Distributed Systems

Lec 10: Distributed File Systems – GFS

Slide acks:

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Distributed File Systems

- NFS
- AFS
- GFS

- Some themes in these classes:
  - Workload-oriented design
  - Tradeoff between consistency and scalability

NFS (Reminder)

- NFS provides transparent, remote file access
- Simple, portable, *really popular*
  - (it’s gotten a little more complex over time)
- Weak consistency semantics due to caching in client/server RAMs
- Requires hefty server resources to scale
  - Server queried for lots of operations
- CORRECTION for last time: NFS v4+ does support file locking!

AFS

- NFS limitations
  - Doesn't scale well (one server hit with frequent requests)
  - Is very sensitive to network latency

- How to address these?
  - More aggressive caching (AFS caches on disk in addition to RAM)
  - Prefetching (on open, AFS gets entire file from server)
    - When would this be more efficient than prefetching N blocks?
    - With traditional hard drives, large sequential reads are much faster than small random reads. So it's more efficient to do { client A: read whole file; client B: read whole file } than having them alternate. Improves scalability, particularly if client is going to read whole file anyway eventually.

How to Cope with That Caching?

- Close-to-open consistency only
  - Once file closed, updates start getting pushed, but are forced to the server when client re-opens file
  - 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 for improved consistency
- What if server crashes?
  - Reconstruct: Ask all clients “dude, what files you got?”

AFS Summary

- Lower server load than NFS
  - Full-files cached on clients
  - Callbacks: server does nothing for read-only files (common case)

- But may be slower: Access from local disk is much slower than from another machine’s memory over a LAN

- For both AFS and NFS, central server is:
  - Bottleneck: reads / writes hit it at least once per file use
  - Single point of failure
  - Expensive: to make server fast, beefy, and reliable, you need to pay $$$

Distributed File Systems

- NFS
- AFS
- GFS
  - We’ll use the GFS SOSP ‘2003 talk
  - That means that some of the facts may be out-of-date

GFS Overview

- 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 writes
  - At least once, atomic record appends

GFS Design Constraints

1. Machine failures are the norm
   - 1000s of components
   - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
   - Monitoring, error detection, fault tolerance, automatic recovery must be integral parts of a design

2. Design for big-data workloads
   - Search, ads, Web analytics, Map/Reduce, …

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' design is geared toward apps' characteristics
  - And Google apps have been geared toward GFS

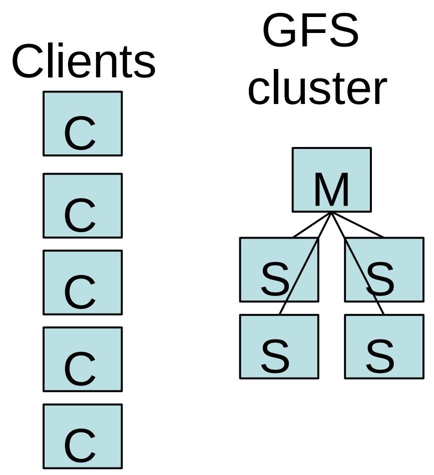
Interface Design

- 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

Architectural Design

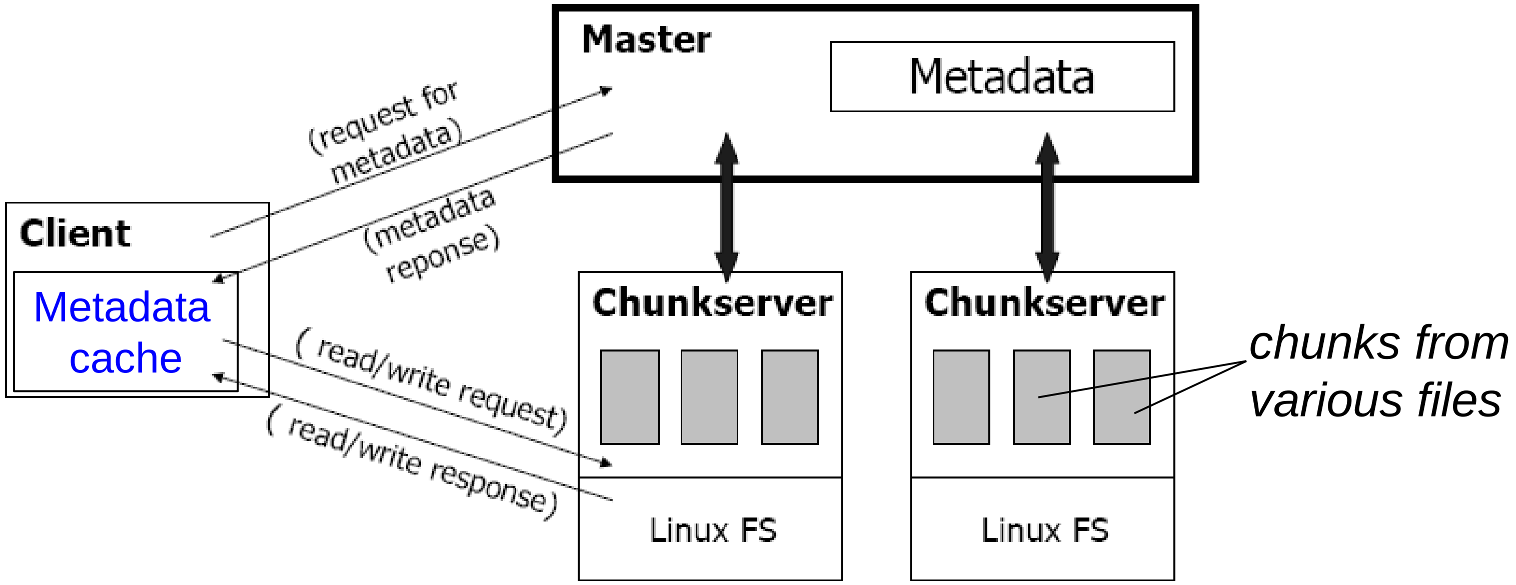
- A GFS cluster
  - A single master
  - Many chunkservers
    - Accessed by many *clients*



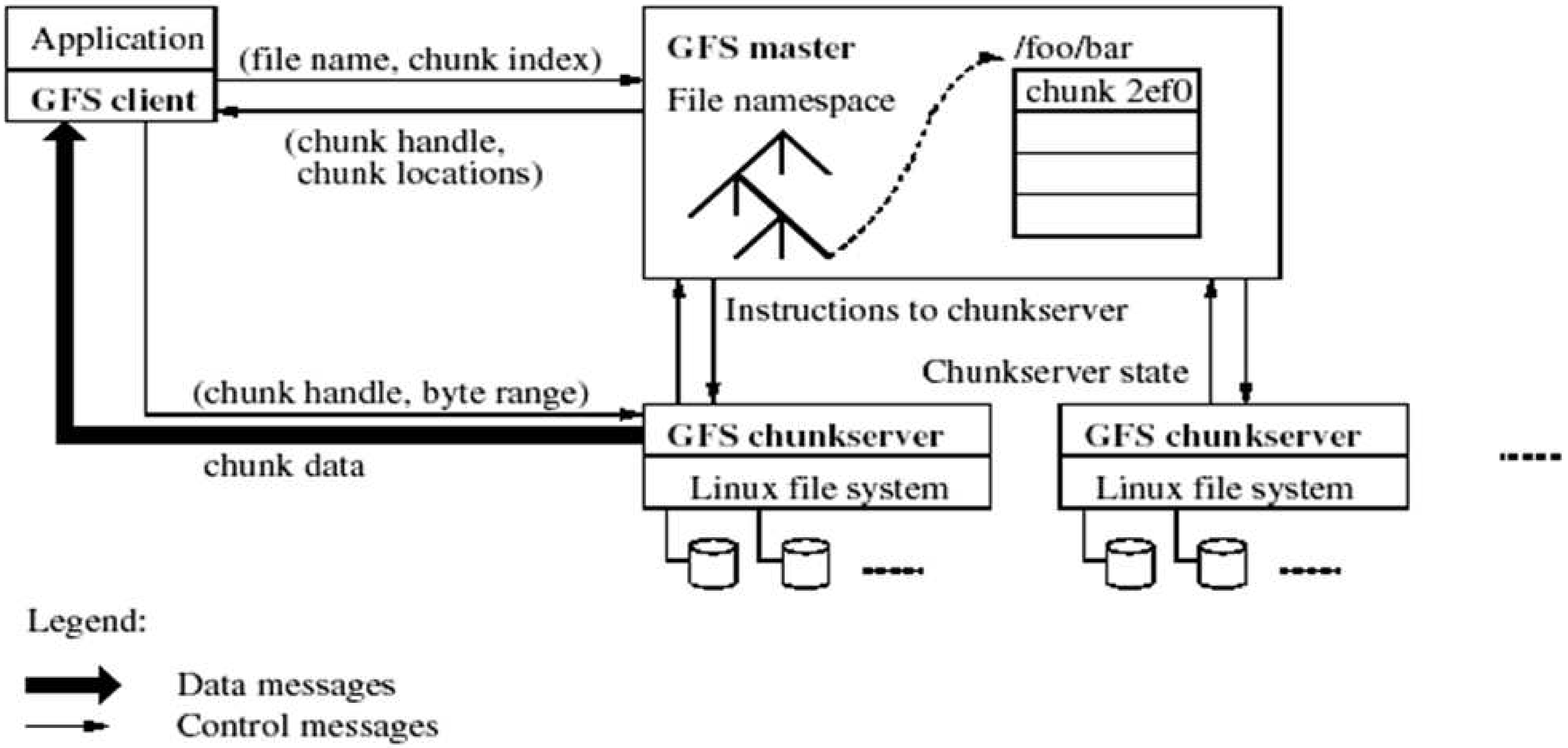
- A file
  - Divided into fixed-sized chunks (similar to FS blocks)
    - Labeled with 64-bit unique global IDs (called *handles*)
    - Stored at chunkservers
    - 3-way replicated across chunkservers
    - Master keeps track of metadata (e.g., which chunks belong to which files)

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



Detailed Architecture



Chunks

- Analogous to FS blocks, except larger
- Size: 64 MB!
  - Normal FS block sizes are 512B - 8KB

- Pros of big chunk sizes?
- Cons of big chunk sizes?

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
  - The replicas use Paxos, an algorithm that we'll talk about later to keep coordinate, i.e., to act as one (think of the voting algorithm we looked at for distributed mutual exclusion)

- Stores all metadata
  1) File and chunk namespaces
     - Hierarchical namespace for files, flat namespace for chunks
  2) File-to-chunk mappings
  3) Locations of a chunk’s replicas

Chunk Locations

- Kept in memory, no persistent states
  - Master polls chunkservers at startup
- What does this imply?
  - Upsides
  - Downsides

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?

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

Master Controls/Coordinates Chunkservers

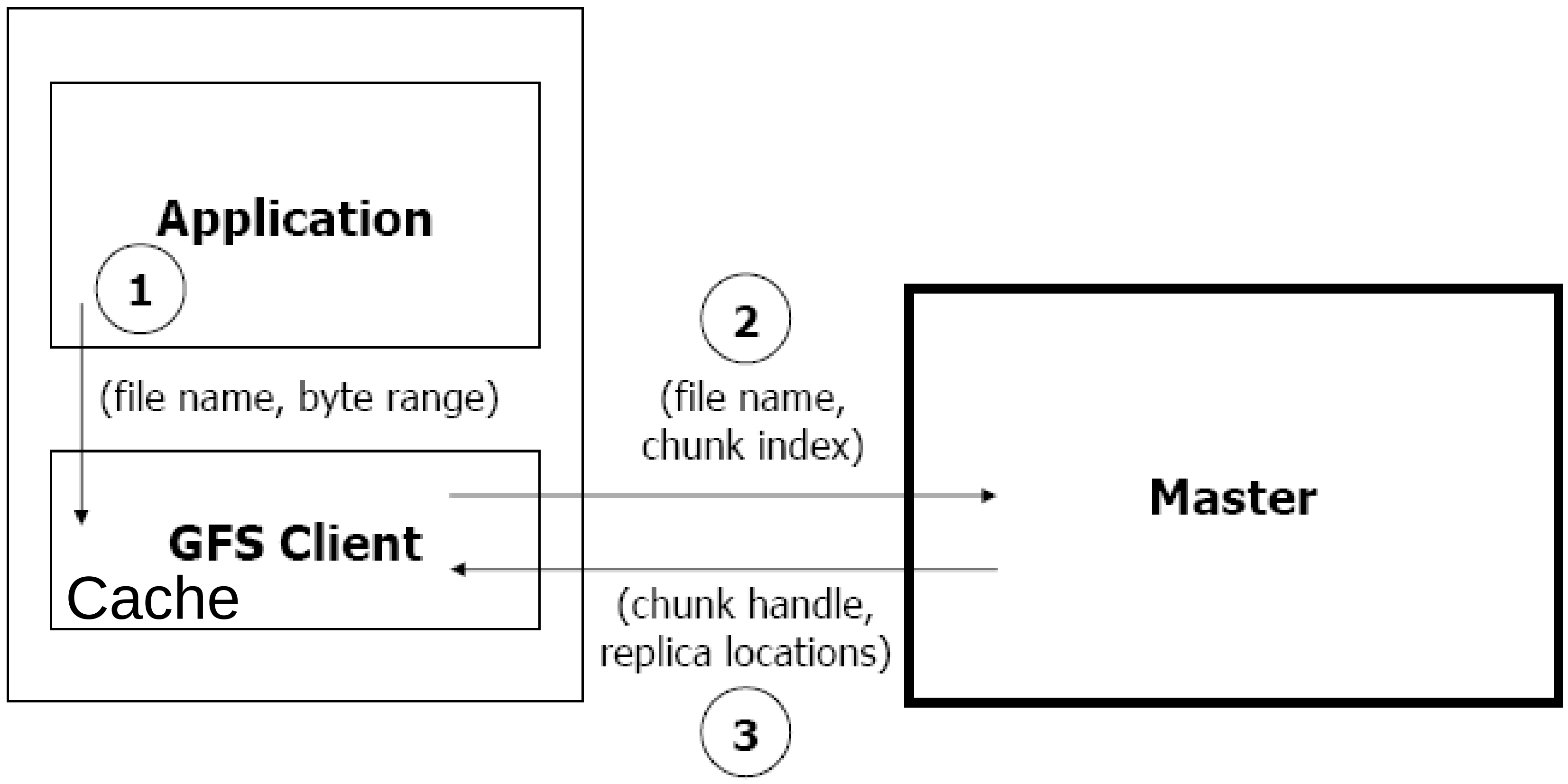
- Master and chunkserver communicate regularly (heartbeat):
  - Is chunkserver down?
  - Are there disk failures on chunkserver?
  - Are any replicas corrupted?
  - Which chunks does chunkserver store?

- 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?

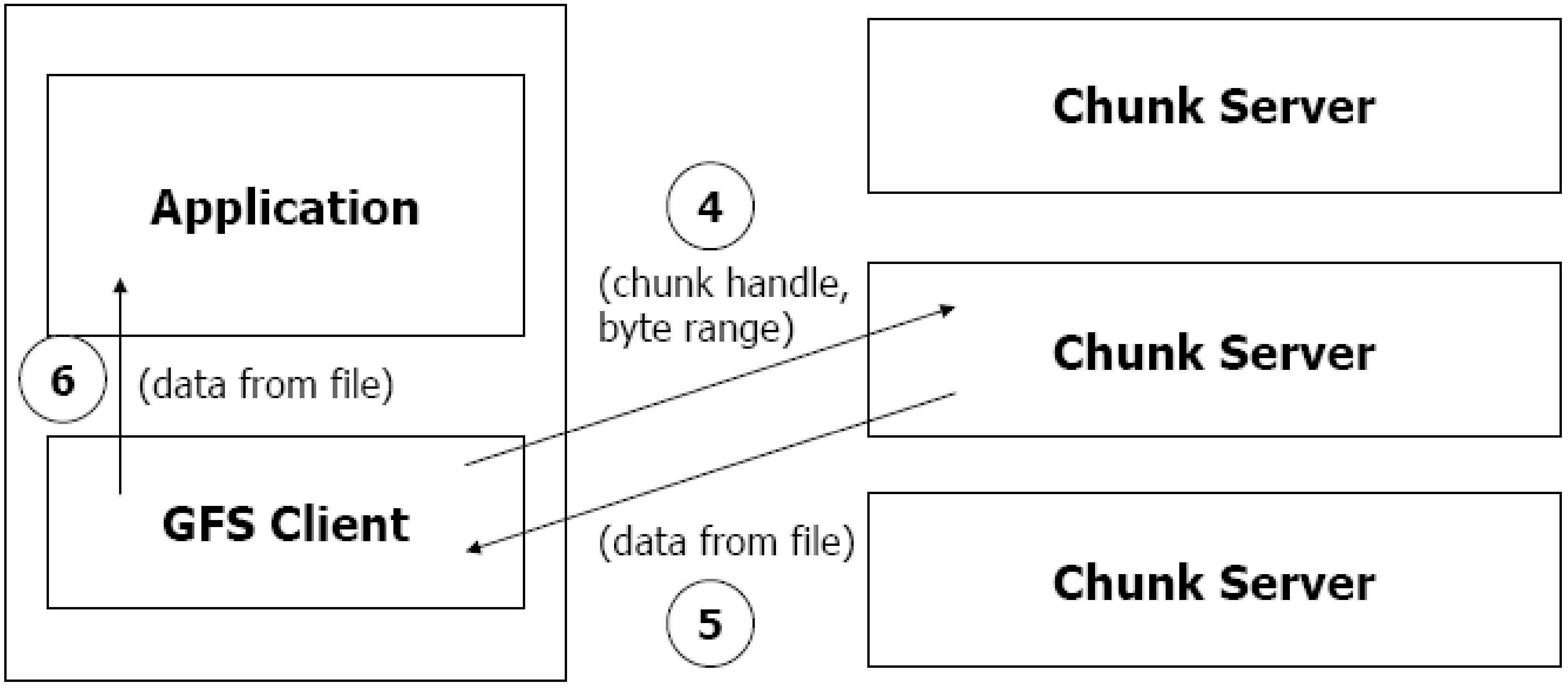
File System Operations

- Reads
- Updates:
  - Writes
  - Record appends

Read Protocol



Read Protocol



Updates

- Writes:
  - Cause data to be written at application-specified file offset
- Record appends:
  - Operations that append data to a file
  - Cause data to be appended atomically at least once
  - Offset chosen by GFS, not by the client
- Goals:
  - Clients can read, write, append records at max throughput and in parallel
  - Some consistency that we can understand (kinda)
  - Hence, favor concurrency / performance over semantics
    - Compare to AFS/NFS' design goals

Update Order

- 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

Primary Leases

- 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

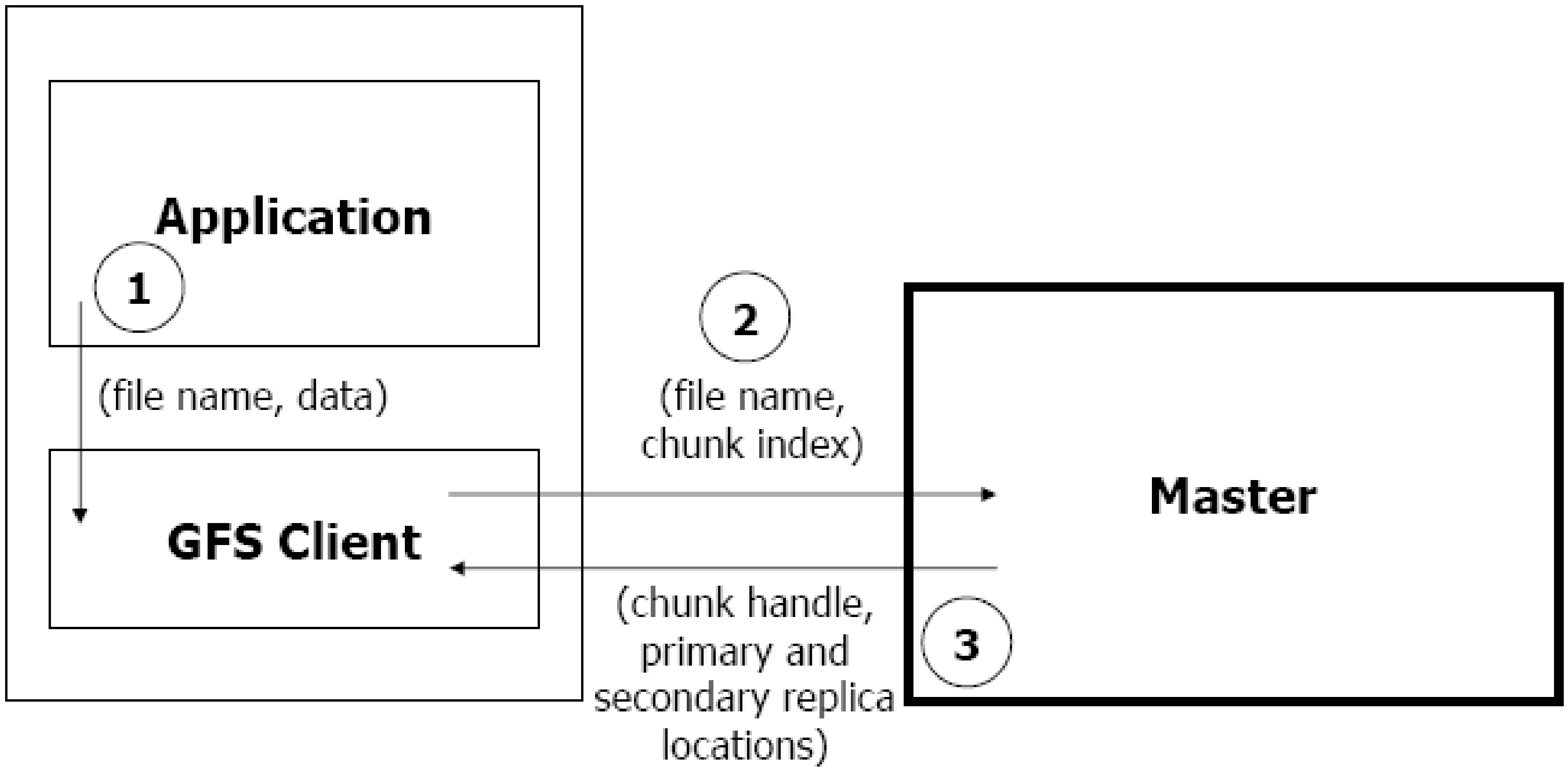
Primary Leases

- 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

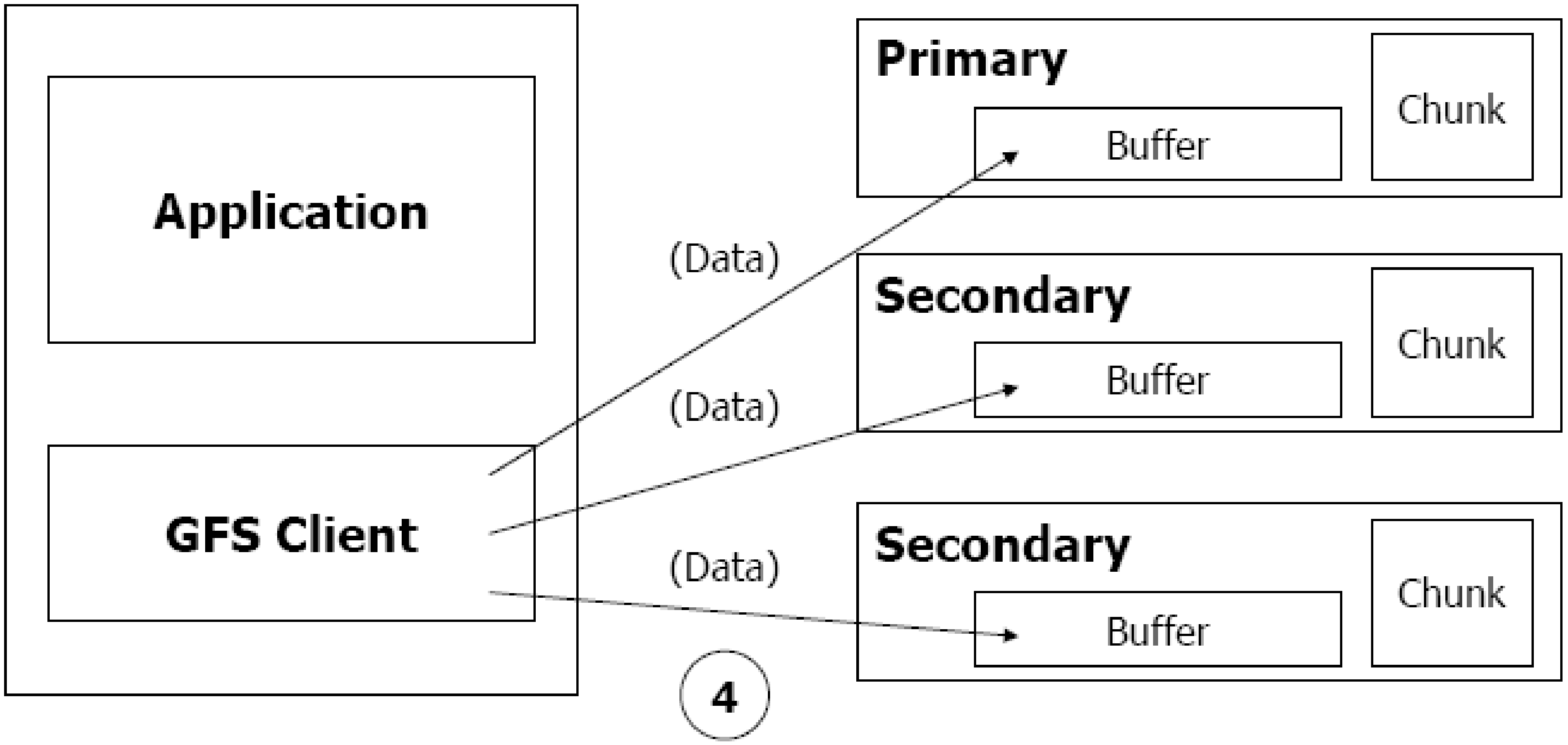
Primary Leases

- 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

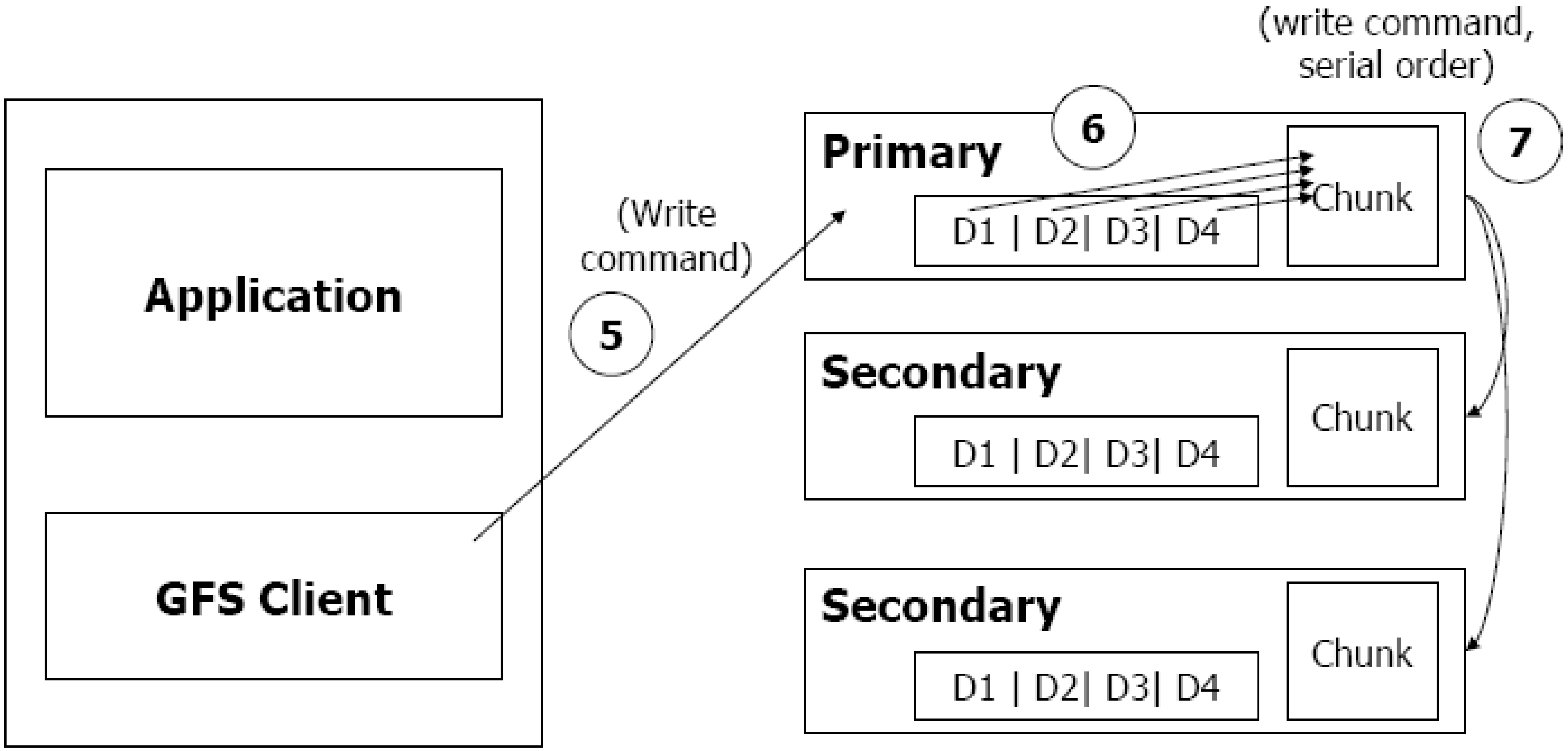
Write Algorithm



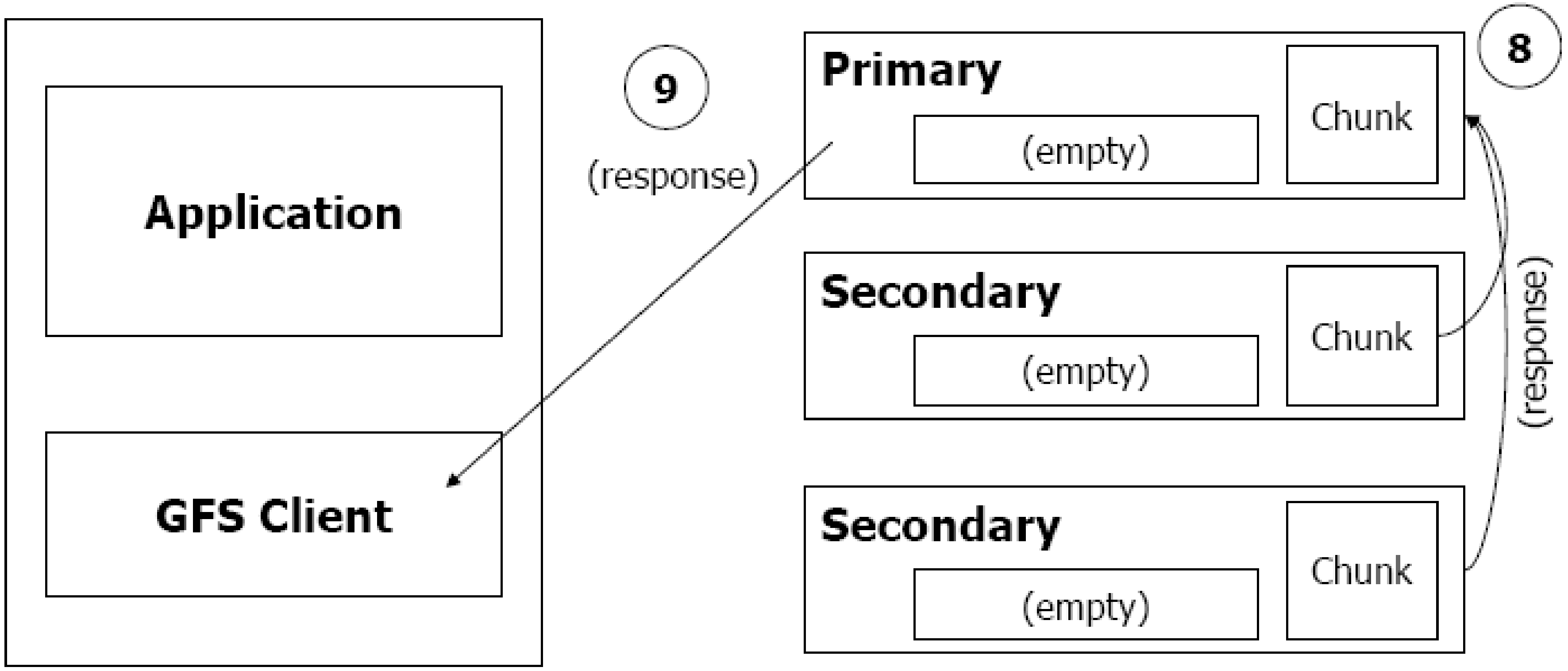
Write Algorithm



Write Algorithm



Write Algorithm



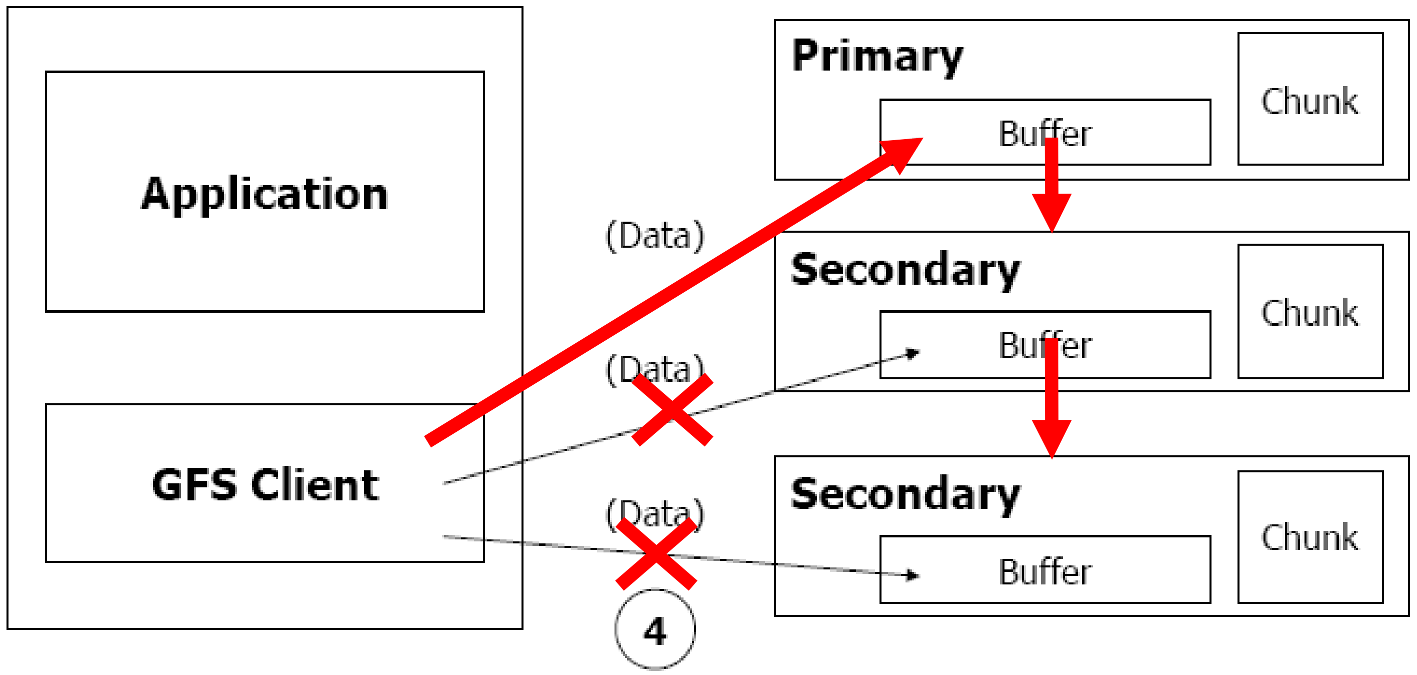
Write Consistency

- 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

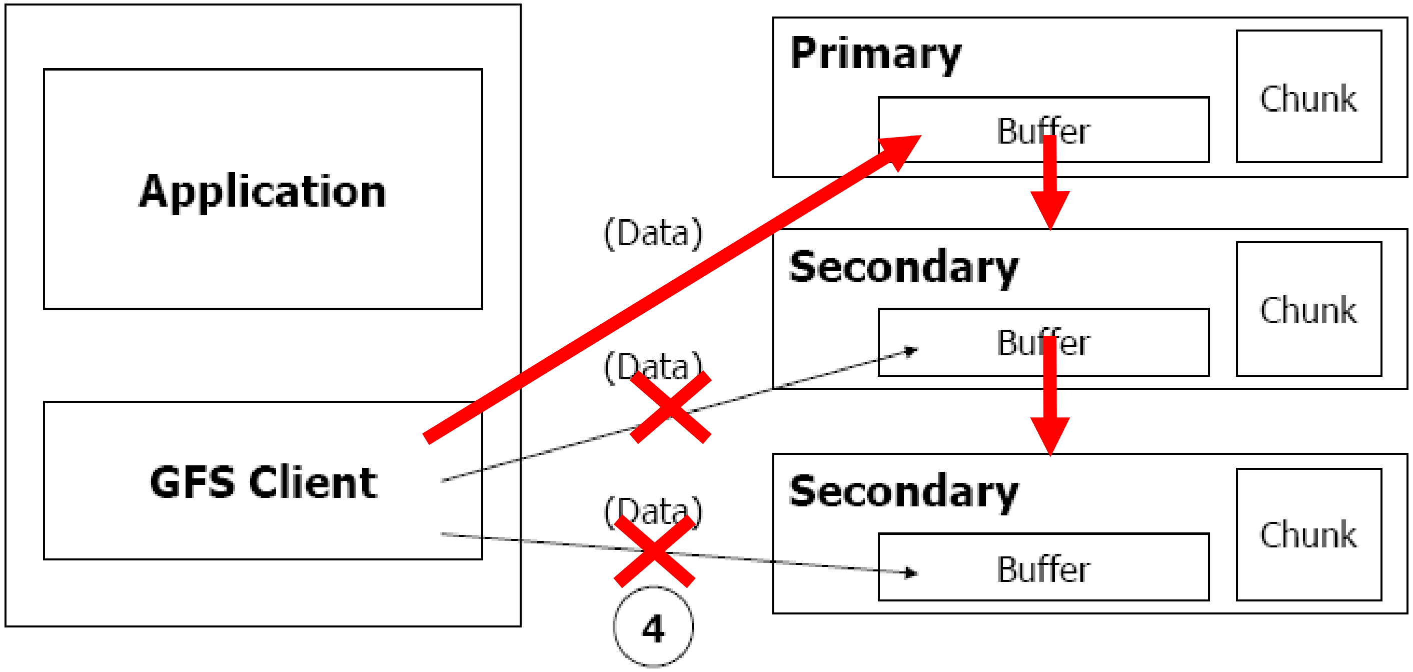
One Little Correction

- 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?



One Little Correction

- 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 Appends

- 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 Appends

- 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
- Rough mechanism:
  - If record fits in chunk, primary chooses the offset and communicates it to all replicas → *offset is arbitrary*
  - If record doesn't fit in chunk, the chunk is padded and client gets failure → *file may have blank spaces*
  - If a record append fails at any replica, the client retries the operation → *file may contain record duplicates*

Record Append Algorithm

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. 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 at some offset in chunk,
    - tells secondaries to do the same (specifies offset),
    - receives responses from secondaries,
    - and sends final response to the client.

Implications of Weak Consistency for Applications

- Relying on appends rather on overwrites
- Writing self-validating records
  - Checksums to detect and remove *padding*
- Self-identifying records
  - Unique Identifiers to identify and discard *duplicates*
- Hence, applications need to adapt to GFS and be aware of its inconsistent semantics
  - We'll talk soon about several consistency models, which make things easier/harder to build apps against

GFS Summary

- 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

AFS/NFS/GFS Lessons

- Distributed (file)systems always involve a tradeoff: consistency, performance, scalability.

- We've learned a lot since AFS/NFS/GFS, but the general lesson holds. *Especially* in the wide-area.
  - We'll see a related tradeoff, also involving consistency, in a while: the CAP tradeoff (Consistency, Availability, Partition-resilience)

More Lessons

- Client-side caching is a fundamental technique to improve scalability and performance
  - But raises important questions of cache consistency
- Periodic refreshes, invalidations, leases are common methods for providing (some forms of) consistency
  - They solve different problems, though
- Often times one must define one's own consistency model & operations, informed by target workloads
  - AFS provides close-to-open semantic, and, they argue, that's what users care about when they don't share data
  - GFS provides atomic-at-leas-once record appends, and that's what some big-data apps care about

Next Time

- We’ll start talking about standard consistency models
- We’ll analyze the models of some of these file systems against the standard models

GFS Slide Refs/Acks

- http://www.cs.fsu.edu/~awang/courses/cis6935_s2004/google.ppt
- http://ssrnet.snu.ac.kr/course/os2008-2/GFS-2008.ppt