Distributed Systems

Spring 2018

International Institute of Information Technology

Hyderabad, India

Most slides taken from the GFS conference talk at ACM SOSP 2003. Slides on NFS and AFS are from the lecture notes of Roxana Geambasu, Columbia U.

Distributed File Systems

- Earliest versions of distributed file systems known as
  - NFS – Networked File System
    - Developed at Sun Microsystems which believed in the slogan “The Network is the Computer”
  - AFS – Andrew File System
    - Part of the Andrew project at CMU that created a distributed computing environment at CMU.
- Modern projects on distributed file systems include
  - GFS – The Google File System,
  - HDFS – the OpenSource version of GFS
  - Colossus – the next version of GFS
- Among all, there is a common thread that we will explore today.

Goals of NFS and AFS

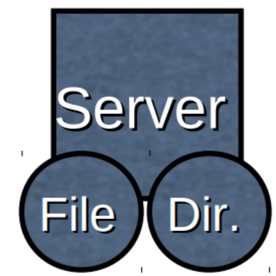
- Have a consistent namespace for files across computers
- Let authorized users access their files from any computer
- On the other hand, distributed systems have a variety of goals including
  - Scalability
  - Fault tolerance
  - Concurrency
  - Security
  - ...
- Cannot meet all goals all the time
  - Prioritize towards most important goals.

Design Principle of Distributed File Systems

- Workload-oriented design
  - Measure characteristics of target workloads to inform the design
- For instance, AFS and NFS are user-oriented, hence they optimize to how users use files (vs. big programs)
  - Most files are privately owned
  - Not too much concurrent access
  - Sequential is common; reads more common than writes
- Other distributed FSes (e.g., Google FS) are geared towards big-program/big-data workloads

What is a File System?

File Ops	Directory Ops
Open	Create file
Read	Mkdir
Write	Rename file
Read	Rename directory
Write	Delete file
Close	Delete directory



Early Design Options

- Use RPC to forward every FS operation to the server
  - Server orders all accesses, performs them, and sends back result
  - NFS v1 was built like this.
- Plus: Same behavior as if both programs were running on the same local filesystem!
- Minus: Performance will suffer. Latency of access to remote server often much higher than to local memory.
- Moreover server would get overloaded as accessing server is needed for every action.
- We need designs that avoid accessing the server for everything? What can we avoid this for? What do we lose in the process?

A Better solution

All problems in computer science can be solved by adding a level of indirection; but this will usually cause other problems" -- David Wheeler (of the Burrows-Wheeler Transform used in data compression)

- Per above, we will try the same idea for distributed file systems along with caching.
- But we should understand the risks of caching in terms of consistency.

NFS

- Cache file blocks, directory metadata in RAM at both clients and servers.
- Plus: No network traffic if open/read/write/close can be done locally.
- Minus: failures and cache consistency are big concerns with this approach
- NFS trades some consistency for increased performance...

Problems with Caching

- Server crashes
  - Any data that’s in memory but not on disk is lost
  - What if client does seek(); /* SERVER CRASH */; read()
    - If server maintains file position in RAM, the read will return bogus data
- Lost messages
  - What if we lose acknowledgement for delete(“foo”)
  - And in the meantime, another client created foo anew?
  - The first client might retry the delete and delete new file
- Client crashes
  - Might lose data updates in client cache

NFS Solution

- Stateless design
  - Flush-on-close: When file is closed, all modified blocks sent to server. close() does not return until bytes safely stored.
- Stateless protocol: requests specify exact state.
  - read() -> read([position]). no seek on server.
- Operations are idempotent
  - How can we ensure this? Unique IDs on files/directories.
  - It’s not delete(“foo”), it’s delete(1337f00f), where that ID won’t be reused.
    - See the level of indirection

Caching and lack of Consistency

- A writer and a reader may notice inconsistency due to writes not taking effect at the server but only in the cache.
- NFS allows for this inconsistency.
- Requires flush on close.
  - Flush all updates from cache to server on close operation.
- This means the system can be inconsistent for a few seconds: two clients doing a read() at the same time for the same file could see different results if one had old data cached and the other didn't.
- Periodic checks to minimize damage.
- Called as weak consistency.
- NFS provides no guarantees at all on multiple writes.

AFS in Brief

- With some commonalities, we will study about AFS in brief.
  - Read in detail offline.
- AFS includes
  - More aggressive caching (AFS caches on disk in addition to RAM)
  - Prefetching (on open, AFS gets entire file from server, making subsequent ops local & fast)
  - Close-to-open consistency only
    - Why does this make sense? (Hint: user-centric workloads)
  - Cache invalidation callbacks
    - Clients register with server that they have a copy of file
    - Server tells them: “Invalidate!” if the file changes
    - This trades server-side state (read: scalability) for improved consistency

Summary of NFS and AFS

- For both AFS and NFS, the central server is:
  - Bottleneck: reads / writes hit it at least once per file use
  - Single point of failure
  - Expensive: to make server fast and reliable, you need to go for expensive hardware
- GFS addresses some of these concerns
  - But tailored for slightly different users/workloads.

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

GFS Workload Characteristics

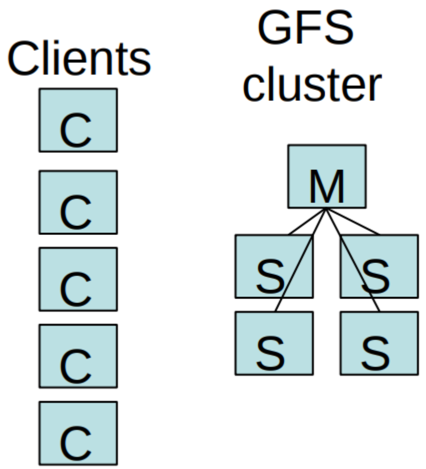
- Files are huge by traditional standards
  - Multi-GB files are common
- Most file updates are appends
  - Random writes are practically nonexistent
  - Many files are written once, and read sequentially
- High bandwidth is more important than latency
- Lots of concurrent data accessing
  - E.g., multiple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

GFS Interface

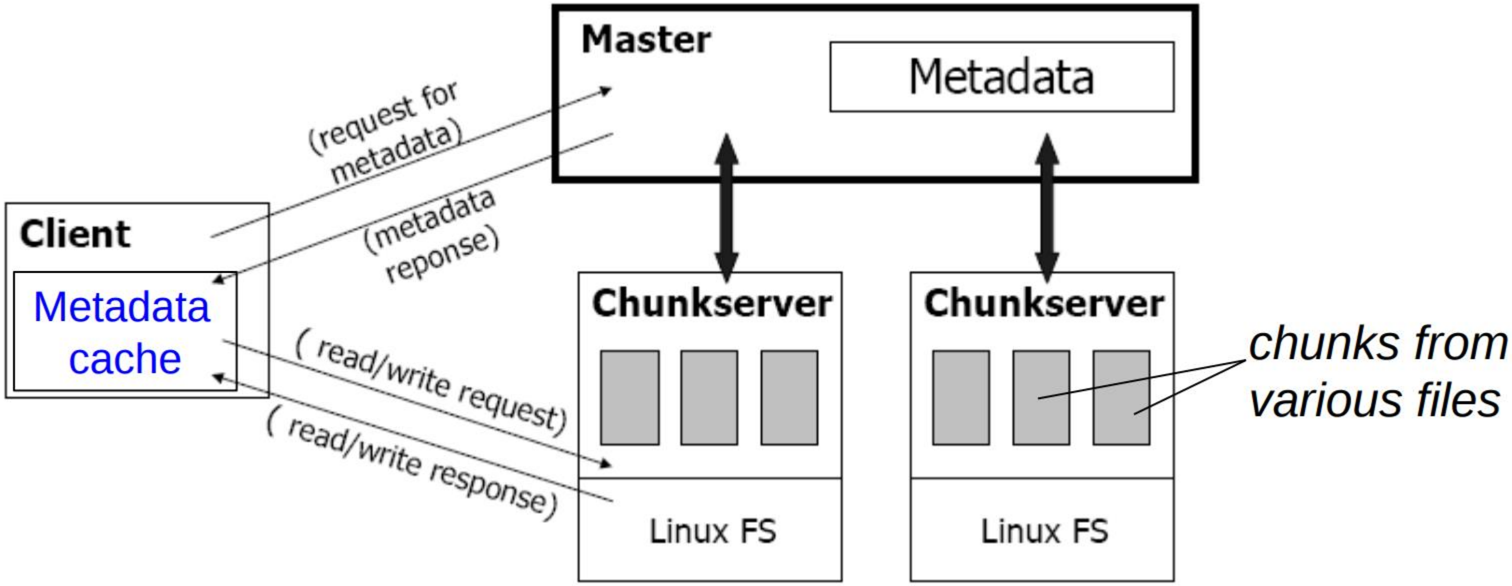
- Not POSIX compliant
  - Supports only popular FS operations, and semantics are different
  - That means you wouldn't be able to mount it
- Additional operation: record append
  - Frequent operation at Google:
  - Merging results from multiple machines in one file (Map/Reduce)
  - Using file as producer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

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)

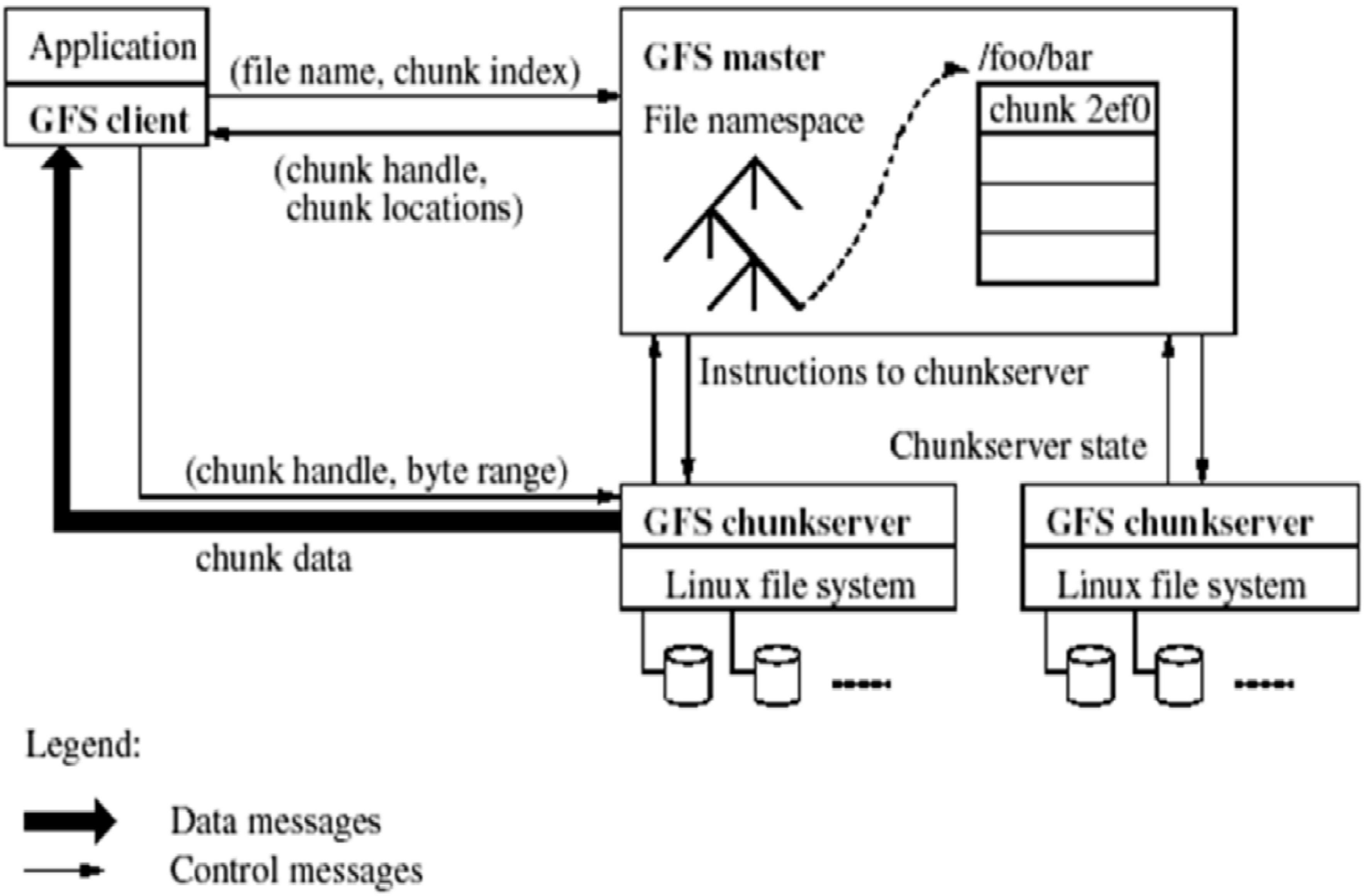


GFS Basic Functioning



- 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

- 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

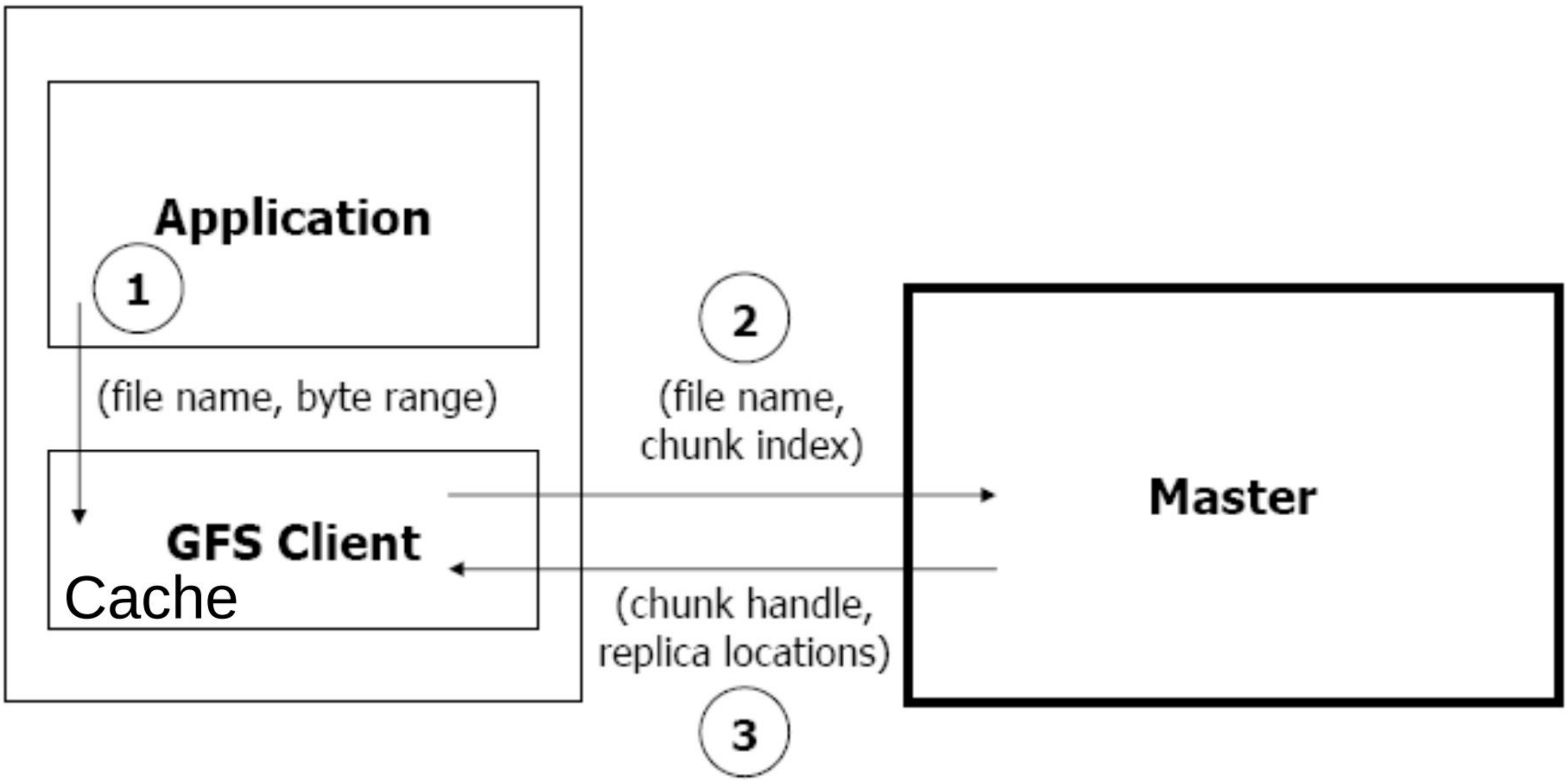
Chunk Locations

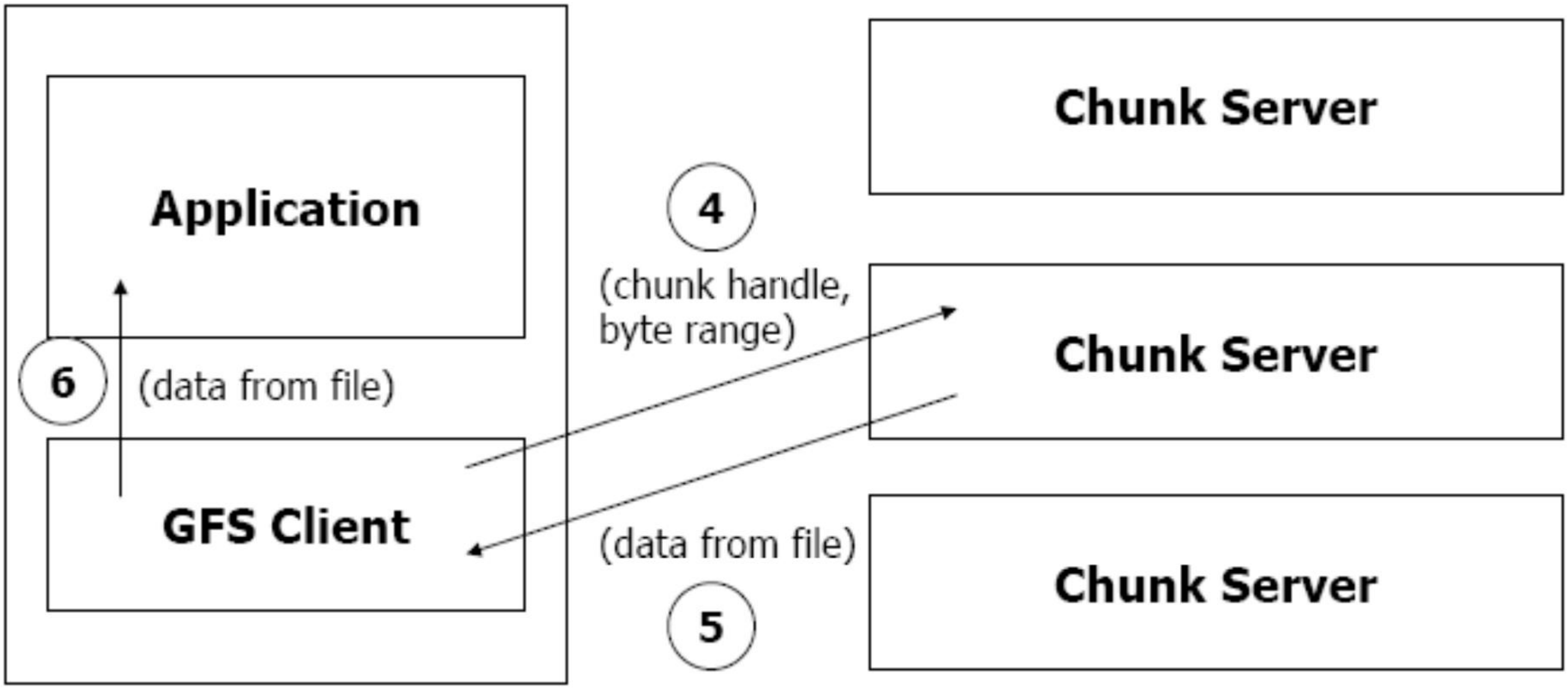
- Kept in memory, no persistent states
  - Master polls chunkservers at startup
- What does this imply?
  - Upsides: master can restart and recover chunks from chunkservers
    - Note that the hierarchical file namespace is kept on durable storage in the master
  - Downside: restarting master takes a 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 chunkserver down?
  - Are there disk failures on chunkserver?
  - Are any replicas corrupted?
  - Which chunks does chunkserver store?
- Master sends instructions to chunkserver:
  - Delete a chunk
  - Create new chunk
  - Replicate and start serving this chunk (chunk migration)
    - Why do we need migration support?

GFS Operations – Read





Application
6
(data from file)
GFS Client
4
(chunk handle,
byte range)
(data from file)
5
Chunk Server
Chunk Server
Chunk Server

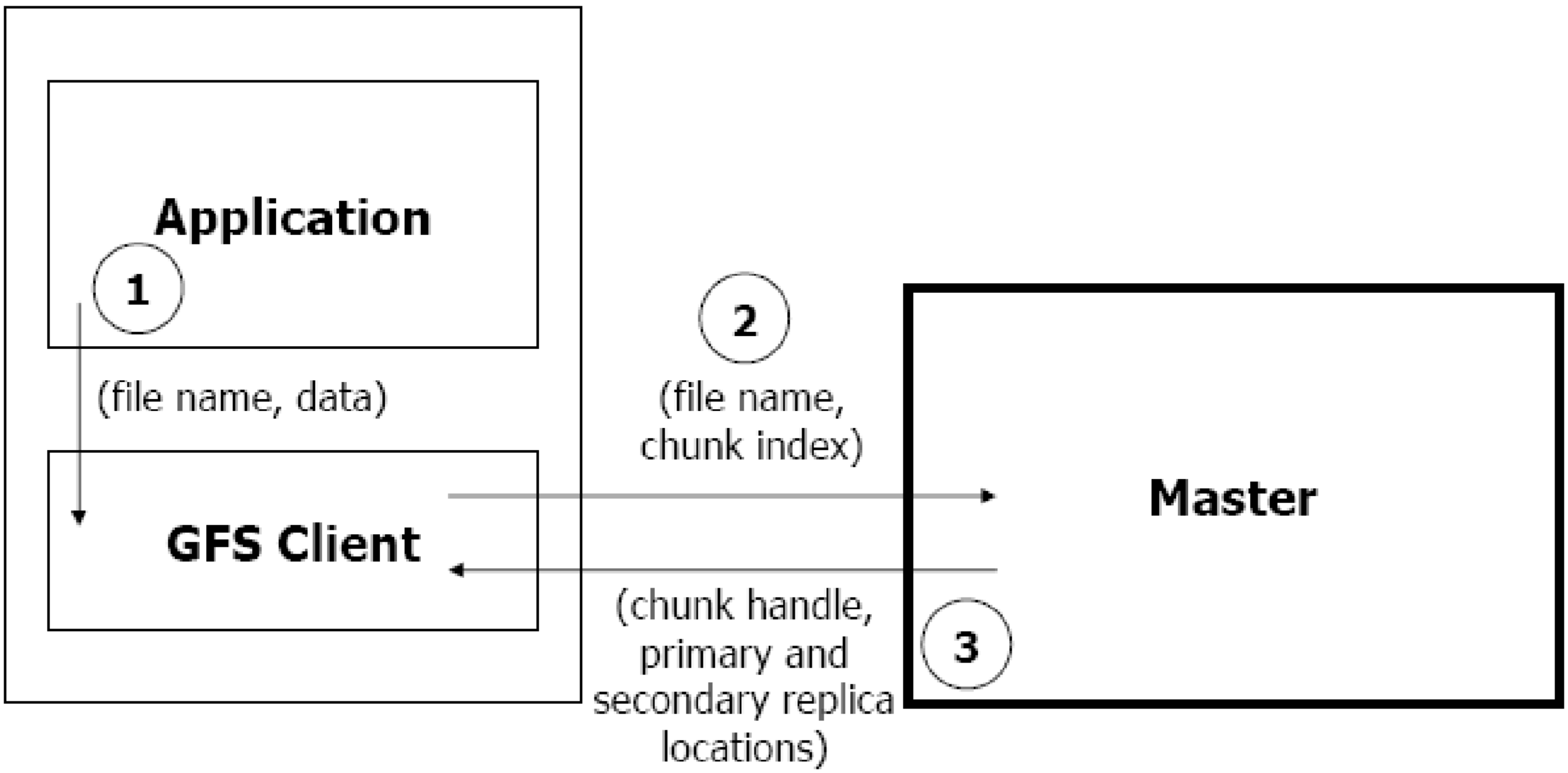
GFS Operations – Updates

- 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

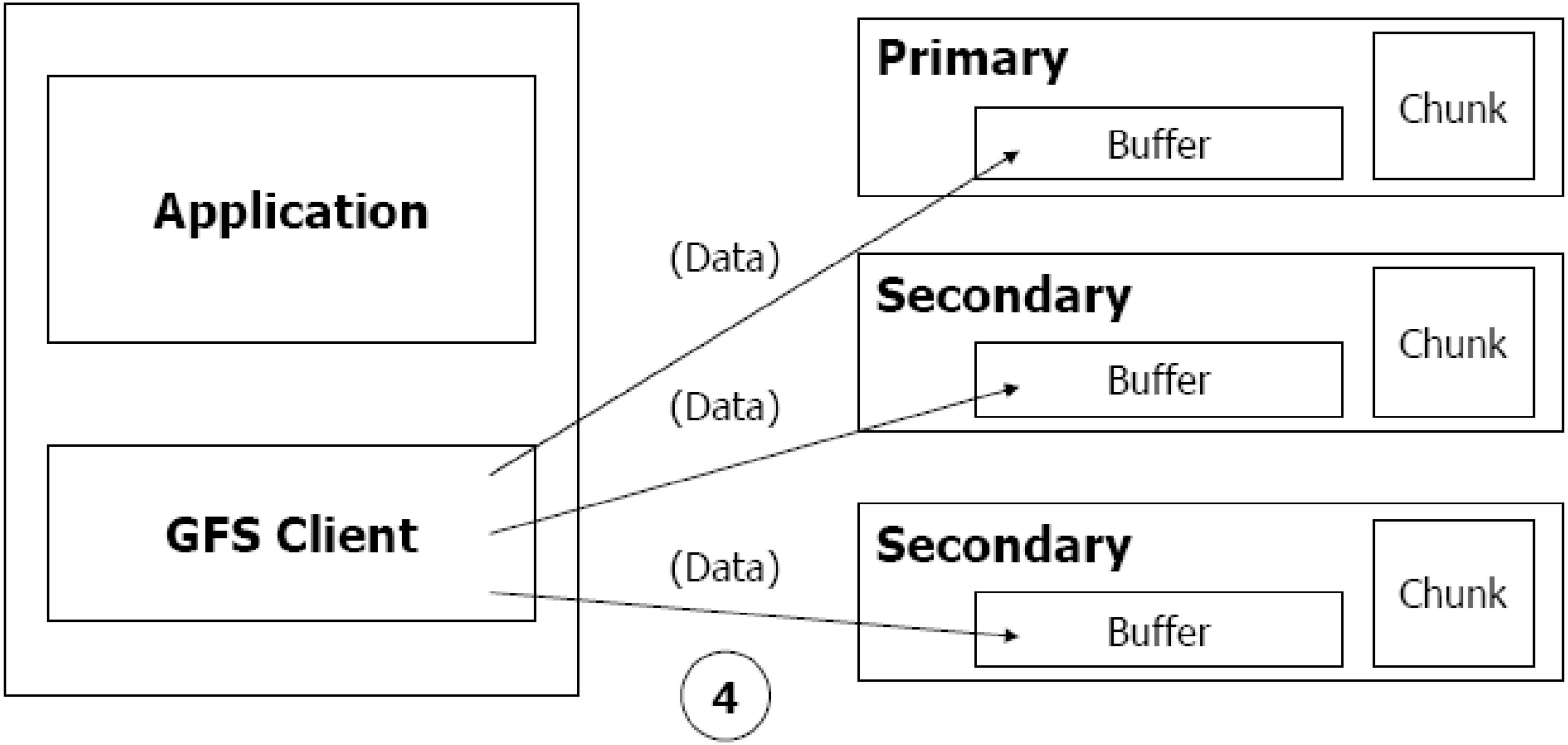
Primaries

- 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, he stops being a primary
- If master doesn't hear from primary chunkserver for a period, he gives the lease to someone else
- So, at any time, at most one server is primary for each chunk
  - But different servers can be primaries for different chunks.

Write

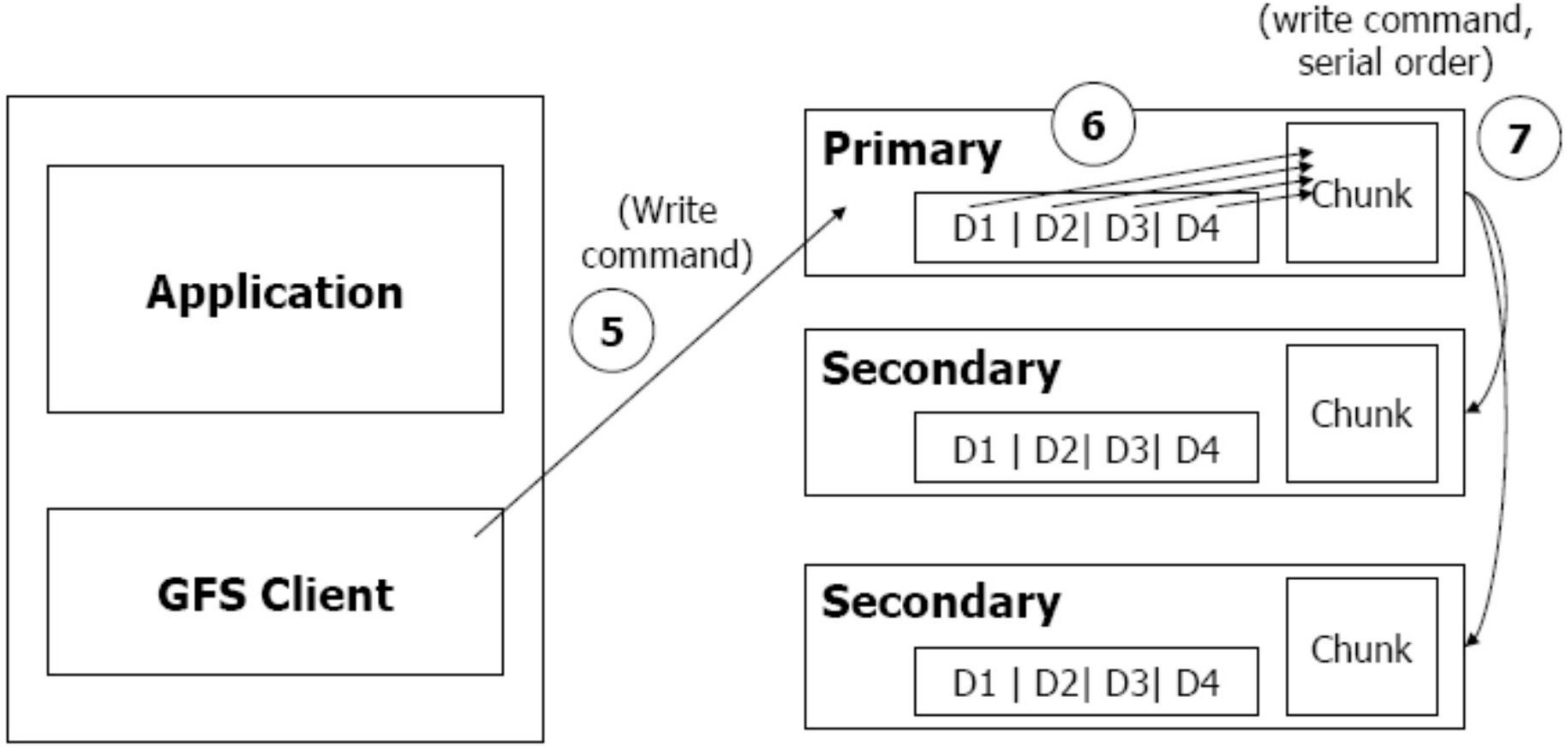

Application
1
(file name, data)
GFS Client
2
(file name, chunk index)
Master
(chunk handle, primary and secondary replica locations)
3

Write



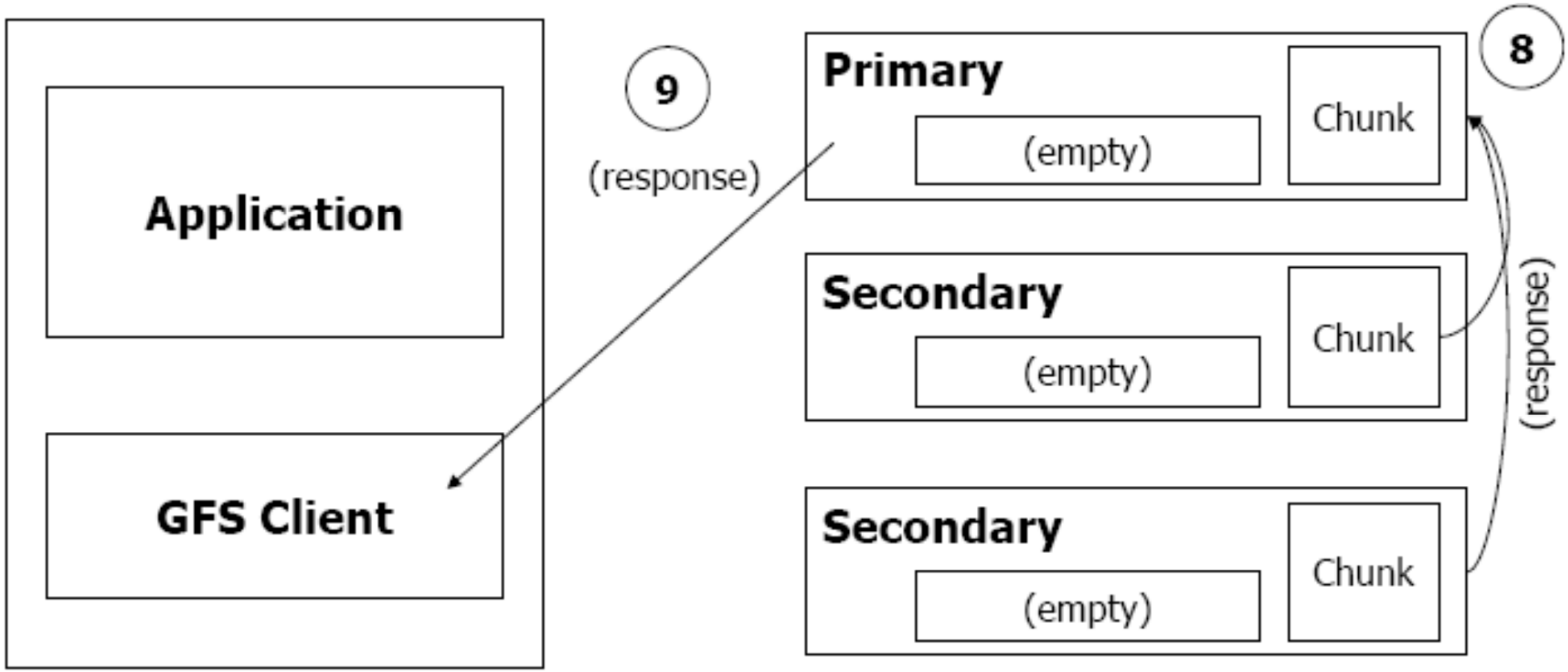
Application
GFS Client
(Data)
(Data)
(Data)
4
Primary
Buffer
Chunk
Secondary
Buffer
Chunk
Secondary
Buffer
Chunk

Write



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

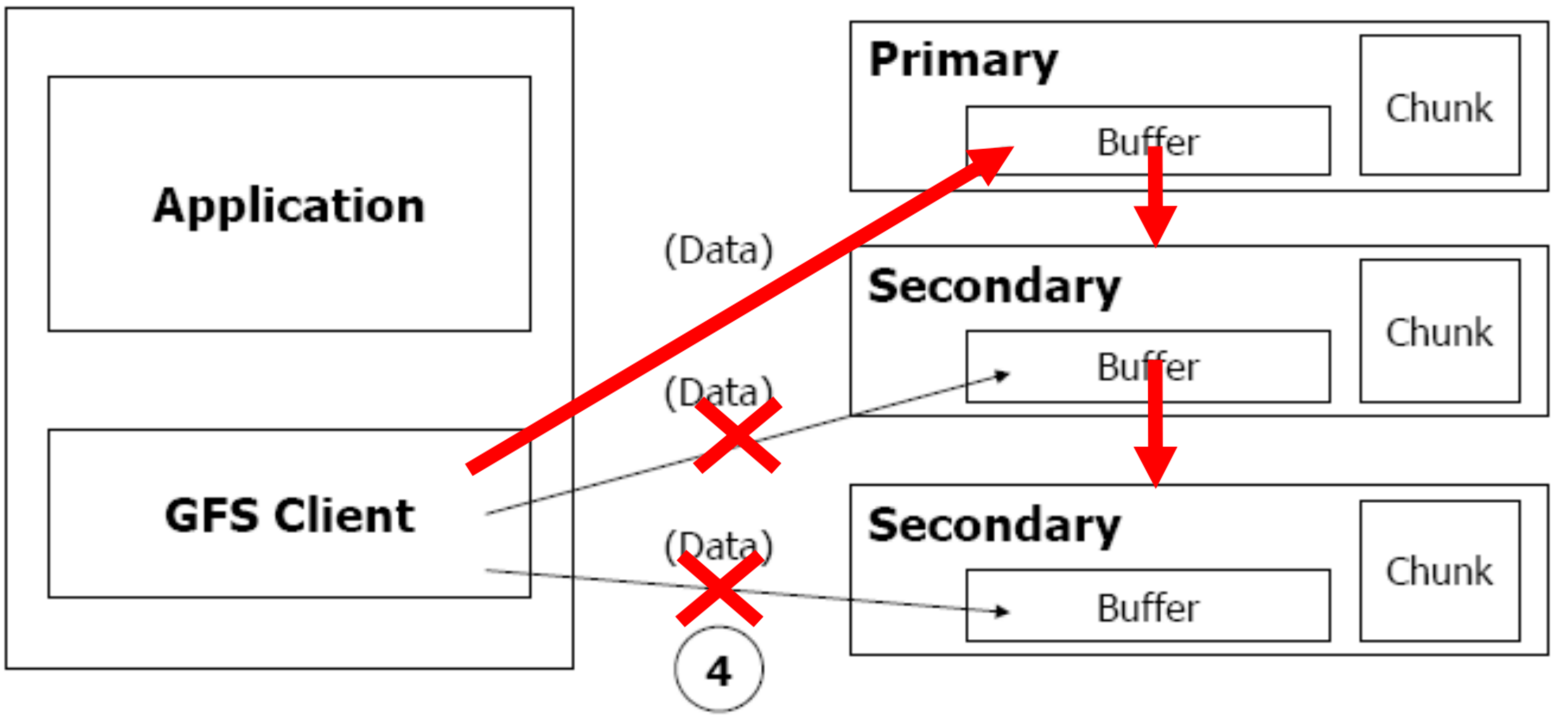
Write


Application
GFS Client
9
(response)
Primary
(empty)
Chunk
8
Secondary
(empty)
Chunk
Secondary
(empty)
Chunk
(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 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 Re: where to place data, it can get atomicity without sacrificing concurrency

Record Append Steps

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.
5. Primary checks if record fits in specified chunk.
6. I f 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 at some offset in chunk,
   - tells secondaries to do the same (specifies offset),
   - receives responses from secondaries,
   - and sends final response to the client.

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