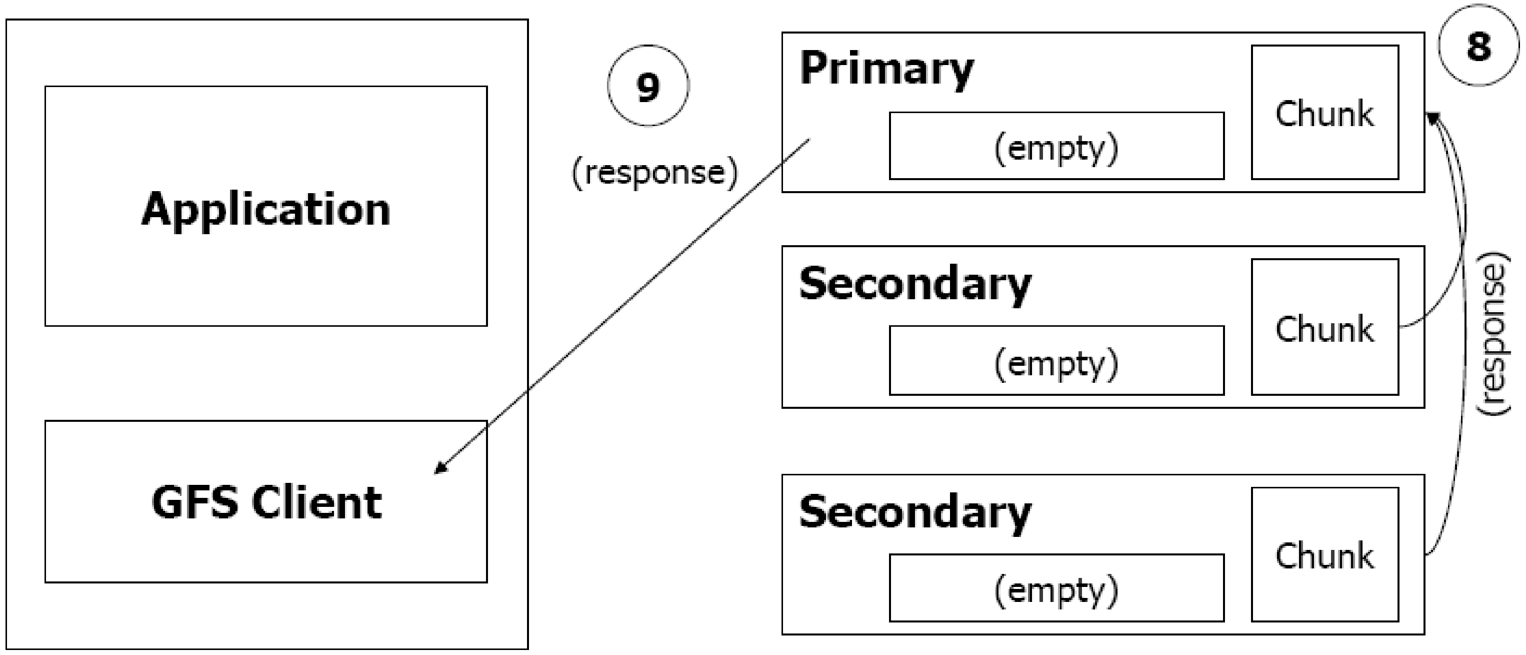
Write Algorithm

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



Write Algorithm



Write Algorithm

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.

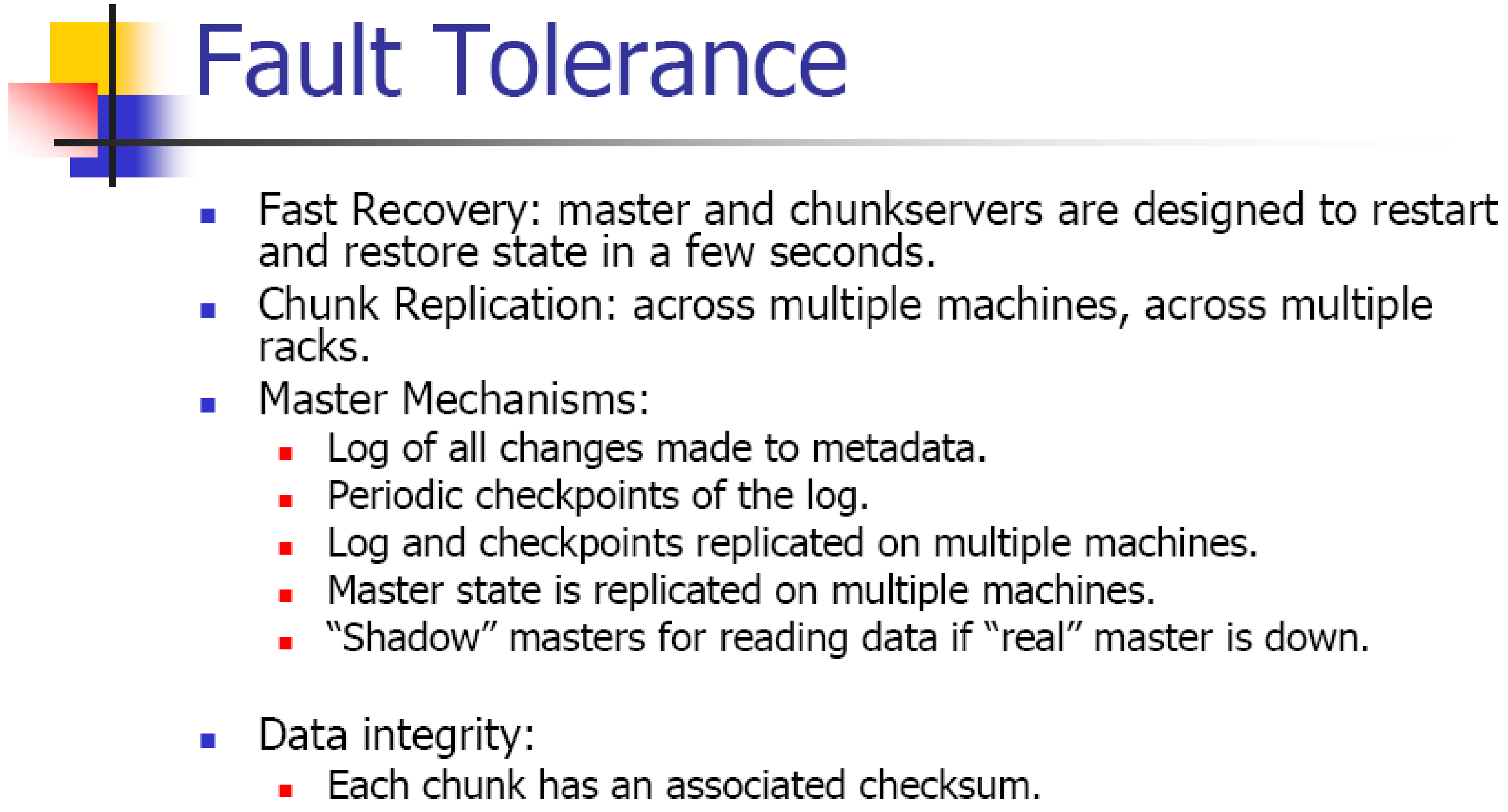
Record Append Algorithm

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.

Record append

- Client pushes data to all replicas of last chunk of the file
- Sends request to primary
- Primary checks if will exceed 64MB, if so send message to retry on next chunk, else primary appends, tells 2ndary to write, send reply to client
- If append fails, retries
  - Replicas of same chunk may contain different data, *including duplication of same record in part or whole*
  - Does not guarantee all replicas bytewise identical
  - Guarantees **data written at least once as atomic unit**
  - Data must have been written at same offset on all replicas of same chunk, but not same offset in file

GFS is fault tolerance?



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.

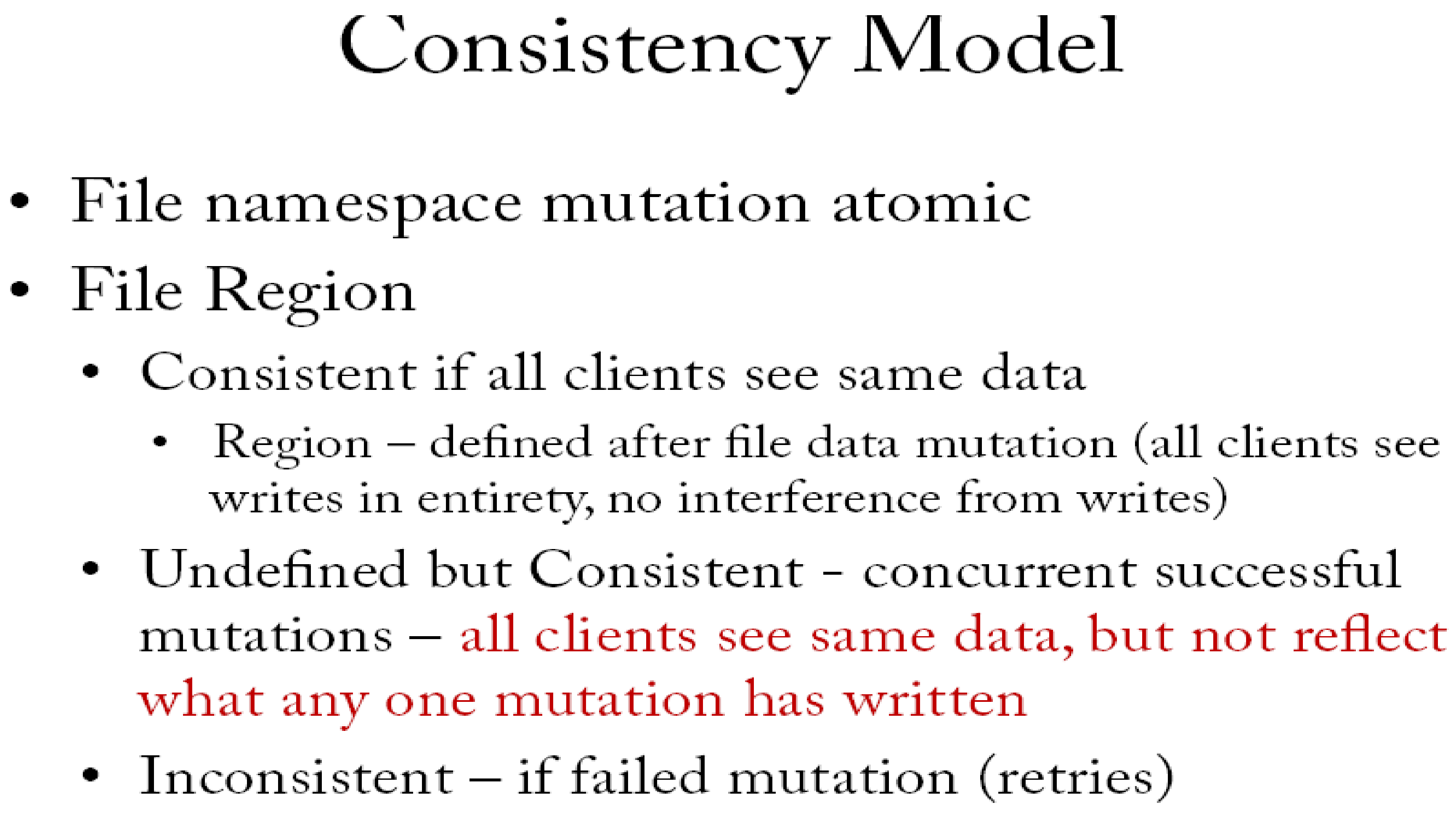
Metadata

- 3 types:
  - File and chunk namespaces
  - Mapping from files to chunks
  - Location of each chunk’ s replicas

Metadata

- Instead of keeping track of chunk location info
  - Poll – which chunkserver has which replica
  - Master controls all chunk placement
  - Disks may go bad, chunkserver errors, etc.

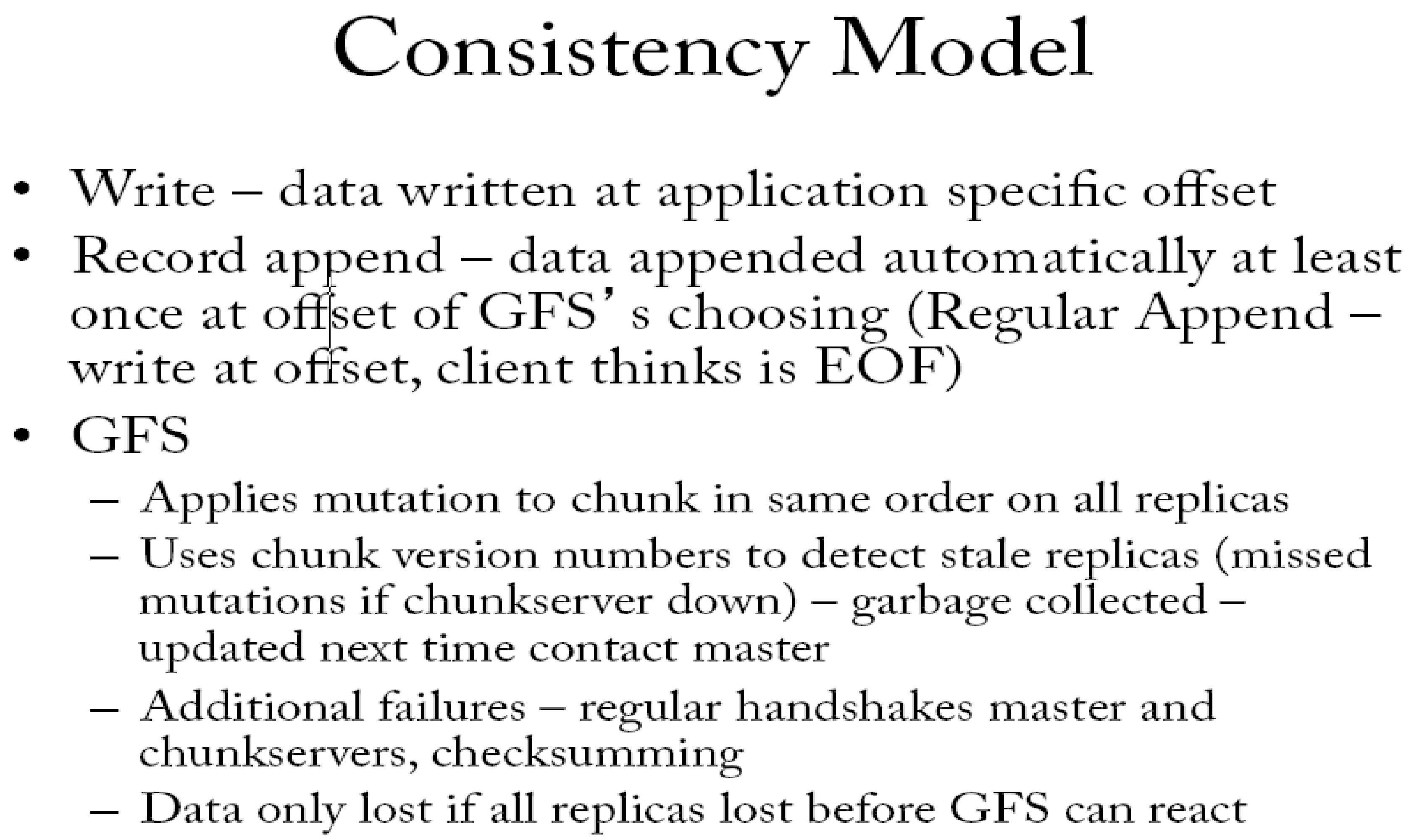
Consistency



Consistency Model

- File namespace mutation atomic
- File Region
  - Consistent if all clients see same data
    - Region – defined after file data mutation (all clients see writes in entirety, no interference from writes)
  - Undefined but Consistent - concurrent successful mutations – all clients see same data, but not reflect what any one mutation has written
  - Inconsistent – if failed mutation (retries)

Consistency



Consistency Model

- Write – data written at application specific offset
- Record append – data appended automatically at least once at offset of GFS's choosing (Regular Append – write at offset, client thinks is EOF)
- GFS
  - Applies mutation to chunk in same order on all replicas
  - Uses chunk version numbers to detect stale replicas (missed mutations if chunkserver down) – garbage collected – updated next time contact master
  - Additional failures – regular handshakes master and chunkservers, checksumming
  - Data only lost if all replicas lost before GFS can react

Consistency

- Relaxed consistency can be accommodated – relying on appends instead of overwrites
- Appending more efficient/resilient to failure than random writes
- Checkpointing allows restart incrementally and no processing of incomplete successfully written data

Write control and data flow

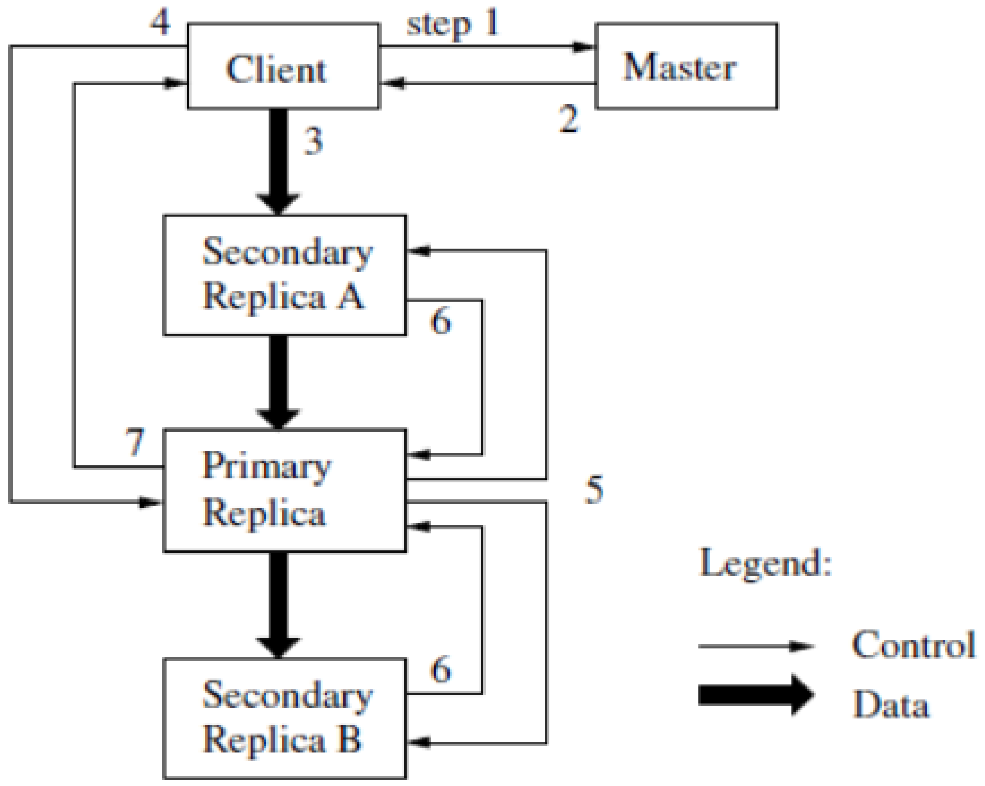
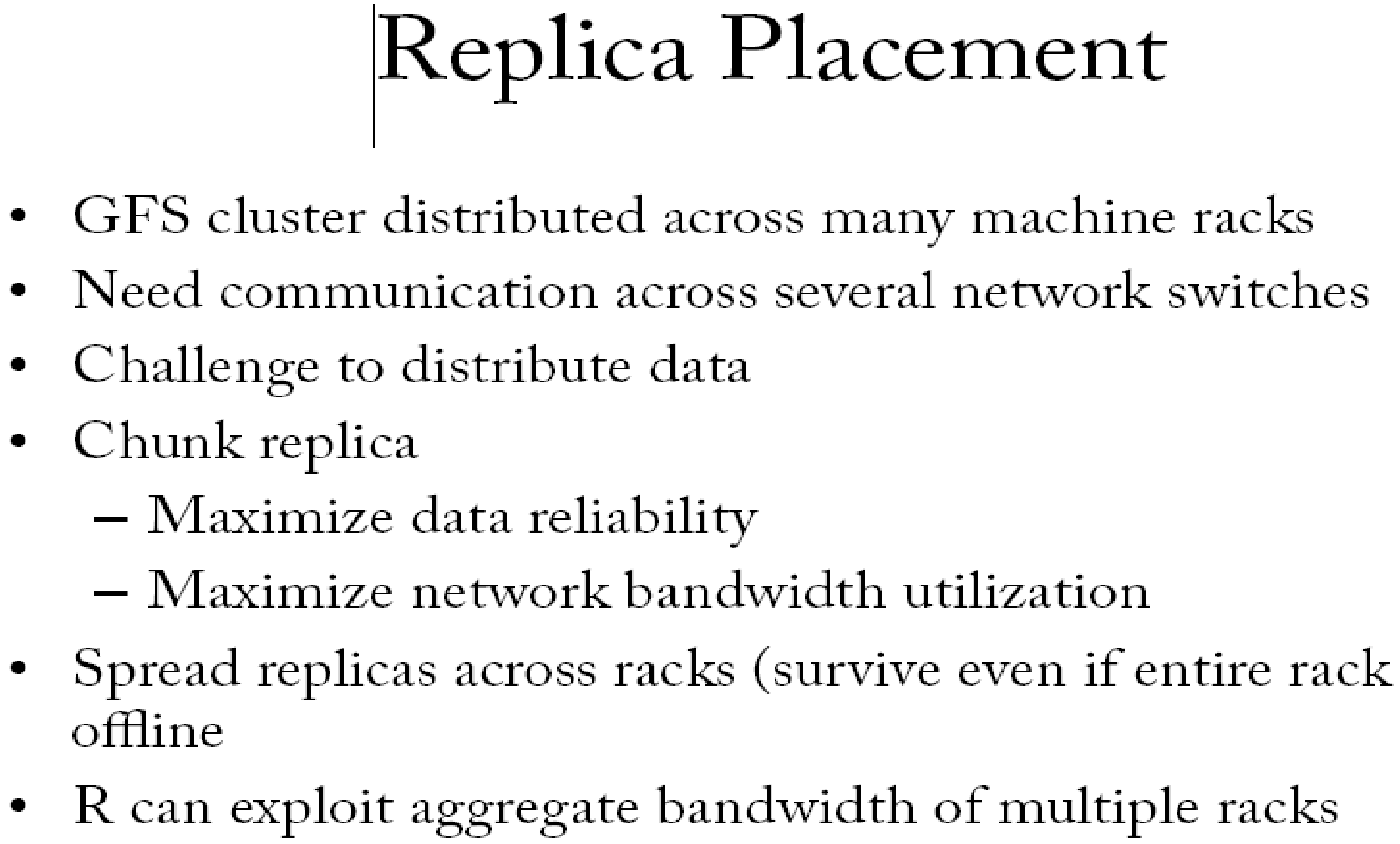


Figure 2: Write Control and Data Flow

Replica placement



Replica Placement

- GFS cluster distributed across many machine racks
- Need communication across several network switches
- Challenge to distribute data
- Chunk replica
  - Maximize data reliability
  - Maximize network bandwidth utilization
- Spread replicas across racks (survive even if entire rack offline
- R can exploit aggregate bandwidth of multiple racks

Replica placement

- Re-replicate
  - When number of replicas falls below goal
    - Chunkserver unavailable, corrupted, etc.
    - Replicate based on priority (fewest replicas)
  - Master limits number of active clone ops
- Rebalance
  - Periodically moves replicas for better disk space and load balancing
  - Gradually fills up new chunkserver
  - Removes replicas from chunkservers with below-average free space

Garbage Collection

- When delete file, file renamed to hidden name including delete timestamp
- During regular scan of file namespace
  - hidden files removed if existed > 3 days
  - Until then can be undeleted
  - When removed, in-memory metadata erased
  - Orphaned chunks identified and erased
  - With HeartBeat message, chunkserver/master exchange info about files, master tells chunkserver about files it can delete, chunkserver free to delete

Garbage Collection

- Advantages
  - Simple, reliable in large scale distributed system
    - Chunk creation may succeed on some servers but not others
    - Replica deletion messages may be lost and resent
    - Uniform and dependable way to clean up replicas
  - Merges storage reclamation with background activities of master
    - Done in batches
    - Done only when master free
  - Delay in reclaiming storage provides against accidental deletion

Garbage Collection

- Disadvantages
  - Delay hinders user effort to fine tune usage when storage tight
  - Applications that create/delete may not be able to reuse space right away
    - Expedite storage reclamation if file explicitly deleted again
    - Allow users to apply different replication and reclamation policies

Shadow Master

- Master Replication
  - Replicated for reliability
  - One master remains in charge of all mutations and background activities
  - If fails, start instantly
  - If machine or disk mails, monitor outside GFS starts new master with replicated log
  - Clients only use canonical name of master

Shadow Master

- Shadow Master
  - Read-only access to file systems even when primary master dow
  - Not mirrors, so may lag primary slightly (fractions of second)
  - Enhance read availability for files not actively mutated or if sta
    OK, e.g. metadata, access control info ???
  - Shadow master read replica of operation log, applies same sequence of changes to data structures as the primary does
  - Polls chunkserver at startup,monitors their status, etc.
  - Depends only on primary for replica location updates

Data Integrity

- Checksumming to detect corruption of stored data
- Impractical to compare replicas across chunkservers to detect corruption
- Divergent replicas may be legal
- Chunk divided into 64KB blocks, each with 32 bit checksums
- Checksums stored in memory and persistently with logging

Data Integrity

- Before read, checksum
- If problem, return error to requestor and reports to master
- Requestor reads from replica, master clones chunk from other replica, delete bad replica
- Most reads span multiple blocks, checksum small part of it
- Checksum lookups done without I/O

Real World Clusters

- Cluster A – research development by 100 engineers
- Cluster B – production data processing
  - Multi TB data sets, longer processing

- Cluster B has more dead files, more chunks, larger files

Conclusions

- GFS – qualities essential for large-scale data processing on commodity hardware
- Component failures the norm rather than exception
- Optimize for huge files appended to
- Fault tolerance by constant monitoring, replication, fast/ automatic recovery
- High aggregate throughput
  - Separate file system control
  - Large file size
- Google currently has multiple GFS clusters deployed for different purposes.
- The largest currently implemented systems have over 1000 storage nodes and over 300 TB of disk storage.