Overview of GFS

- Design goals/priorities
  - Design for big-data workloads
    - Huge files, mostly appends, concurrency, huge bandwidth
  - Design for failures
- Interface: non-POSIX
  - New op: record appends (atomicity matters, order doesn't)
- Architecture: one master, many chunk (data) servers
  - Master stores metadata, and monitors chunk servers
  - Chunk servers store and serve chunks
- Semantics
  - Nothing for traditional write op
  - At least once, atomic record appends

POSIX: Portable Operating System Interface

GFS Workload Characteristics

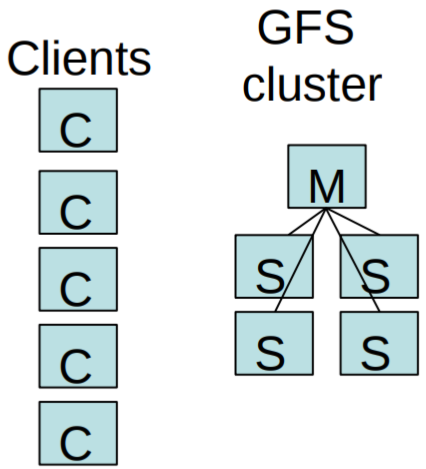
- Files are huge by traditional standards
  - Multi-GB files are common
- Most file updates are appends
  - Random writes are practically nonexistent
  - Many files are written once, and read sequentially
- High bandwidth is more important than latency
- Lots of concurrent data accessing
  - E.g., multiple crawler workers updating an index file

GFS Interface

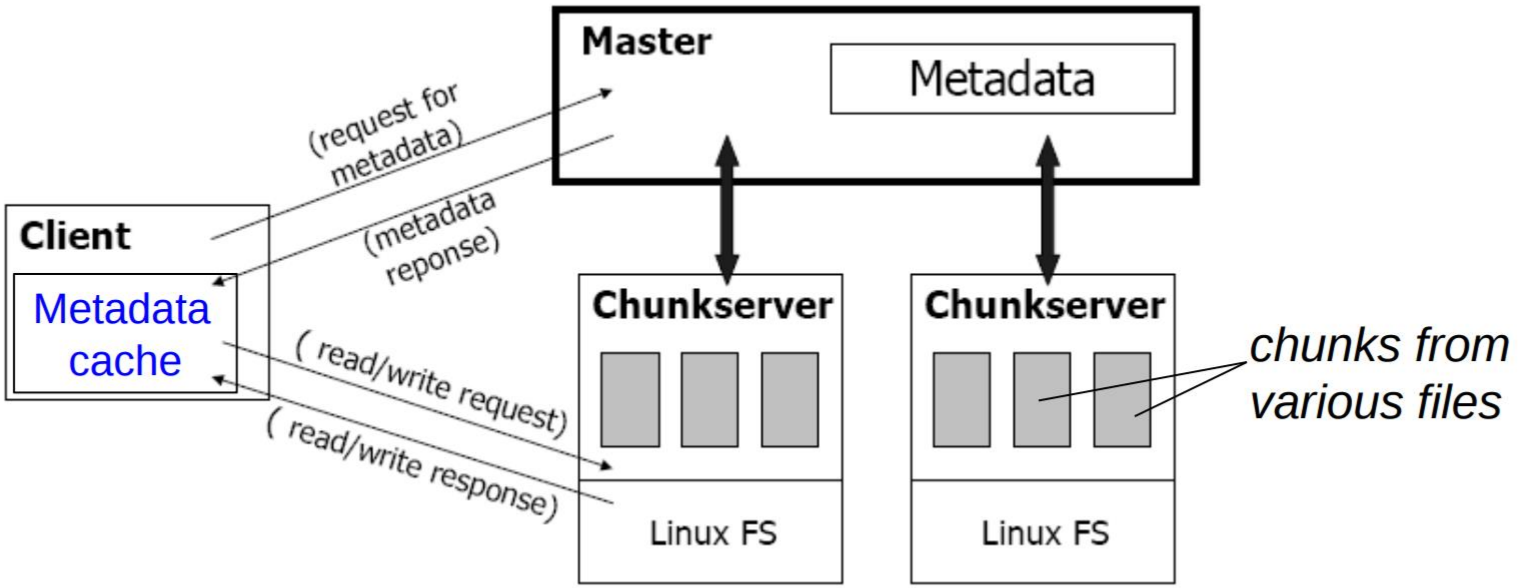
- Not POSIX compliant
  - Supports only popular FS operations, and semantics are different
  - That means you wouldn't be able to mount it
- Additional operation: record append
  - Frequent operation at Google:
  - Merging results from multiple machines in one file (Map/Reduce)
  - Using file as a producer - consumer queue
  - Logging user activity, site traffic
  - Order doesn't matter for appends, but atomicity and concurrency matters

GFS Architecture

- A GFS cluster
  - A single master (replicated later)
  - Many chunkservers
  - Accessed by many clients
- A file
  - Divided into fixed-sized chunks (similar to FS blocks)
  - Labeled with 64-bit unique global IDs (called handles)
  - Stored at chunkservers
  - 3-way replicated across chunkservers
  - Master keeps track of metadata (e.g., which chunks belong to which files)

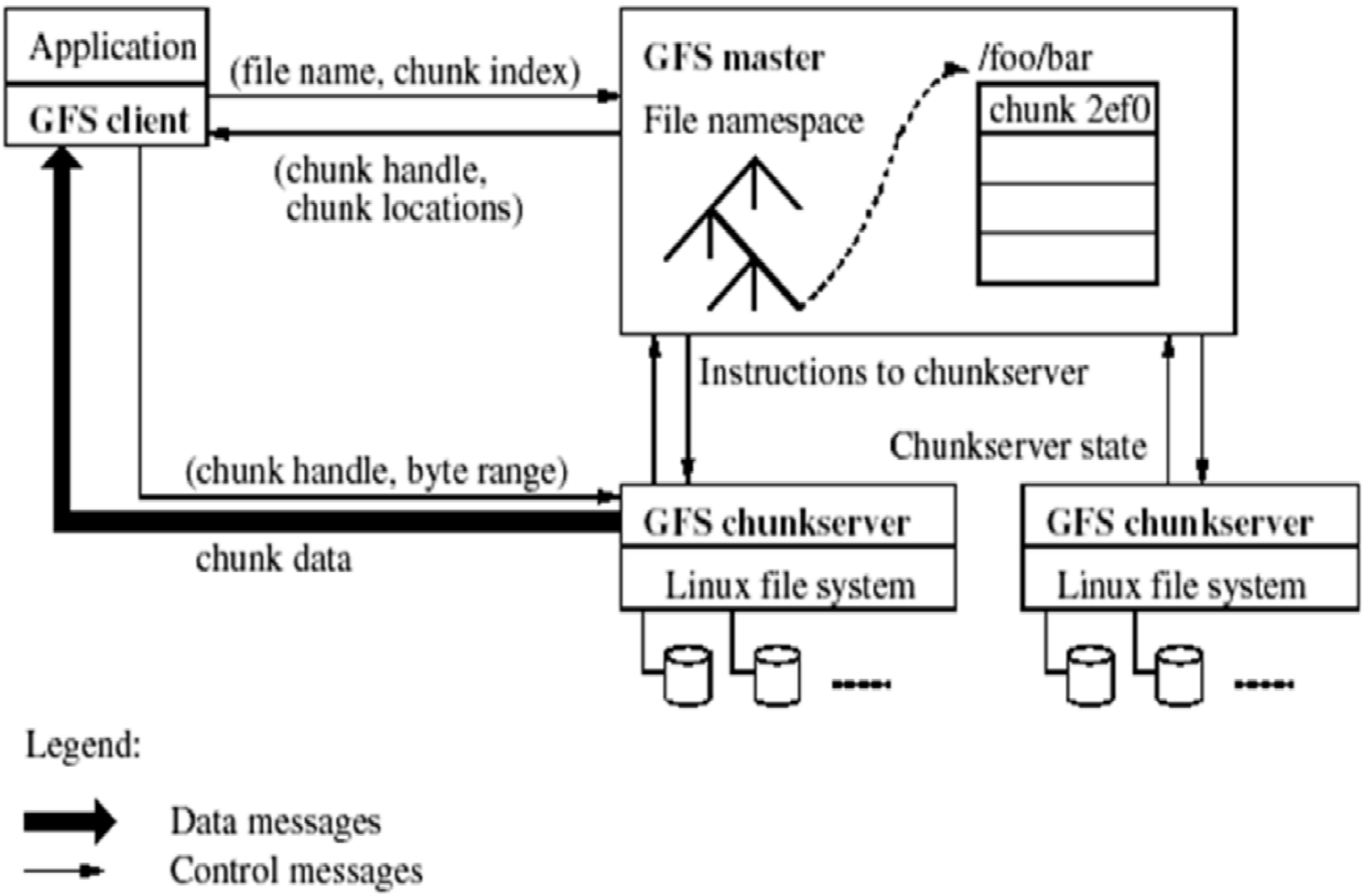


GFS Basic Functioning



- Client retrieves metadata for operation from master
- Read/Write data flows between client and chunkserver
- Minimizing the master's involvement in read/write operations alleviates the single-master bottleneck

Architecture in Picture



Chunks

- Larger blocks: Analogous to FS blocks, except larger
  - Size: 64 MB!
  - Normal FS block sizes are 512B - 8KB
- Pros of big chunk sizes:
  - Less load on server (less metadata, hence can be kept in master's memory)
  - Suitable for big-data applications (e.g., search)
  - Sustains large bandwidth, reduces network overhead
- Cons of big chunk sizes:
  - Fragmentation if small files are more frequent than initially believed

The GFS Master

- A process running on a separate machine
  - Initially, GFS supported just a single master, but then they added master replication for fault-tolerance in other versions/distributed storage systems
- Stores all metadata
  - File and chunk namespaces
    - Hierarchical namespace for files, flat namespace for chunks
  - File-to-chunk mappings
  - Locations of a chunk’s replicas

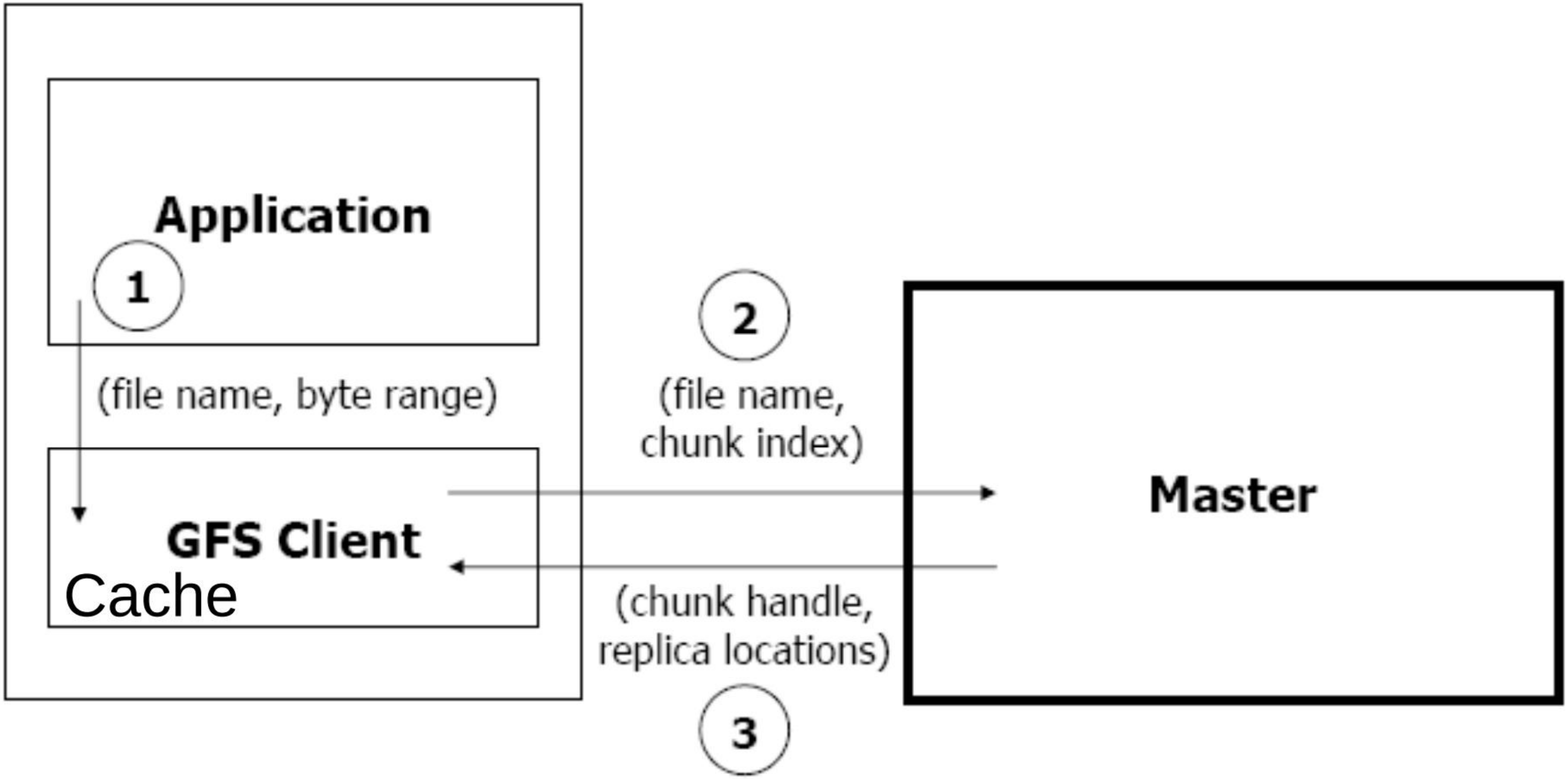
Chunk Locations

- Kept in memory, no persistent states
  - Master polls chunkservers at startup
- What does this imply?
  - Upsides: master can restart and recover chunks from chunkservers
    - Note that the hierarchical file namespace is kept on durable storage in the master
  - Downside: restarting master takes a long time
- Why do you think they do it this way?
  - Design for failures
  - Simplicity
  - Scalability – the less persistent state master maintains, the better

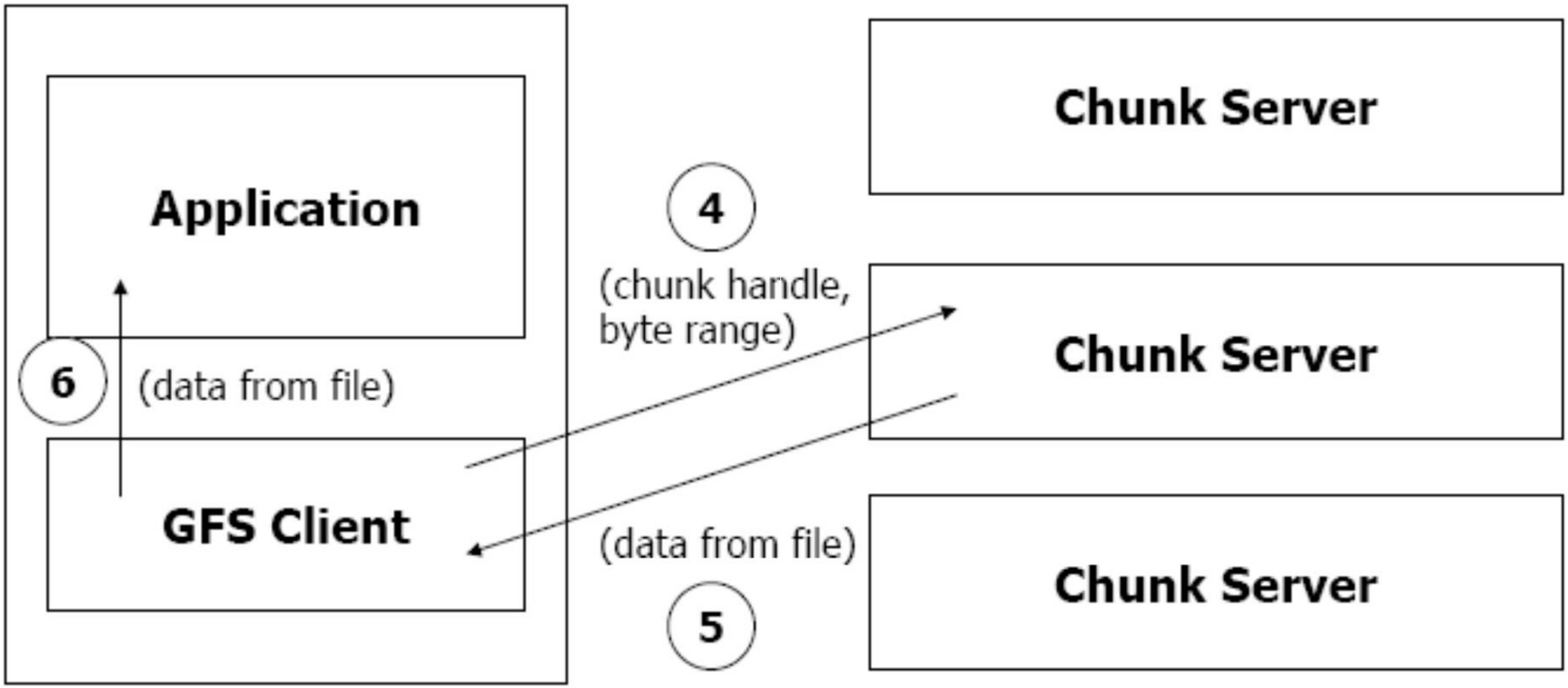
GFS Master <--> Chunkservers

- Master and chunkserver communicate regularly (heartbeat):
  - Is a chunkserver down?
  - Are there disk failures on a chunkserver?
  - Are any replicas corrupted?
  - Which chunks does a chunkserver store?
- Master sends instructions to chunkserver:
  - Delete a chunk
  - Create a new chunk
  - Replicate and start serving this chunk (chunk migration)
    - Why do we need migration support?

GFS Operations – Read



GFS Operations – Read



GFS Operations – Updates

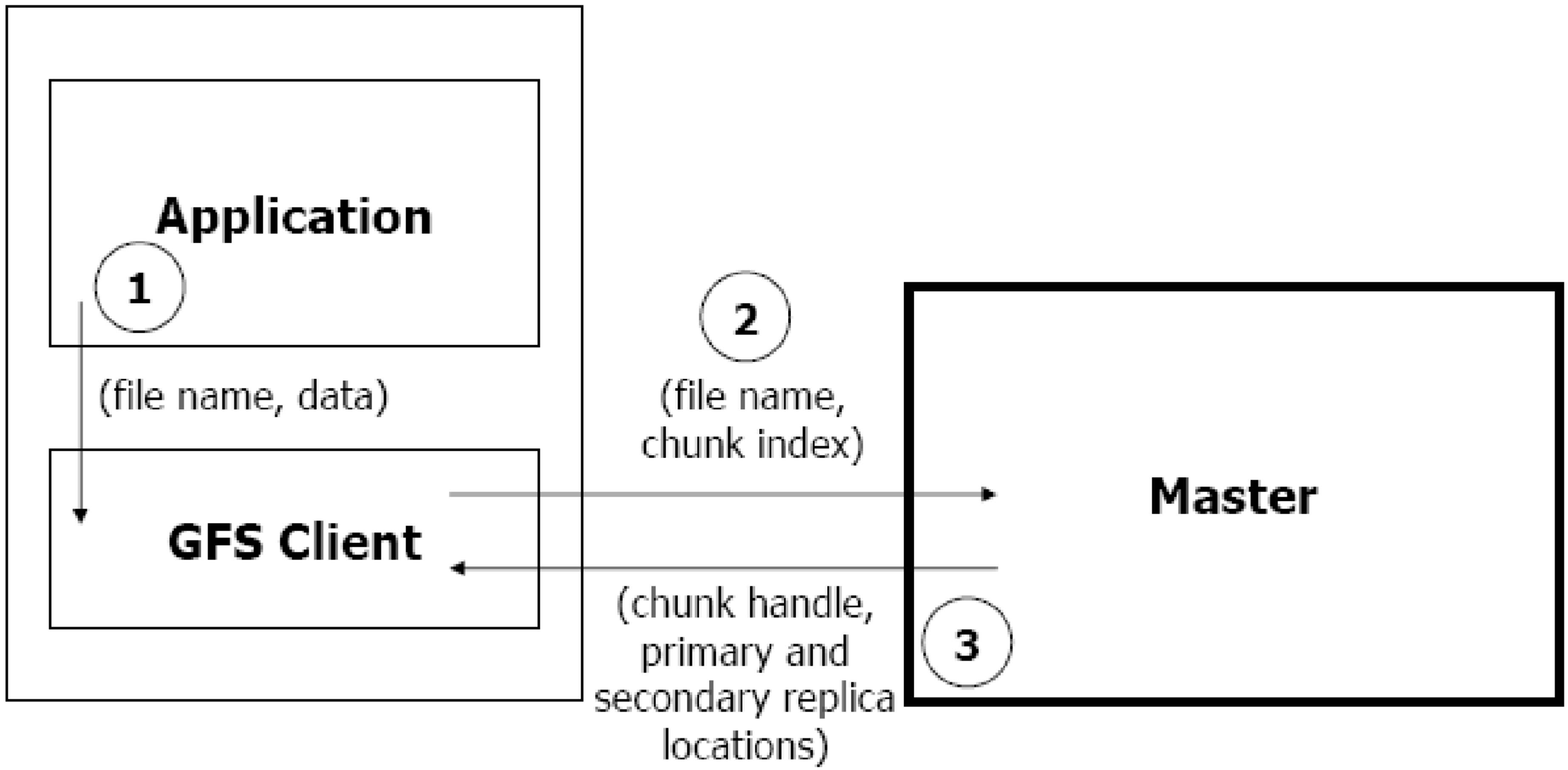
- Two update operations supported
  - Write – to a specific location in a file
  - RecordAppend
- For consistency, updates to each chunk must be ordered in the same way at the different chunk replicas
  - Consistency means that replicas will end up with the same version of the data and not diverge
- For this reason, for each chunk, one replica is designated as the primary
- The other replicas are designated as secondaries
- Primary defines the update order
- All secondaries follows this order

Primaries

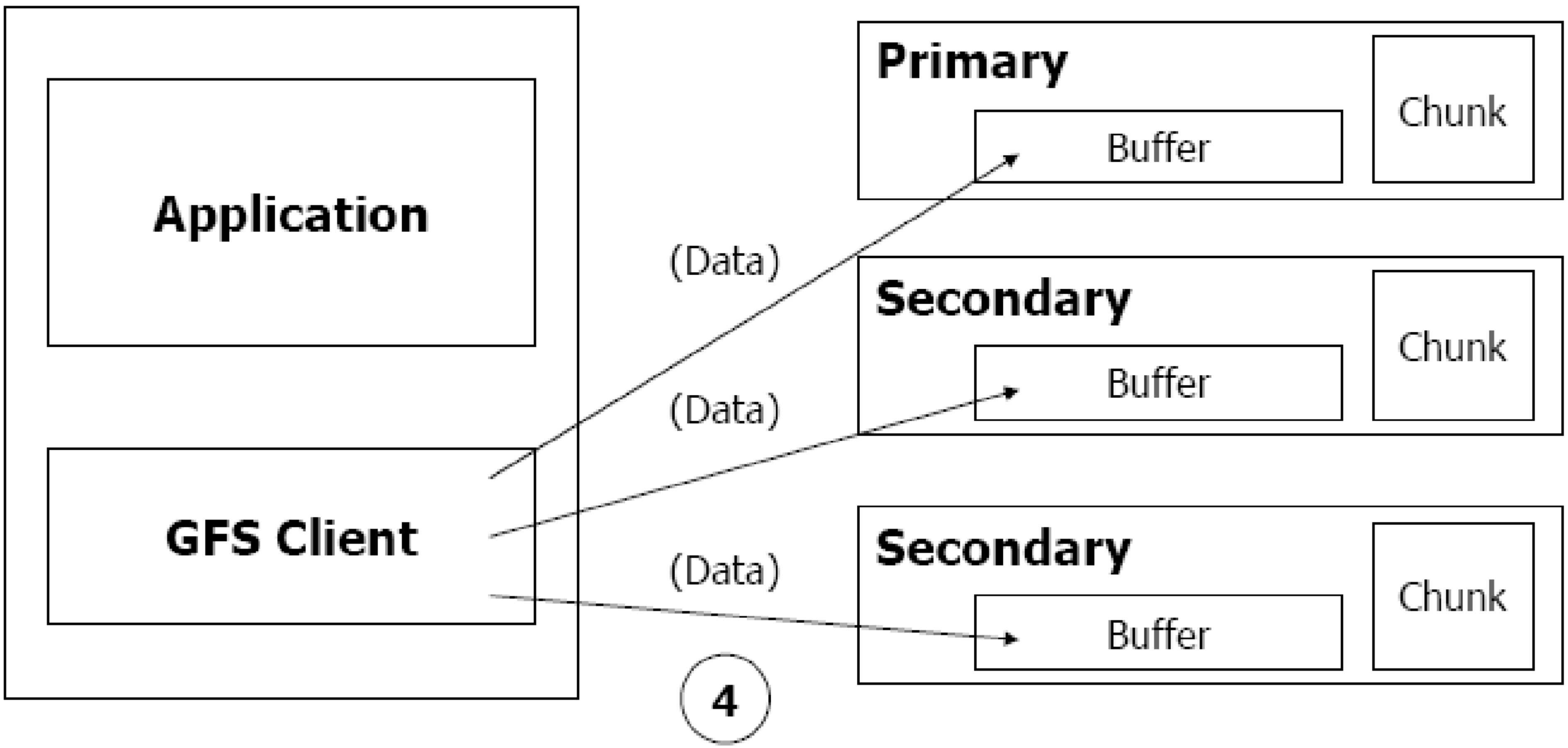
- For correctness, at any time, there needs to be one single primary for each chunk
  - Or else, they could order different writes in different ways
- To ensure that, GFS uses leases
  - Master selects a chunkserver and grants it lease for a chunk
- The chunkserver holds the lease for a period T after it gets it, and behaves as primary during this period
- The chunkserver can refresh the lease endlessly
- But if the chunkserver can't successfully refresh lease from master, it stops being a primary
- If master doesn't hear from primary chunkserver for a period, the master gives the lease to some other chunkserver
- So, at any time, at most one server is the primary for each chunk
  - But different servers can be primaries for different chunks.

Write

4 pages of pictures

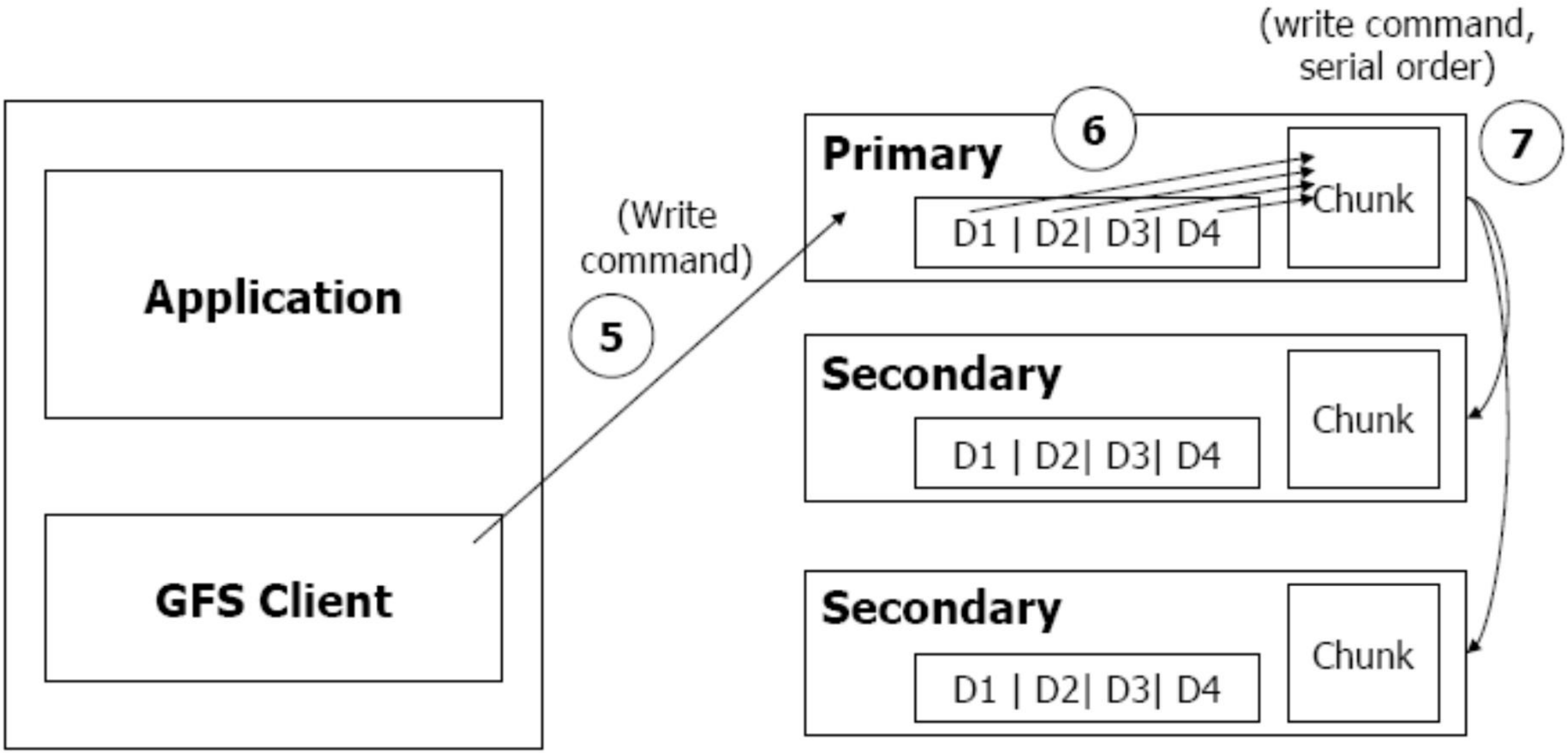


Write



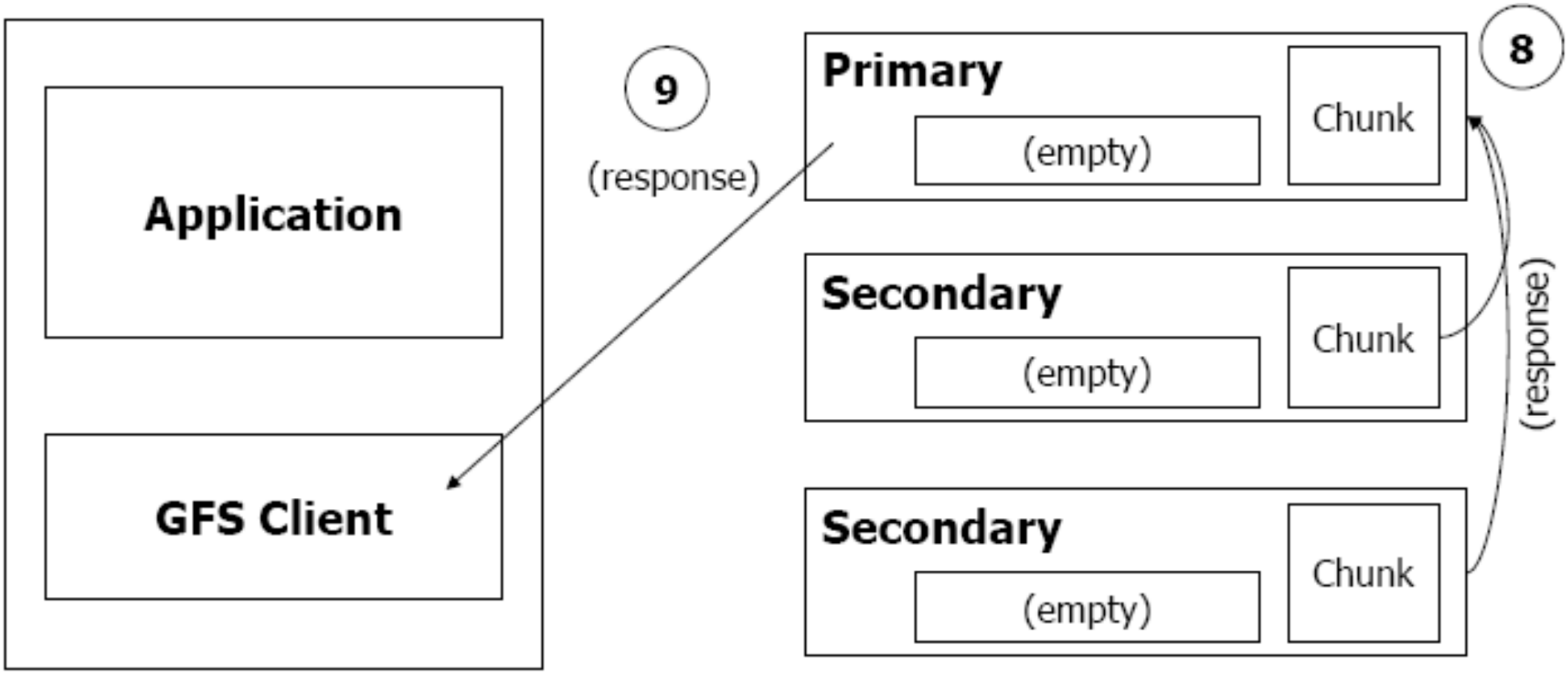
Application
GFS Client
(Data)
(Data)
(Data)
4
Primary
Buffer
Chunk
Secondary
Buffer
Chunk
Secondary
Buffer
Chunk

Write



Application
GFS Client
(Write command)
5
Primary
6
D1 | D2| D3| D4
Chunk
(write command, serial order)
7
Secondary
D1 | D2| D3| D4
Chunk
Secondary
D1 | D2| D3| D4
Chunk

Write

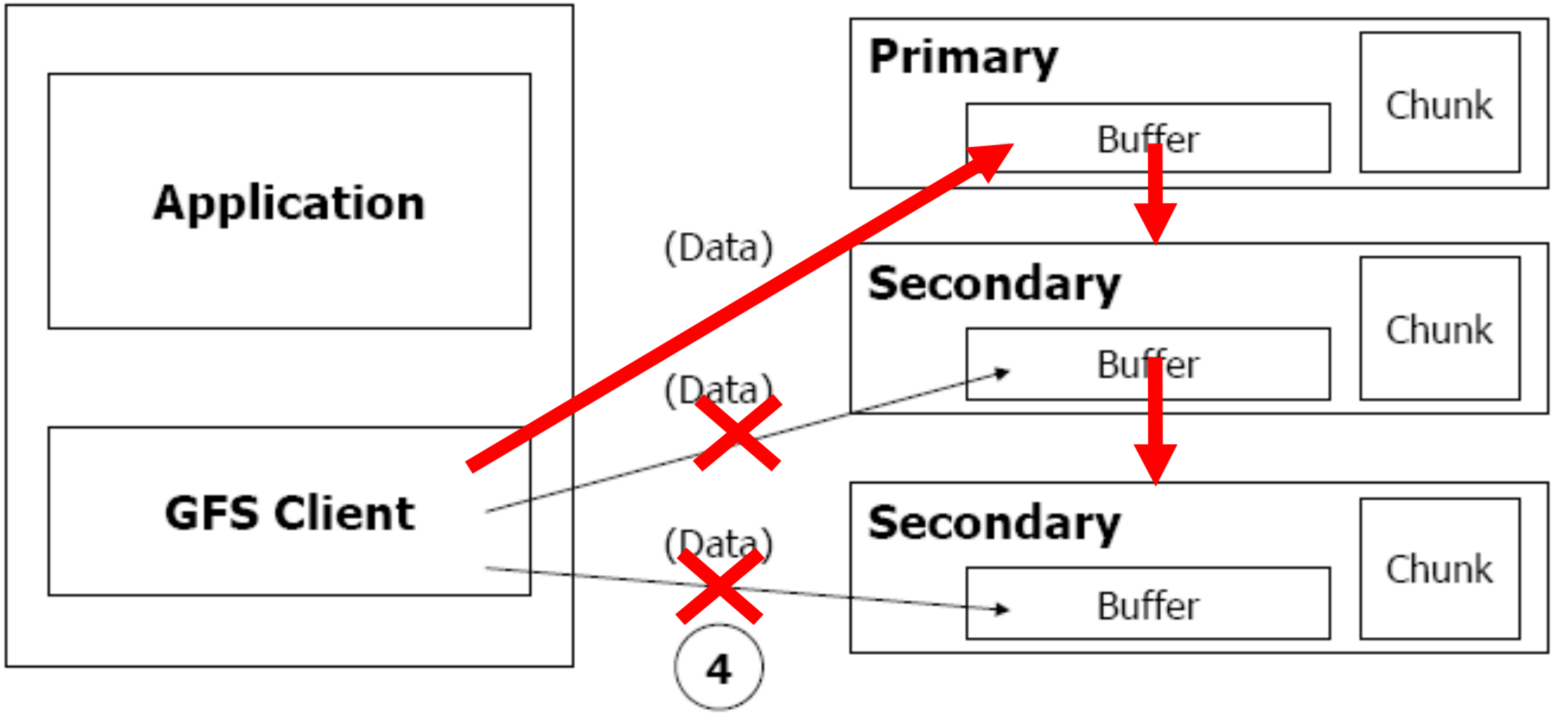


Application
GFS Client
9
(response)
Primary
(empty)
Chunk
Secondary
(empty)
Chunk
Secondary
(empty)
Chunk
8
(response)

Write

- Primary enforces one update order across all replicas for concurrent writes
- It also waits until a write finishes at the other replicas before it replies
- Therefore:
  - We’ll have identical replicas
  - But, file region may end up containing mingled fragments from different clients
  - E.g. writes to different chunks may be ordered differently by their different primary chunkservers
- Thus, writes are consistent but the impact of concurrent is undefined in GFS

Data From Client to Chunkservers



- Refer back to the picture earlier as it shows the client sending the data to be written to each chunkserver.
- Actually, the client doesn't send the data to everyone.
- It sends the data to one replica, then replicas send the data in a chain to all other replicas
- Why? To maximize bandwidth and throughput!

Record Append

- The client specifies only the data, not the file offset
  - File offset is chosen by the primary
  - Why do they have this?
- Provide meaningful semantic: at least once atomically
- Because FS is not constrained wrt where to place data, it can get atomicity without sacrificing concurrency

Record Append Steps

1. Application originates a 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.
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 of the inability/error.
   - Client then retries the append with the next chunk.
     - The primary chunk server for the next chunk may not be the same as this primary! So client has to retry.
7. If record fits, then the primary:
   - appends the record at some offset in chunk,
   - tells secondaries to do the same (specifies offset),
   - receives responses from secondaries,
   - and sends final response to the client.

Record Append Steps

- GFS may insert padding data in between different record append operations
- Preferred that applications use this instead of write
- Applications should use mechanisms such as checksums with unique IDs to handle padding

Summary of GFS

- Optimized for large files and sequential appends
  - large chunk size
- File system API tailored to stylized workload
- Single-master design to simplify coordination
  - But minimize workload on master by not involving master in large data transfers
- Implemented on top of commodity hardware
  - Unlike AFS/NFS, which for scale, require a pretty hefty server