

The Google File System (GFS)



Rajiv Misra

Dept. of Computer Science &
Engineering

Indian Institute of Technology Patna

rajivm@iitp.ac.in

Introduction: File System

- File systems determine **how** data is stored and retrieved
- **Distributed file systems** manage the storage across a network of machines
 - Added complexity due to the network
- **GFS/HDFS** are distributed file systems

Introduction: 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.

Introduction: GFS

- **GFS is a scalable distributed file system** for large data intensive applications
- Shares many of the same goals as previous distributed file systems such as **performance, scalability, reliability, and availability.**
- The design of GFS is driven by **four key observations**
 - **Component failures, huge files, mutation of files, and benefits of co-designing the applications and file system API**

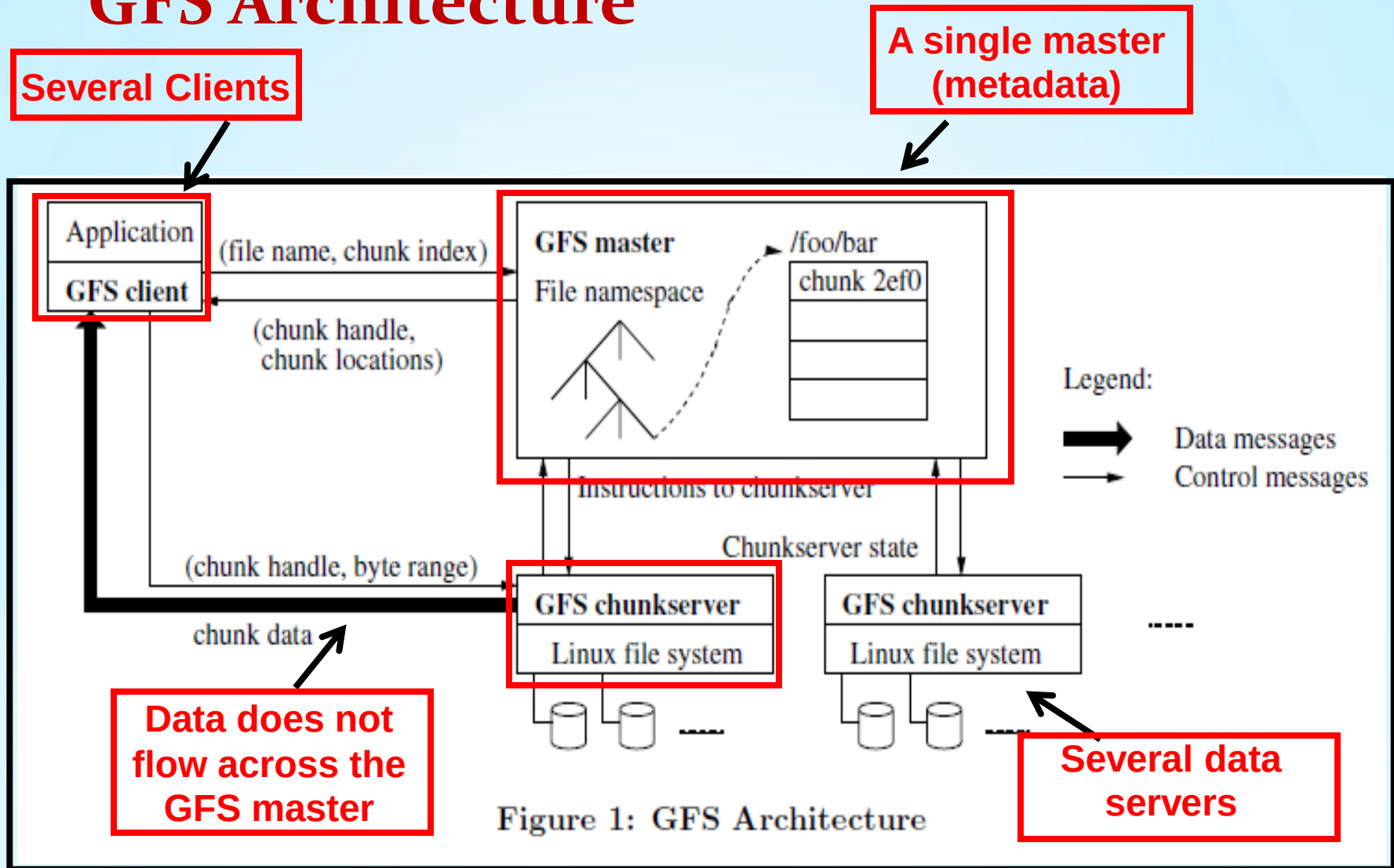
GFS Assumptions

- **Hardware failures are common**
 - System is built from many inexpensive commodity components
- **Modest number of huge files**
 - A few million files, each typically 100MB or larger (Multi-GB files are common)
 - No need to optimize for small files
- **Workloads : two kinds of reads, and writes**
 - Large streaming reads (1MB or more) and small random reads (a few KBs)
 - Small random reads
 - Sequential appends to files by hundreds of data producers
- **High sustained throughput is more important than latency**
 - Response time for individual read and write is not critical

GFS Design Overview

- **Single Master**
 - Centralized management
- **Files stored as chunks**
 - With a fixed size of 64MB each.
- **Reliability through replication**
 - Each chunk is replicated across 3 or more chunk servers
- **Data caching**
 - Due to large size of data sets
- **Interface**
 - Suitable to Google apps
 - Create, delete, open, close, read, write, snapshot, record append

GFS Architecture



GFS: Master

The Role of the Master

Metadata Server : Maintain all file system metadata

- File namespace
- File to chunk mapping
- Chunk location information
- Keep all the metadata in the master's memory (FAST)

Location of chunks and their replicas:

- Master does not keep a persistent record (no disk I/O)
- Operations log for persistence and recovery
- Poll it from chunkservers (master as monitor, no extra cost)

Master as metadata server

Monitor

- Heartbeat messages to detect the state of each chunkserver
- Communicate with chunkservers periodically

Centralized Controller

- System-wide activities
- Manage chunk replication in the whole system

One Master

- **Single master**

- Simplify the design of the system
- Control chunk placement using global knowledge
- Potential bottleneck?

- **Avoiding bottleneck**

- Clients do no read/write data through master
- Clients cache metadata (e.g., chunk handles, locations)
- Client typically asks for multiple chunks in the same request
- Master can also include the information for chunks immediately following those requested

Caching

- **Metadata**

- Cached on client after being retrieved from the Master
- Only kept for a period of time to prevent stale data

- **File caching**

- Clients never cache file data
- Chunkservers never cache file data (Linux's buffer will for local files)
- File working sets too large
- Simplifies cache coherence

GFS: Chunks

Chunks

- 1 chunk = 64MB or 128MB (can be changed); chunk stored as a **plain Linux file** on a chunk server
- Advantages of large (but not too large) chunk size
 - **Reduced** need for client/master interaction
 - 1 request per chunk suits the target workloads
 - Client can **cache all the chunk locations** for a multi-TB working set
 - **Reduced size of metadata** on the master (kept in memory)
- Disadvantage: chunkserver can become **hotspot** for popular file(s)

Chunk Locations

- Master does **not** keep a **persistent record** of chunk replica locations
- Master **polls** chunkservers about their chunks at startup
- Master keeps up to date through **periodic HeartBeat messages**
- Master/chunkservers easily kept in sync when chunk servers leave/join/fail/restart [regular event]
- Chunkserver has the final word over what chunks it has

Operation log

- Persistent record of critical metadata changes
- Critical to the recovery of the system
- Changes to metadata are only made visible to clients after they have been written to the operation log
- Operation log **replicated** on multiple remote machines
 - **Before** responding to client operation, log record must have been **flushed locally and remotely**
- **Master** recovers its file system from checkpoint + operation

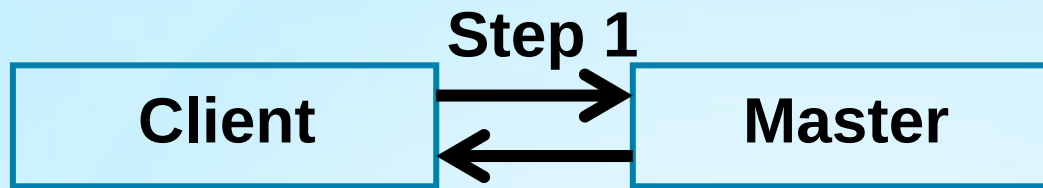
Consistency Model

- **Atomicity and correctness** of file namespace are ensured by **namespace locking**.
- After successful **data mutation(writes or record appends)**, changes are applied to a chunk in the same order on all replicas.
- In case **of chunk server failure** at the time of mutation (stale replica), it is garbage collected at the soonest opportunity.
- **Regular handshakes** between Master and chunk servers helps in identifying failed chunk servers and detects data corruption by **checksumming**.

System Interactions

Leases & Mutation Order

- Master grants **chunk lease** to one of the replicas(primary).
- All replicas follow a **serial order** picked by the primary.
- **Leases timeout at 60 seconds.** (also possible to extend the timeout)
- Leases are **revocable.**



2

Step-1. Client asks master which chunk server holds current lease of chunk and locations of other replicas.

Step-2. Master replies with identity of primary and locations of secondary replicas.

Legend:



Control



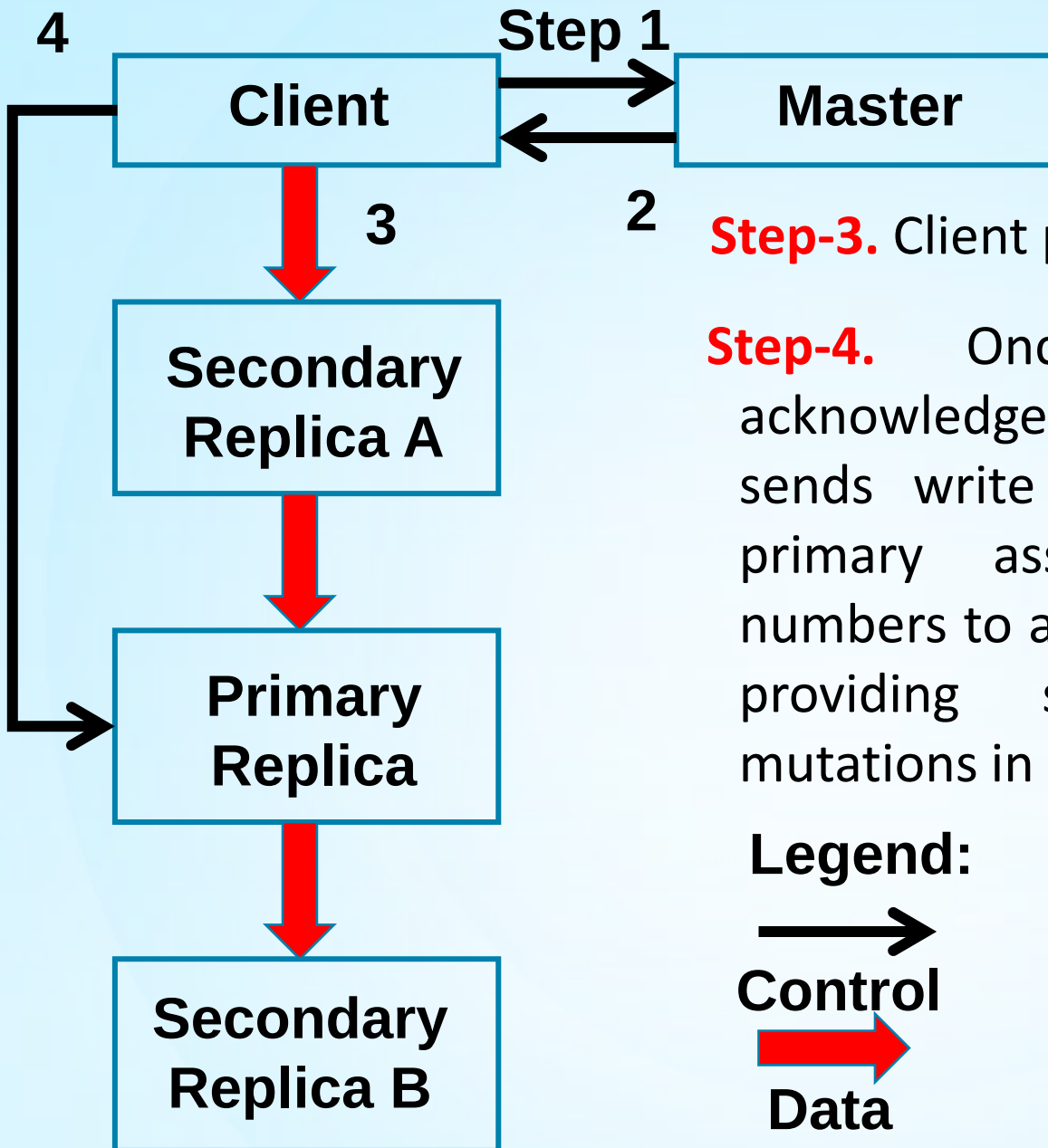
Data

**Secondary
Replica A**

**Primary
Replica**

**Secondary
Replica B**

Figure 2: Write Control and Data Flow



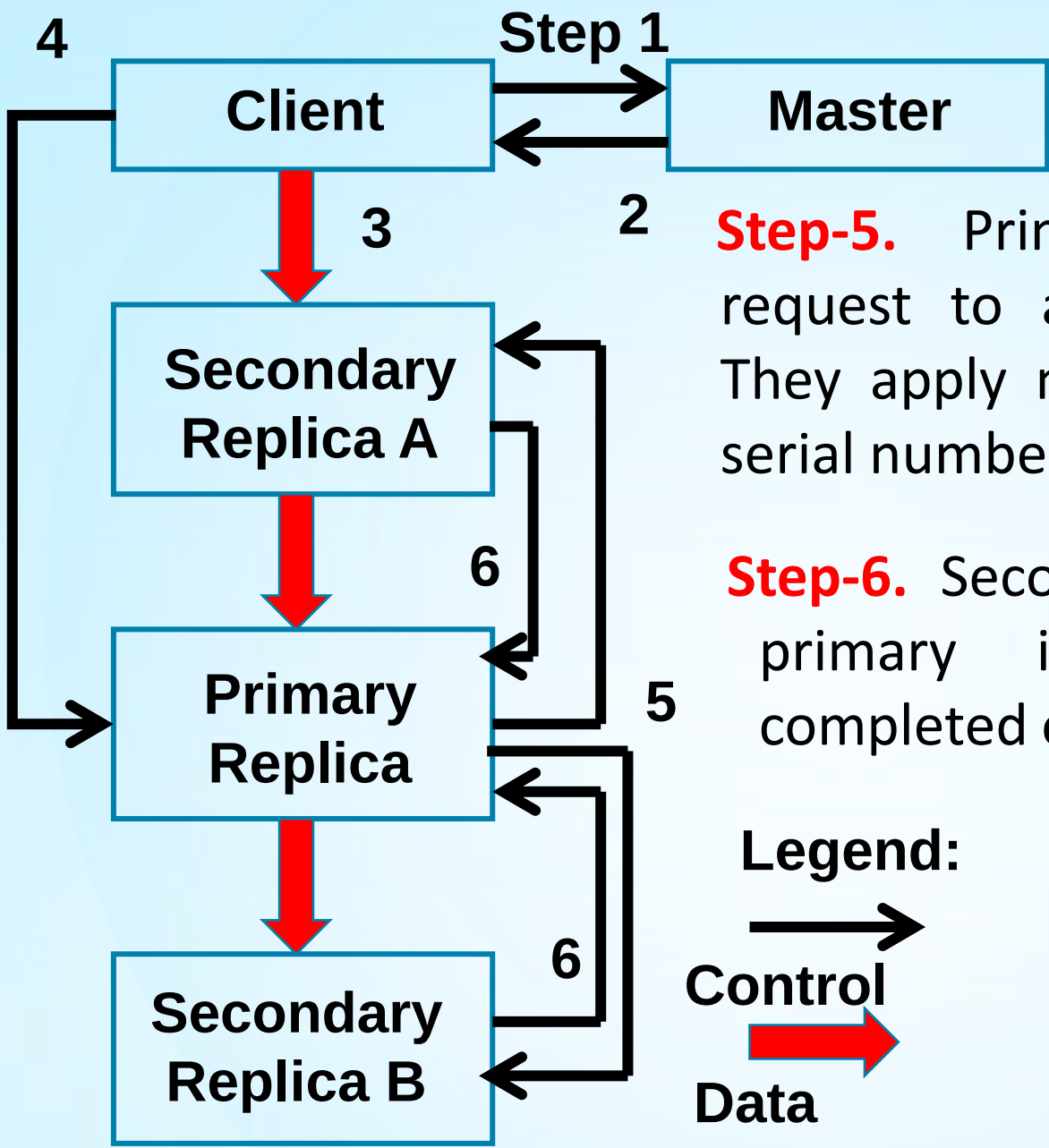
Step-3. Client pushes data to all replicas

Step-4. Once all replicas have acknowledged receiving the data, client sends write request to primary. The primary assigns consecutive serial numbers to all the mutations it receives, providing serialization. It applies mutations in serial number order.

Legend:



Figure 2: Write Control and Data Flow



Step-5. Primary forwards write request to all secondary replicas. They apply mutations in the same serial number order.

Step-6. Secondary replicas reply to primary indicating they have completed operation

Legend:


Control

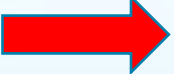
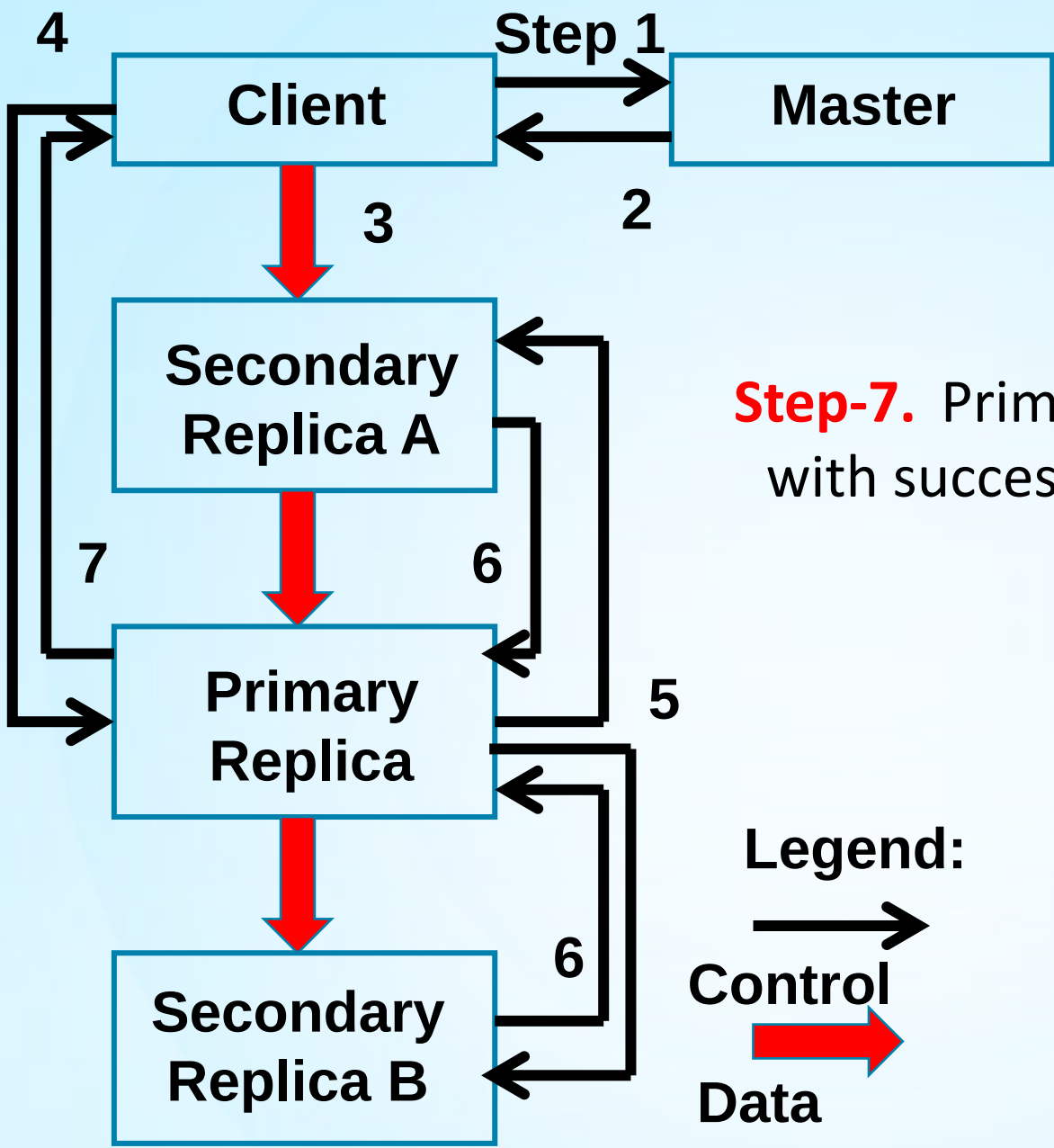

Data

Figure 2: Write Control and Data Flow



Step-7. Primary replies to the client with success or error message

Legend:


Control


Data

Figure 2: Write Control and Data Flow

System Interactions

- **Data Flow**

- Data is pipelined over TCP connections
- A chain of chunk servers form a pipeline
- Each machine forwards data to the closest machine

- **Atomic Record Appends**

- GFS provides an atomic append operation called “record append”

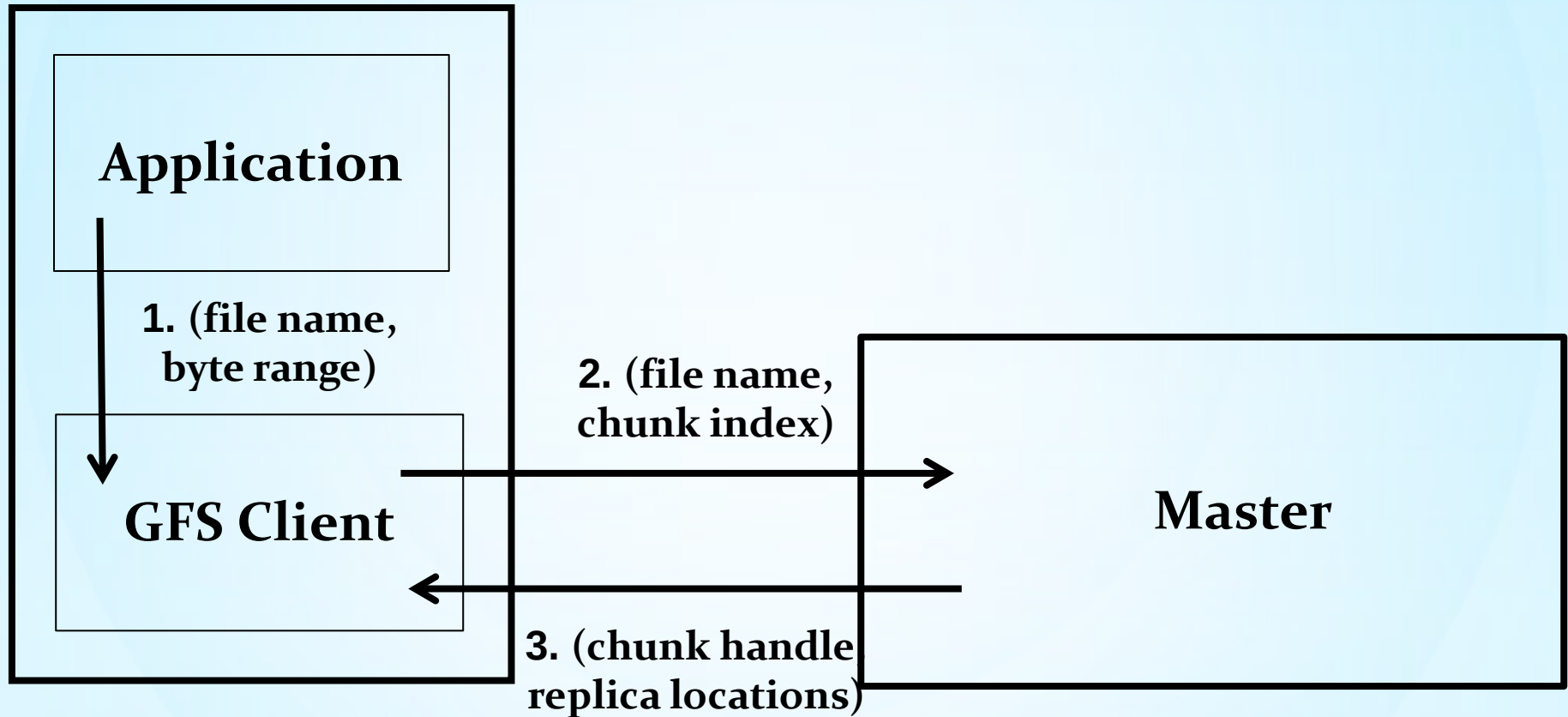
- **Snapshot**

- Makes a copy of file or a directory tree almost instantaneously

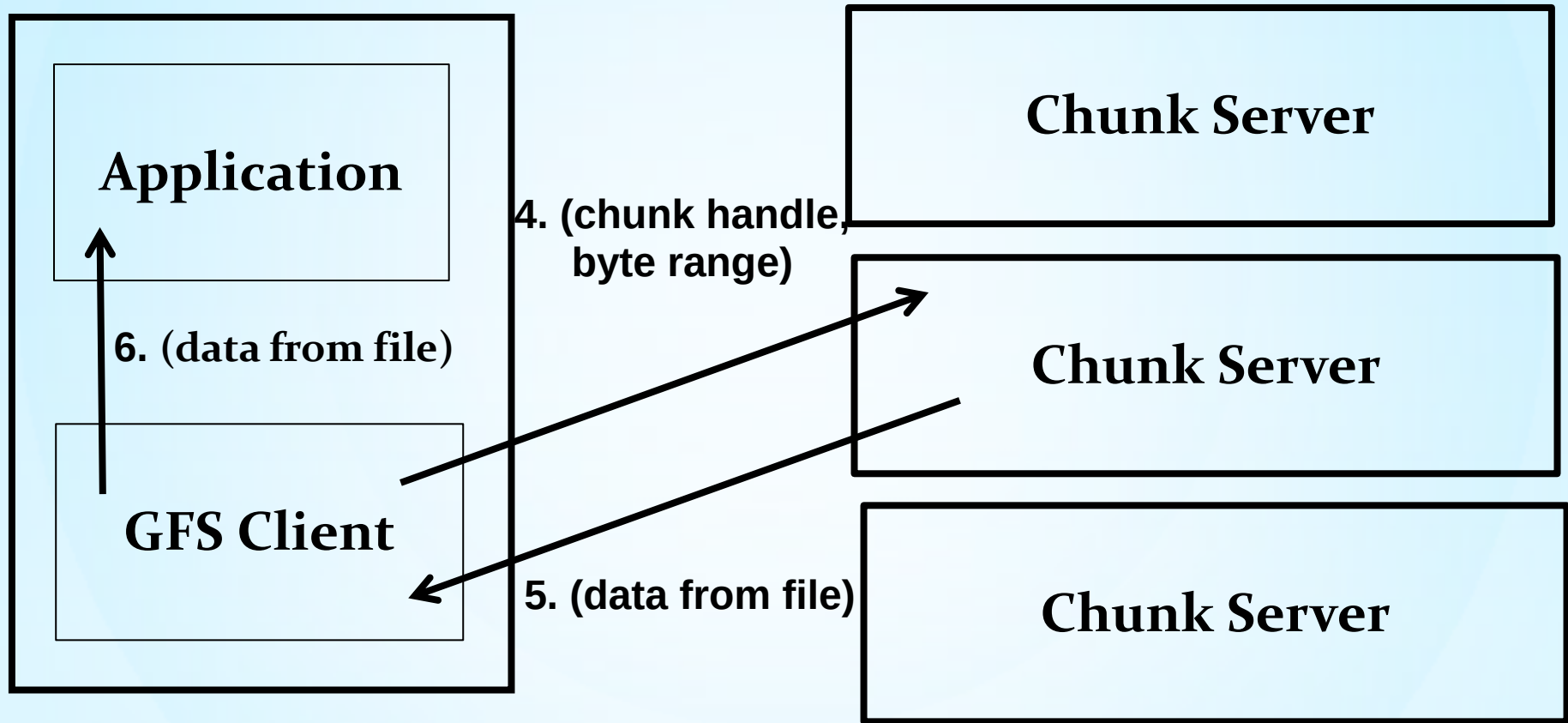
Read Algorithm (1)

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.

Read Algorithm (2)



Read Algorithm (3)



Write Algorithm (1)

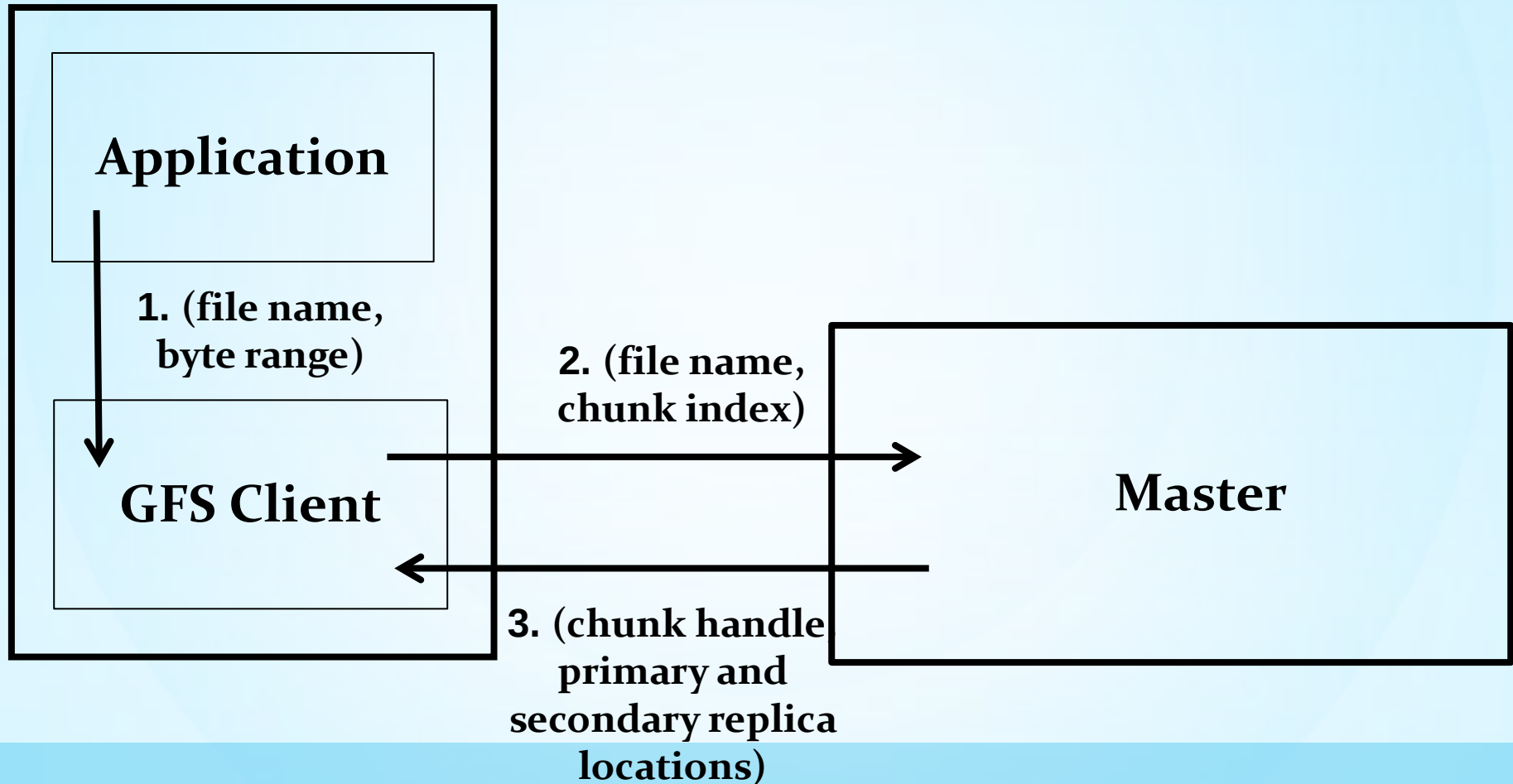
1. Application originates write request.
2. GFS client translates request from (filename, data) -> (filename, chunk index), and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes write data to all locations. Data is stored in chunkservers'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

Write Algorithm (2)

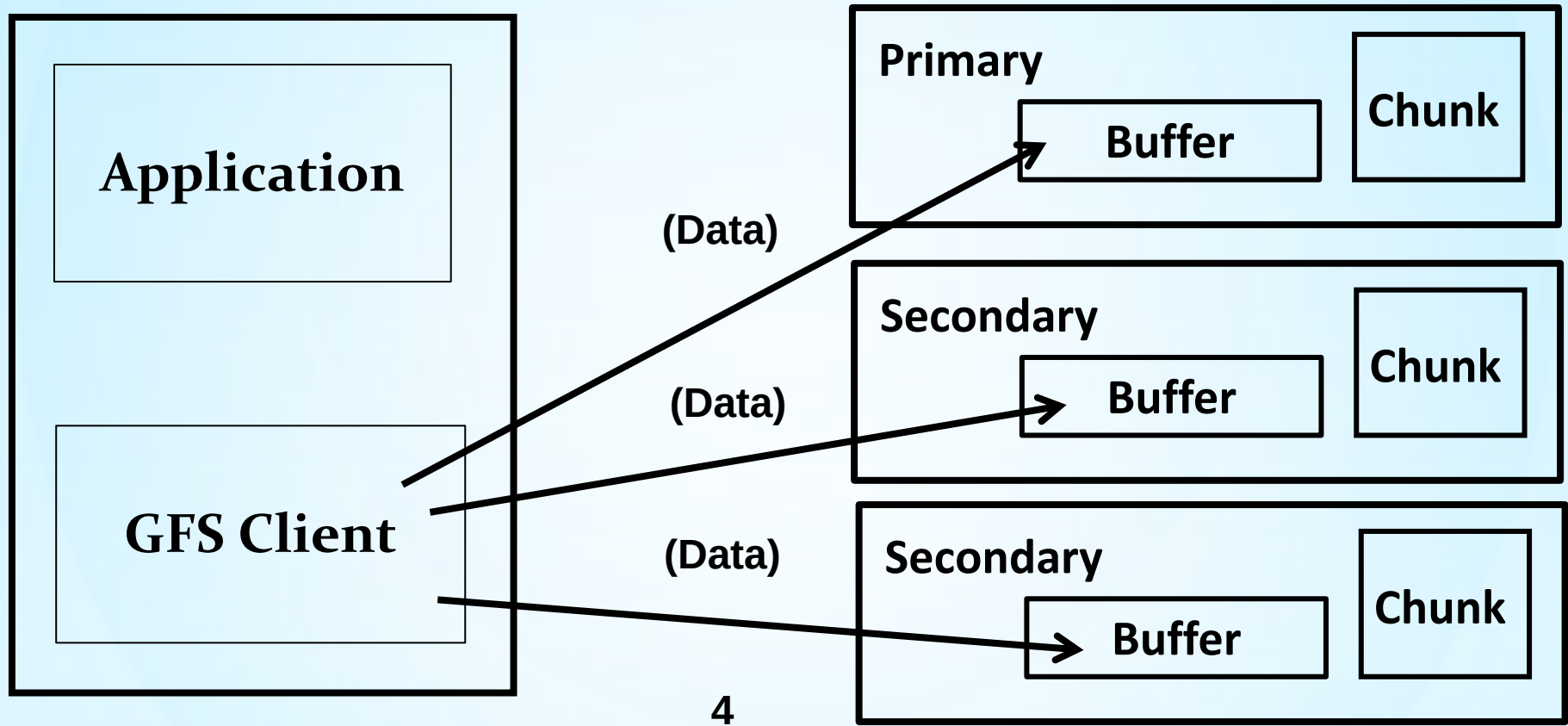
6. Primary determines serial order for data instances stored in its buffer and writes the instances in that order to the chunk.
7. Primary sends serial order to the secondaries and tells them to perform the write.
8. Secondaries respond to the primary.
9. Primary responds back to client.

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

Write Algorithm (3)

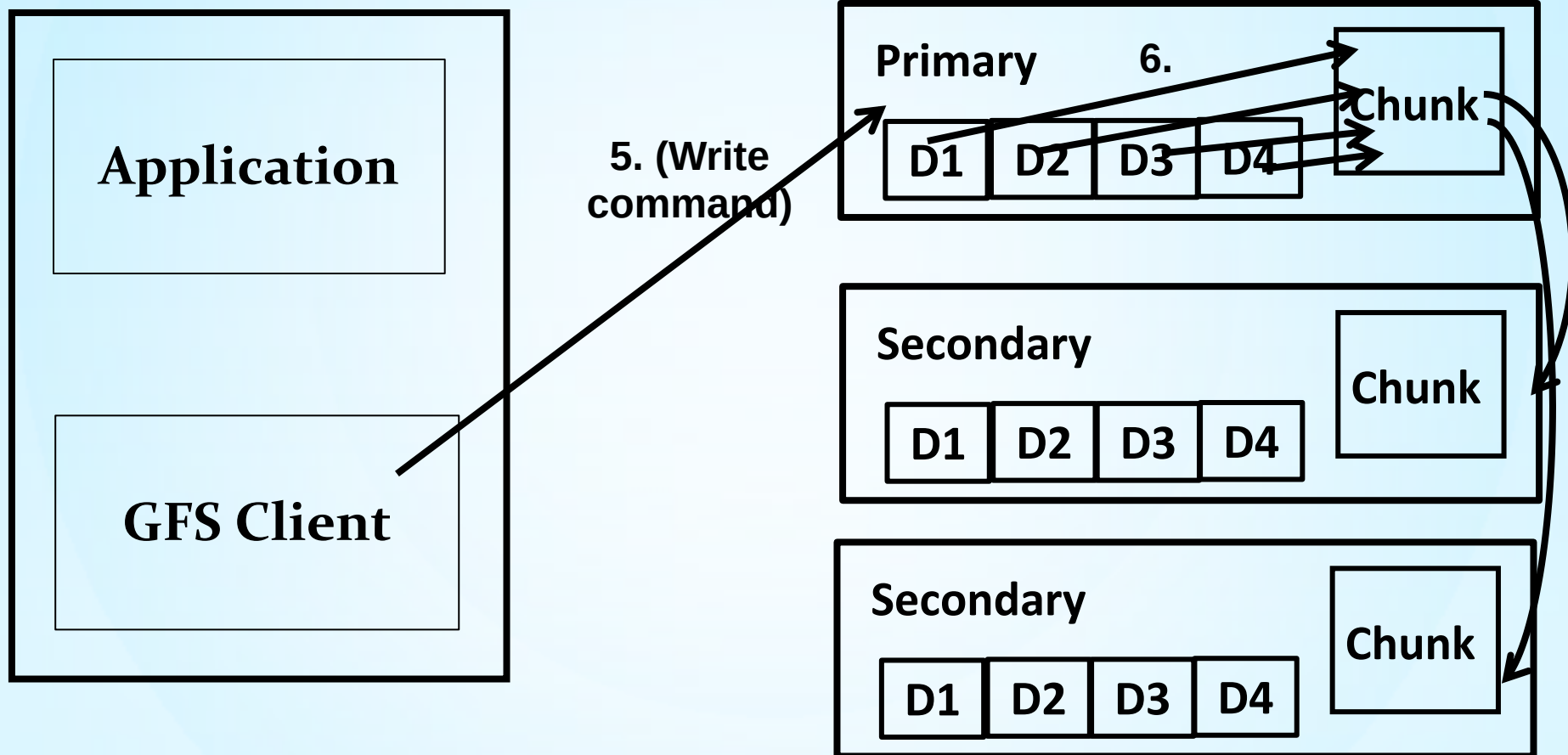


Write Algorithm (4)

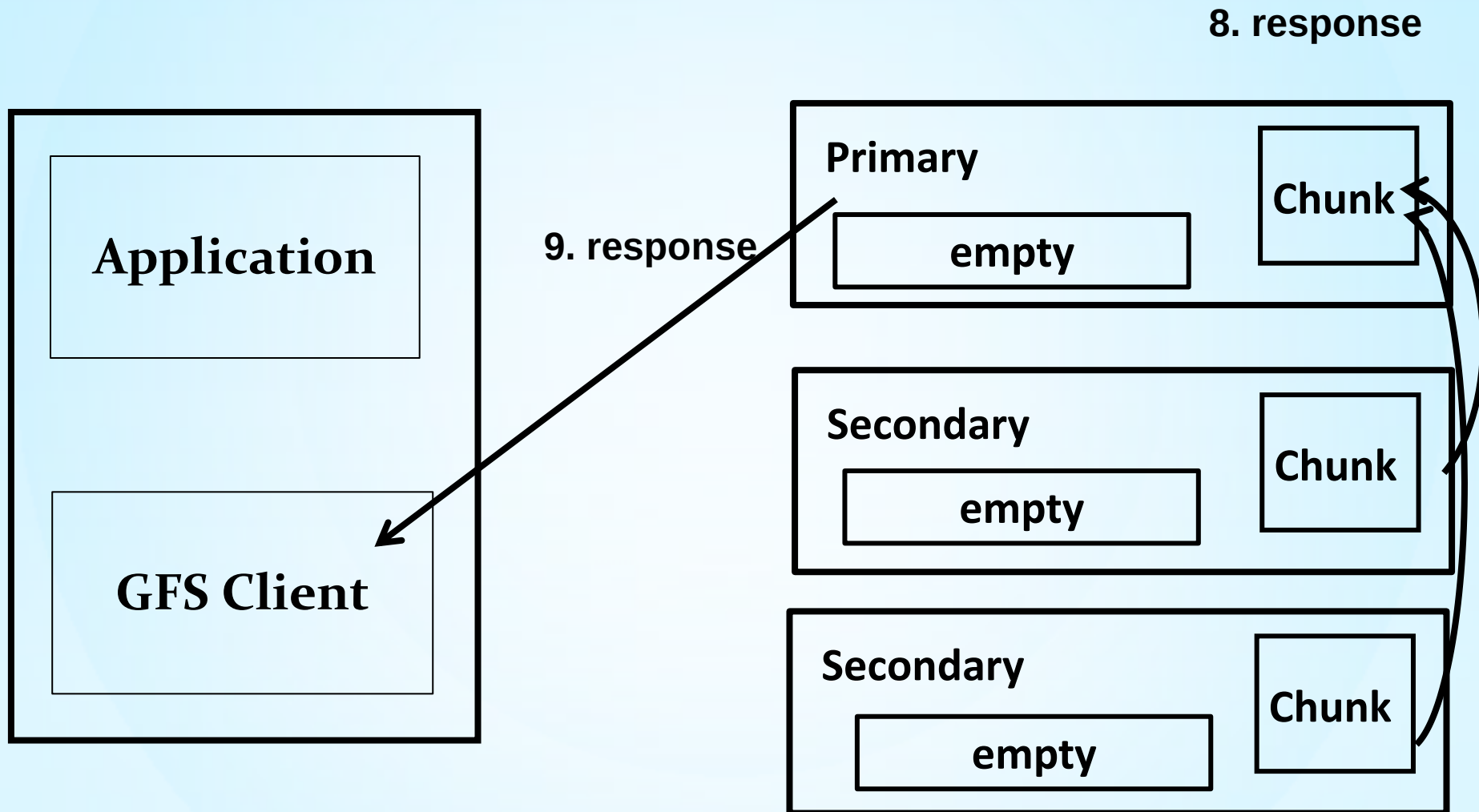


Write Algorithm (5)

7. Write command, serial order



Write Algorithm (6)



Record Append Algorithm (1)

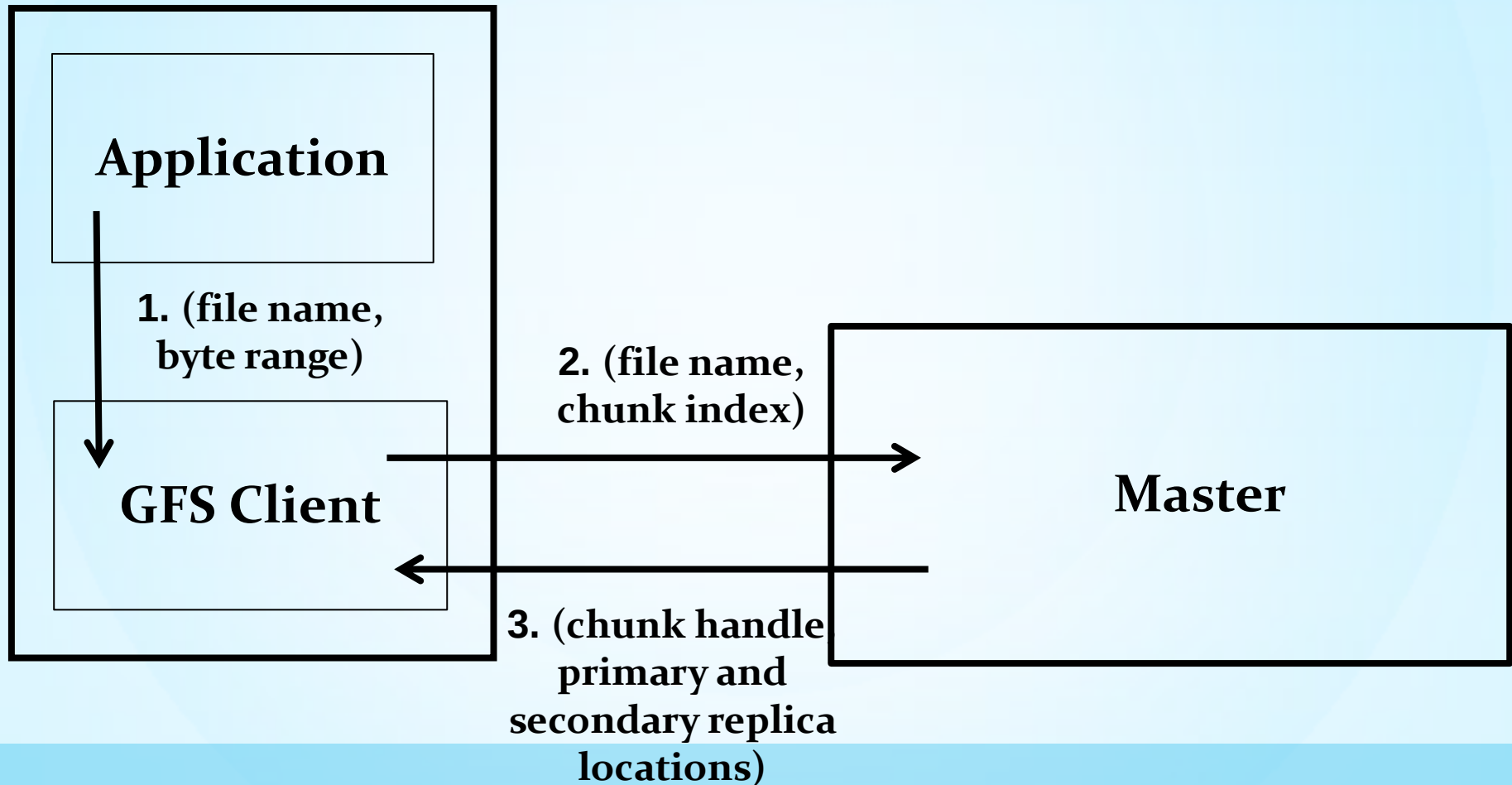
1. Application originates record append request.
2. GFS client translates request and sends it to master.
3. Master responds with chunk handle and (primary + secondary) replica locations.
4. Client pushes writ
5. Primary checks if record fits in specified chunk.
6. If record does not fit, then the primary:
 - pads the chunk,
 - tells secondaries to do the same,
 - and informs the client.
 - Client then retries the append with the next chunk.

Record Append Algorithm (2)

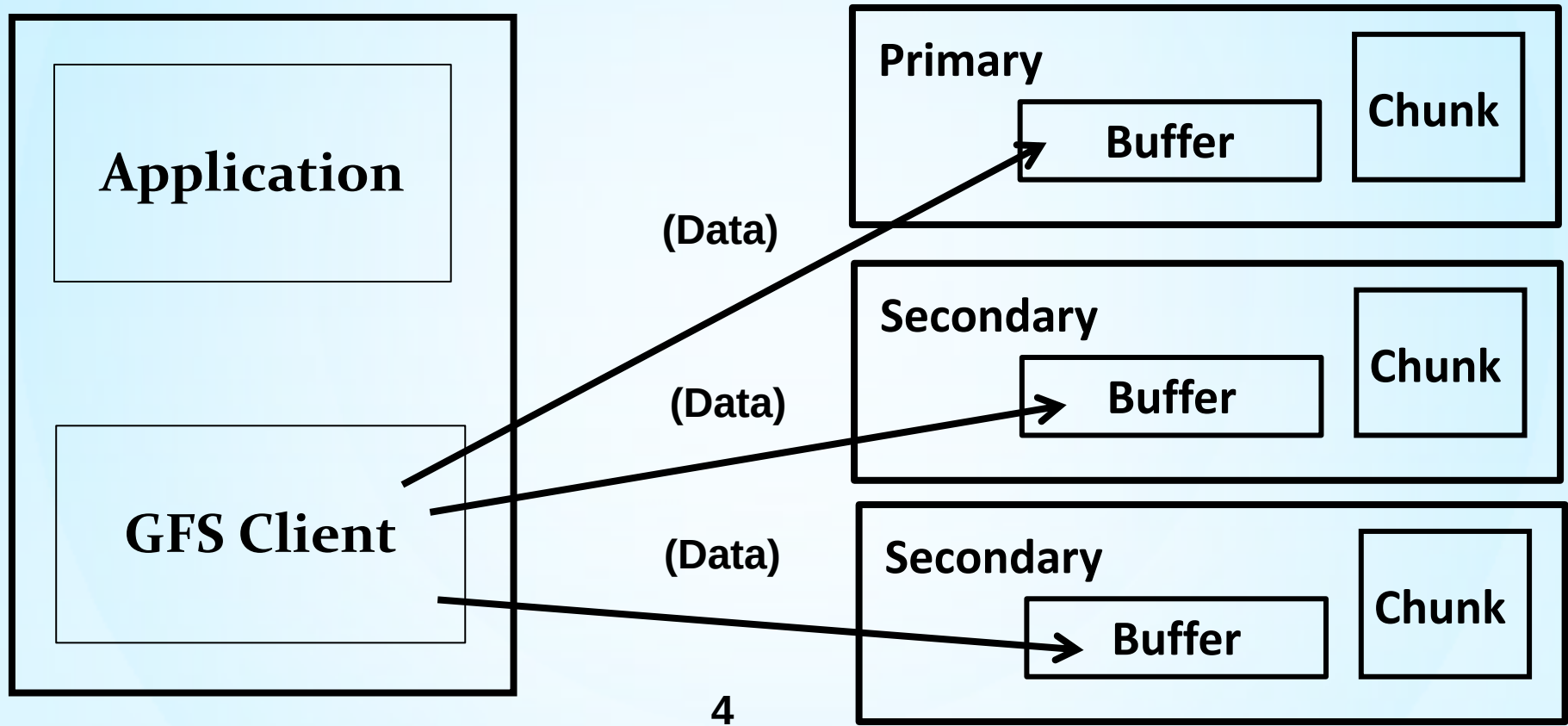
7. If record fits, then the primary:

- appends the record,
- tells secondaries to do the same,
- receives responses from secondaries,
- and sends final response to the client. e data to all locations.

Record Append Algorithm (3)

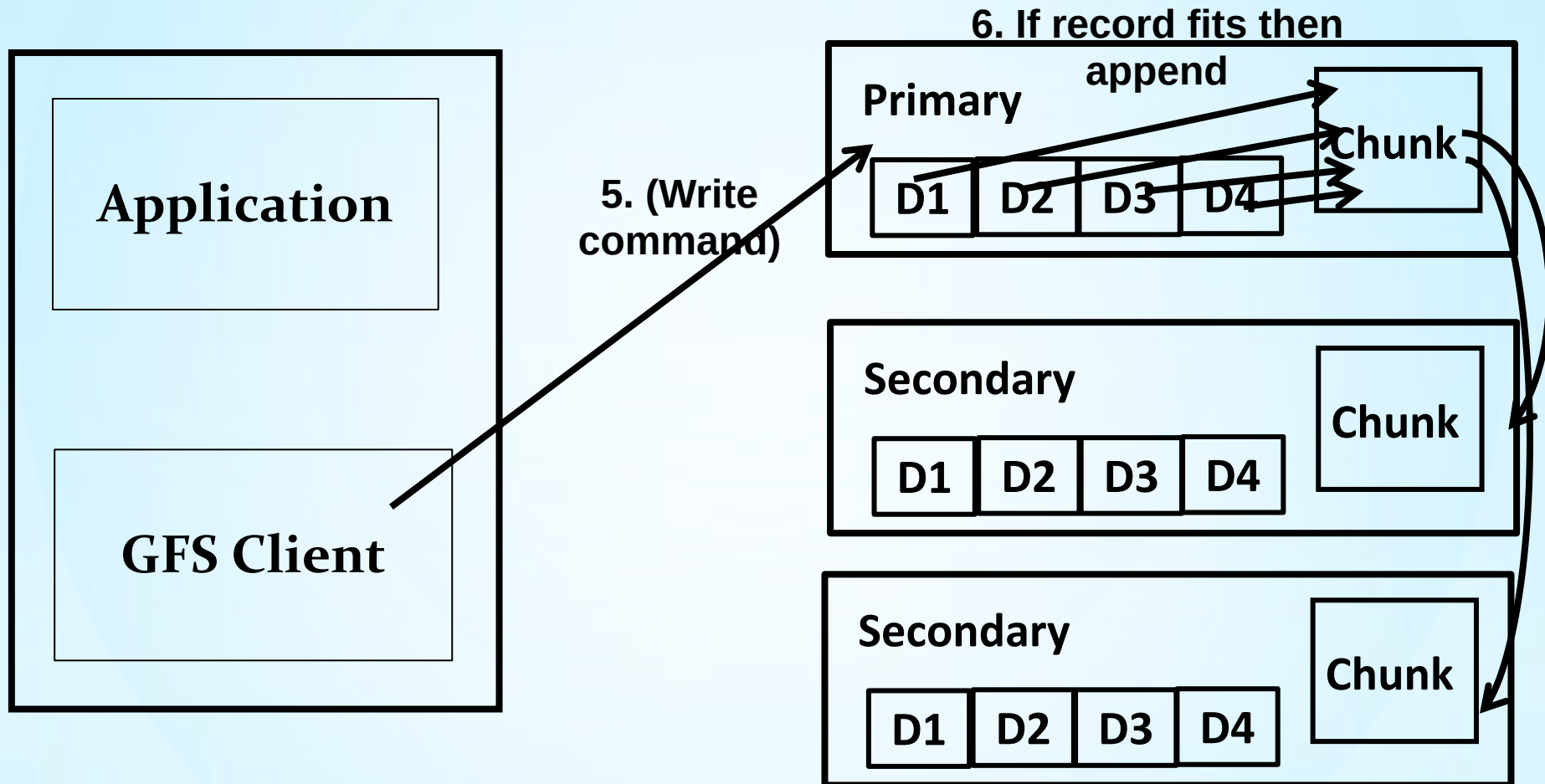


Record Append Algorithm (4)

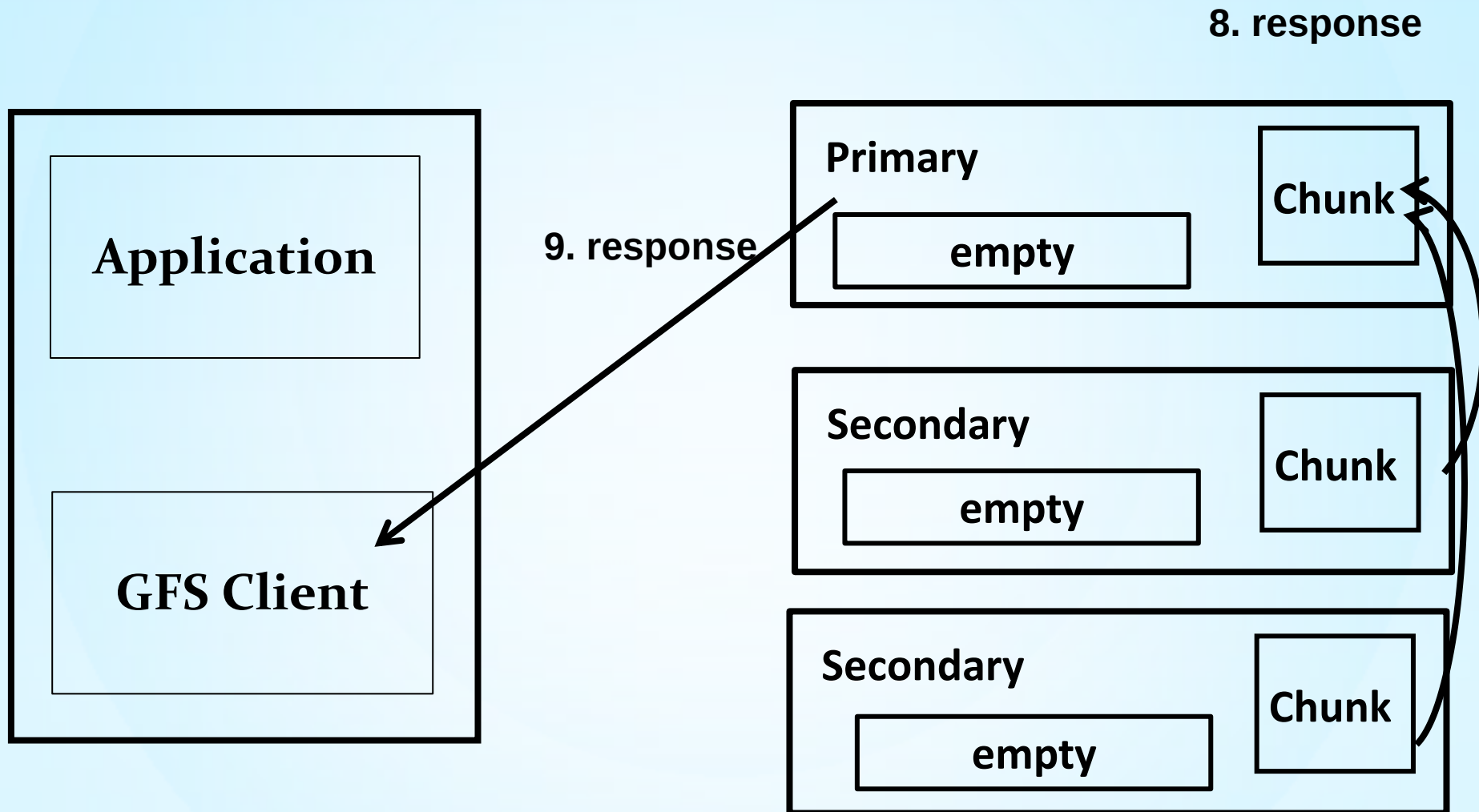


Record Append Algorithm (5)

7. Appends the record



Record Append Algorithm (6)



Master Operation

Namespace Management & Locking

- Locks are used over namespaces to ensure proper serialization
- Read/write locks
- GFS simply uses directory like file names : /foo/bar
- GFS logically represents its namespace as a lookup table mapping full pathnames to metadata
- If a Master operation involves /d1/d2/..dn/leaf, read locks are acquired on d1,/d1/d2,..d1/d2/..leaf and either a read or write lock on the full pathname /d1/d2/.....dn/leaf

Chunk Replica Placement

- Creation of (initially empty) chunks
 - Use under-utilised chunk servers; spread across racks
 - Limit number of recent creations on each chunk server
- Re-replication
 - Started once the available replicas fall below setting
 - Master instructs chunkserver to copy chunk data directly from existing valid replica
 - Number of active clone operations/bandwidth is limited
- Rebalancing
 - The Master Rebalances the replicas periodically (examines replicas distribution and moves replicas for better disk space and load balancing)

Garbage collection

- Deletion logged by master
- File renamed to a hidden file, deletion timestamp kept
- Periodic scan of the master's file system namespace
 - Hidden files older than 3 days are deleted from master's memory (no further connection between file and its chunk)
- Periodic scan of the master's chunk namespace
 - Orphaned chunks (not reachable from any file) are identified, their metadata deleted
- HeartBeat messages used to synchronise deletion between master/chunkserver

Stale replica detection

- Scenario: a chunkserver misses a change (“mutation”) applied to a chunk, e.g. a chunk was appended
- Master maintains a chunk version number to distinguish up-to-date and stale replicas
- Before an operation on a chunk, master ensures that version number is advanced
- Stale replicas are removed in the regular garbage collection cycle

Fault Tolerance

- **Fast Recovery:** master and chunkservers are designed to restart and restore state in few seconds
- **Chunk Replication:** across multiple machines, across multiple racks
- **Master Mechanisms:**
 - Keep 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 master for reading data if real master is down

Reading

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

“The Google File System”

<http://labs.google.com/papers/gfs.html>

Conclusion

- **GFS is a distributed file system** that support large-scale data processing workloads on commodity hardware
- GFS has different points in the design space
 - Component failures as the norm
 - Optimize for huge files
- GFS provides **fault tolerance**
 - Replicating data
 - Fast and automatic recovery
 - Chunk replication
- GFS has the **simple, centralized master** that does not become a bottleneck.