Network File System

CS 202: Advanced Operating Systems

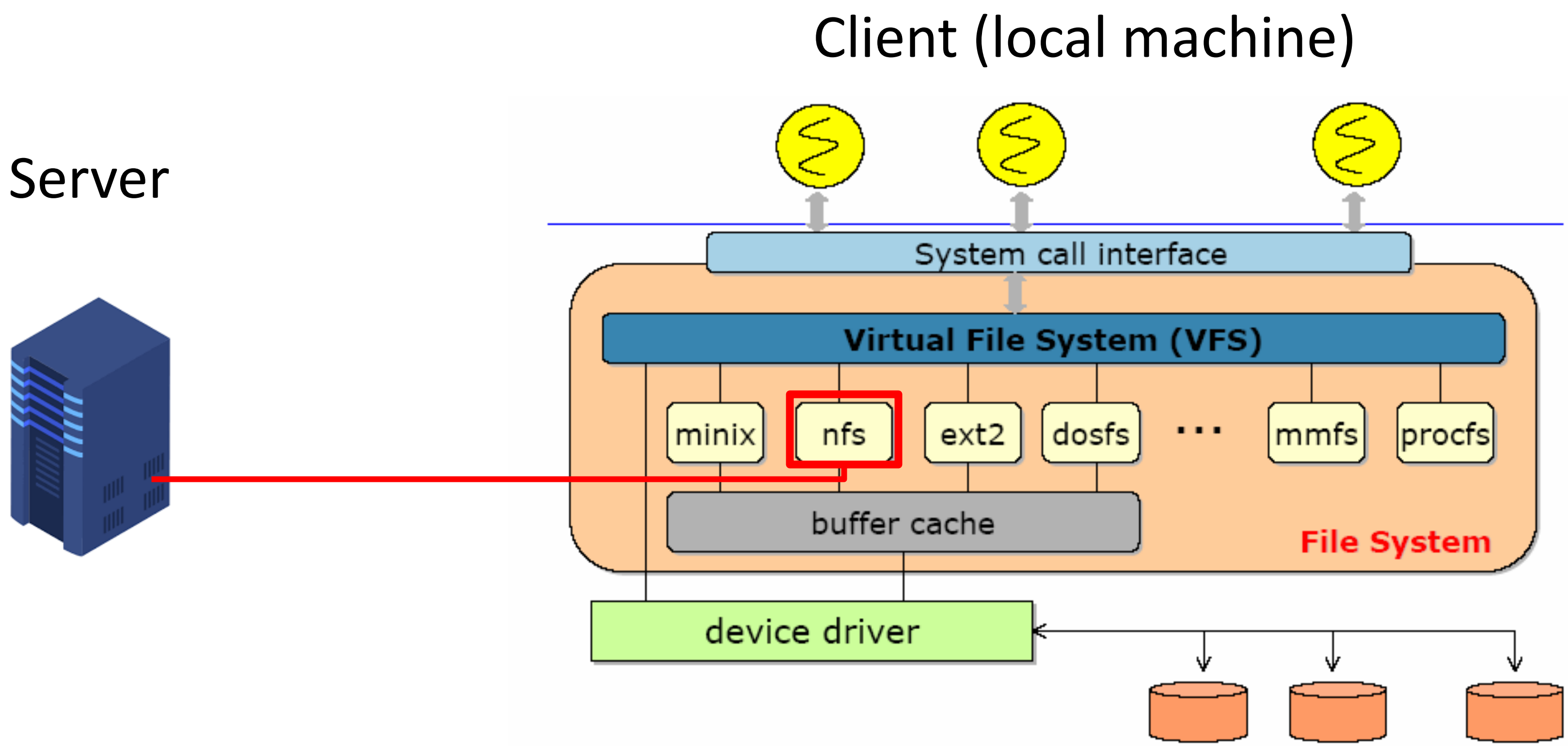
Recap: FFS & LFS

- FFS (Fast File System)
  - Cylinder groups: improve data placement locality
  - Larger block size: support larger files / improve bandwidth
  - Sub-blocks: address internal fragmentation (extra I/O; optional)

- LFS (Log-based File System)
  - Buffer all updates
  - Write them sequentially to the disk with fewer # of write operations
    - Good for writing, but not good for reading (data is scattered; bad for sequential reads)
  - Inode map & checkpoint region
  - Segment cleaning/garbage collection

- FFS optimizes for locality, LFS optimizes for writes

Network File System



- Goal: allow a computer to access files over a network as if they were on a local disk
  - Provide a standard way for clients to access files on remote servers, making it easier for clients to share files across a local network

Intuition and Challenges

- Translate VFS requests into remote procedure calls to server
  - Instead of translating them into local disk accesses

- Challenges:
  - Server can crash or be disconnected
  - Client can crash or be disconnected
  - How to coordinate multiple clients on same file?

Protocols Design Choices

- **Stateful protocol**: Server keeps track of past requests
  - Server maintains client states (e.g., file lock)
- **Stateless protocol**: Server does **not** keep track of past requests
  - Client should send all necessary state with a single request

Protocols Design Choices

- **Stateful protocol**: Server keeps track of past requests
  - Server maintains client states (e.g., file lock)
- **Stateless protocol**: Server does **not** keep track of past requests
  - Client should send all necessary state with a single request

- Challenge of stateful: <u>Recovery</u> from crash/disconnect
  - Server side challenges:
    - Knowing when a connection has failed (timeout)
    - Tracking state that needs to be cleaned up on a failure
  - Client side challenges:
    - If server thinks we failed (timeout), must recreate server state (e.g., reconnect)
- Drawbacks of stateless:
  - Complex procedure messages; <u>more messages</u> in general

Earlier versions of NFS are Stateless

- Each procedure call contains all the information necessary to complete the call
  - User credentials (for security checking)
  - File handle and offset (explained later)
  - Server maintains no "between call" information
- Each request matches a VFS operation
  - e.g., lookup, read, write, unlink, stat
  - there is no open or close among NFS operations
- Default NFS transport protocol (up to NFSv3) was UDP

Advantages of statelessness

- Crash recovery is very easy:
  - When a server crashes, client just resends request until it gets an answer from the rebooted server
  - Client cannot tell difference between a server that has crashed and recovered and a slow server

- Client can always **repeat** any request

Consequences of statelessness

- Read and writes must specify their start offset
  - Server does not keep track of current position in the file
  - User still use conventional UNIX reads and writes

Consequences of statelessness

- Read and writes must specify their start <u>offset</u>
  - Server does not keep track of current position in the file
  - User still use conventional UNIX reads and writes
- Open() system call translates into several LOOKUP calls to server
- No NFS equivalent to UNIX close() system call

How does NFS handle a file?

- Client side translates user requests to protocol messages to implement the request remotely

Client	Server
fd = open("/foo", ...);	
Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request
	look for "foo" in root dir
	return foo's FH + attributes (File Handle)
Receive LOOKUP reply	
allocate file desc in open file table	
store foo's FH in table	
store current file position (0)	
return file descriptor to application	

File Lookup

- One single open call such as:

  fd = open("/home/foo/bar.txt")

  will be result in several calls to lookup

  lookup(rootfh, "home") returns (fh0, attr)
  lookup(fh0, "foo") returns (fh1, attr)
  lookup(fh1, "bar.txt") returns (fh2, attr)

- Why all these steps?

File Lookup

- One single open call such as:

  fd = open("/home/foo/bar.txt")

  will be result in several calls to lookup

  lookup(rootfh, "home") returns (fh0, attr)
  lookup(fh0, "foo") returns (fh1, attr)
  lookup(fh1, "bar.txt") returns (fh2, attr)

- Why all these steps?
  - Any of components of /home/foo/bar.txt could be a different mount point → similar to path lookups in local file system

File Lookup

- One single open call such as:

  fd = open("/home/foo/bar.txt")

  will be result in several calls to lookup

  lookup(rootfh, "home") returns (fh0, attr)
  lookup(fh0, "foo") returns (fh1, attr)
  lookup(fh1, "bar.txt") returns (fh2, attr)

- Why all these steps?
  - Any of components of /home/foo/bar.txt could be a different mount point → similar to path lookups in local file system
- Once a file handle is available, the client can issue READ and WRITE messages on a file with the offset in the file
  - File handle tells the server which volume and which inode to read from

Caching (delegation)

- NFS operations are expensive
  - Lots of network round-trips
  - NFS server is a user-space daemon

- With **caching on the clients**
  - Only the first reference needs network communication
  - Later requests can be satisfied in local memory

Challenge: Caches and Consistency

1. Clients A and B have file in their cache
2. Client A writes to the file
   - Data stays in A's cache
   - Eventually flushed to the server
3. Client B reads the file
   - Does B see the old contents or the new file contents?
   - Who tells B that the cache is stale?

Challenge: Caches and Consistency

1. Clients A and B have file in their cache
2. Client A writes to the file
   - Data stays in A's cache
   - Eventually flushed to the server
3. Client B reads the file
   - Does B see the old contents or the new file contents?
   - Who tells B that the cache is stale?

- Stateful protocol: Server can tell, but only after A actually wrote the data

Challenge: Caches and Consistency

1. Clients A and B have file in their cache
2. Client A writes to the file
   - Data stays in A's cache
   - Eventually flushed to the server
3. Client B reads the file
   - Does B see the old contents or the new file contents?
   - Who tells B that the cache is stale?

- Stateful protocol: Server can tell, but only after A actually wrote the data
- Stateless protocol: Server does not know which clients are accessing the file
  - Clients do not know either

Consistency/Performance Tradeoff

- Performance: cache always, write later when convenient
  - Other clients can see old data, or make conflicting updates
- Consistency: write everything immediately
  - And tell everyone who may have it cached
    - Requires server to know the clients which cache the file (stateful)
  - Much more network traffic, lower performance
  - Not good for the common case: accessing an unshared file
- So, how does NFS do?

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close

- Server implements a write-through policy
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close

- Server implements a write-through policy
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close
- Server implements a write-through policy
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes
- Clients must

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close
- Server implements a write-through policy
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes
- Clients must
  - Frequently send their modified blocks to the server

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close
- Server implements a <u>write-through</u> policy
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes
- Clients must
  - Frequently send their modified blocks to the server
  - Frequently ask the server to revalidate the blocks they have in their cache

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close
- Server implements a write-through policy
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes
- Clients must
  - Frequently send their modified blocks to the server
  - Frequently ask the server to revalidate the blocks they have in their cache
  - On open, check the cached version's timestamp

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close
- Server implements a write-through policy
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes
- Clients must
  - Frequently send their modified blocks to the server
  - Frequently ask the server to revalidate the blocks they have in their cache
  - On open, check the cached version's timestamp
    - If stale, invalidate the cache

Close-to-Open Consistency

- NFS Approach: Flush all writes on a close
- Server implements a write-through policy
  - Any blocks modified by a write request (including i-nodes and indirect blocks) must be written back to disk before the call completes
- Clients must
  - Frequently send their modified blocks to the server
  - Frequently ask the server to revalidate the blocks they have in their cache
  - On open, check the cached version's timestamp
    - If stale, invalidate the cache
    - Makes sure you get the latest version on the server when opening a file

Challenge: Lost Request

- Request sent to NFS server, no response received
  - Did the message get lost in the network (UDP)?
  - Did the server die?
  - Is the server slow?
    - Don’t want to do things twice → Bad idea: write data at the end of a file twice

Challenge: Lost Request

- Request sent to NFS server, no response received
  - Did the message get lost in the network (UDP)?
  - Did the server die?
  - Is the server slow?
    - Don't want to do things twice → Bad idea: write data at the end of a file twice
- NFS approach: Make all requests ***idempotent***
  - Requests have same effect when executed multiple times
    - E.g., read request for a file
  - Some requests not easy to make idempotent
    - E.g., deleting a file
    - Server keeps a <u>cache of recent requests</u> and ignores requests found in the cache

Challenge: File Locking

- Must have a way to change a file without collision
  - Imagine multiple clients are trying to update the same file
- Solution: Get a server-side lock
  - What happens if the client dies?
    - Lots of options (timeouts, etc)
  - Not part of NFS protocol (except NFSv4)
  - But available as extra locking services, e.g., Network Lock Manager (NLM)

Challenge: Time Synchronization

- Each system's clock ticks at slightly different rates
  - These clocks can drift over time
- Precise file timestamp is required
  - Consistency check for cached data
  - Tools like 'make' use timestamps

```
make[2]: warning: Clock skew detected. Your build may be
incomplete.
```

- Systems using NFS must have clocks synchronized
  - Using external protocols like Network Time Protocol (NTP)
    - Synchronization depends on unknown communication delay
    - Complex protocol but works pretty well in practice

NFS Evolution & Summary

- The simple protocol was version 2 (also the textbook)
- NFSv3 (1995):
  - 64-bit file sizes and offsets (large file support) & Other optimizations
  - Still widely used today
- NFSv4 (2000):
  - Attempts to address many of the problems of v3
    - Security, Performance; Provides a stateful protocol
  - Much more complicated then v3
- Problems?
  - Mainly for use in local area networks with latency/bandwidth are relatively predictable & consistent (Sort of locally distributed file systems)
  - Single point of failure (centralized architecture); not good for scalability

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure
- Performance

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure
- Performance -> Distributing

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure
- Performance -> Distributing -> Faults

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure
- Performance -> Distributing -> Faults -> Fault Tolerance

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure
- Performance -> Distributing -> Faults -> Fault Tolerance

Replicas <-

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG’s rapidly growing infrastructure
- Performance -> Distributing -> Faults -> Fault Tolerance

Inconsistency <- Replicas <-

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure
- Performance -> Distributing -> Faults -> Fault Tolerance

Consistency <- Inconsistency <- Replicas <-

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure
- Performance -> Distributing -> Faults -> Fault Tolerance

Low <- Consistency <- Inconsistency <- Replicas <- performance

Challenges of building truly distributed file systems

- Systems research is all about optimizing tradeoffs and tensions
- Think about creating a scalable, fault-tolerant, and high performance distributed file system that could meet the demands of FAAMG's rapidly growing infrastructure

- Performance -> Distributing -> Faults -> Fault Tolerance

Low <- Consistency <- Inconsistency <- Replicas <- performance

- Learn about the CAP theorem!

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Google*

Why build GFS?

Why build GFS?

- Component fails frequently
  - File system = thousands of storage machines
  - Some % not working at any given time
- Files are huge; Multi-GB files are the norm
  - It doesn't make sense to work with billions of *n*KB-sized files
- Most files are appended, not overwritten
  - Not random access overwrite
- Co-designing apps & file system
  - Better flexibility

Desiderata

- Must monitor & automatic recover from component failures
- Modest number of large files
- Workload
  - Large streaming reads + small random reads
  - Many large sequential writes
    - Random access overwrites don't need to be efficient
- Semantics for concurrent appends
- High sustained bandwidth
  - More important than low latency

Basic Design Idea

- “Normal” file systems
  - Store data & metadata close to each other on the same device
  - Example: UFS, FFS
- GFS: store data and metadata on different servers
  - Metadata = information about the file → Master
    - Includes name, access permissions, timestamps, size, location of data blocks
  - Data = actual file contents → Chunk servers
    - Data storage: fixed-size chunks
    - Chunks replicated on several systems

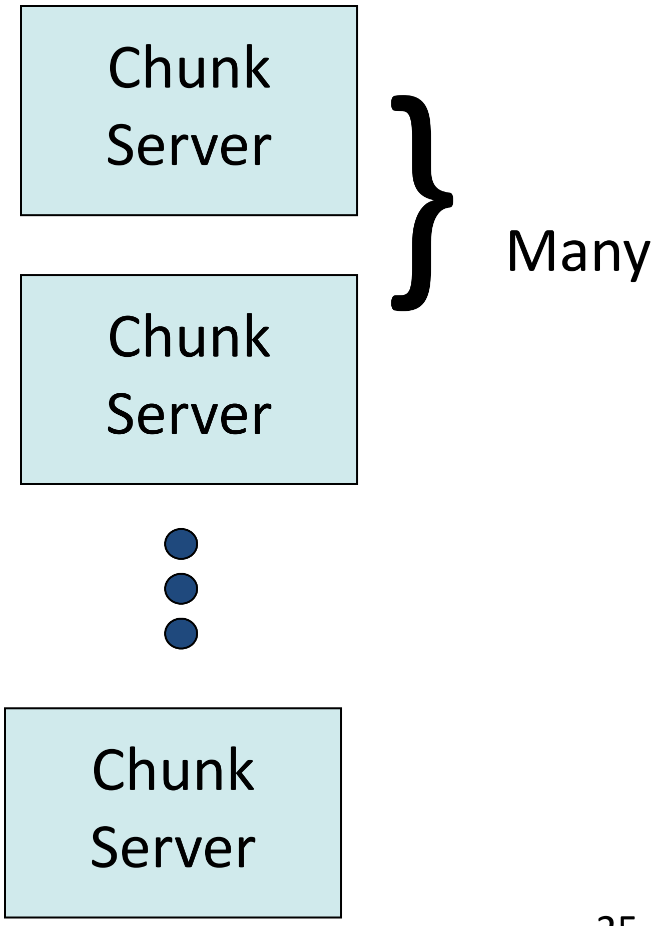
Interface

- GFS does *not* have a standard OS-level API
  - No POSIX API
  - No kernel/VFS implementation
  - User-level API for accessing files
  - GFS servers are implemented in user space on top of native Linux FS

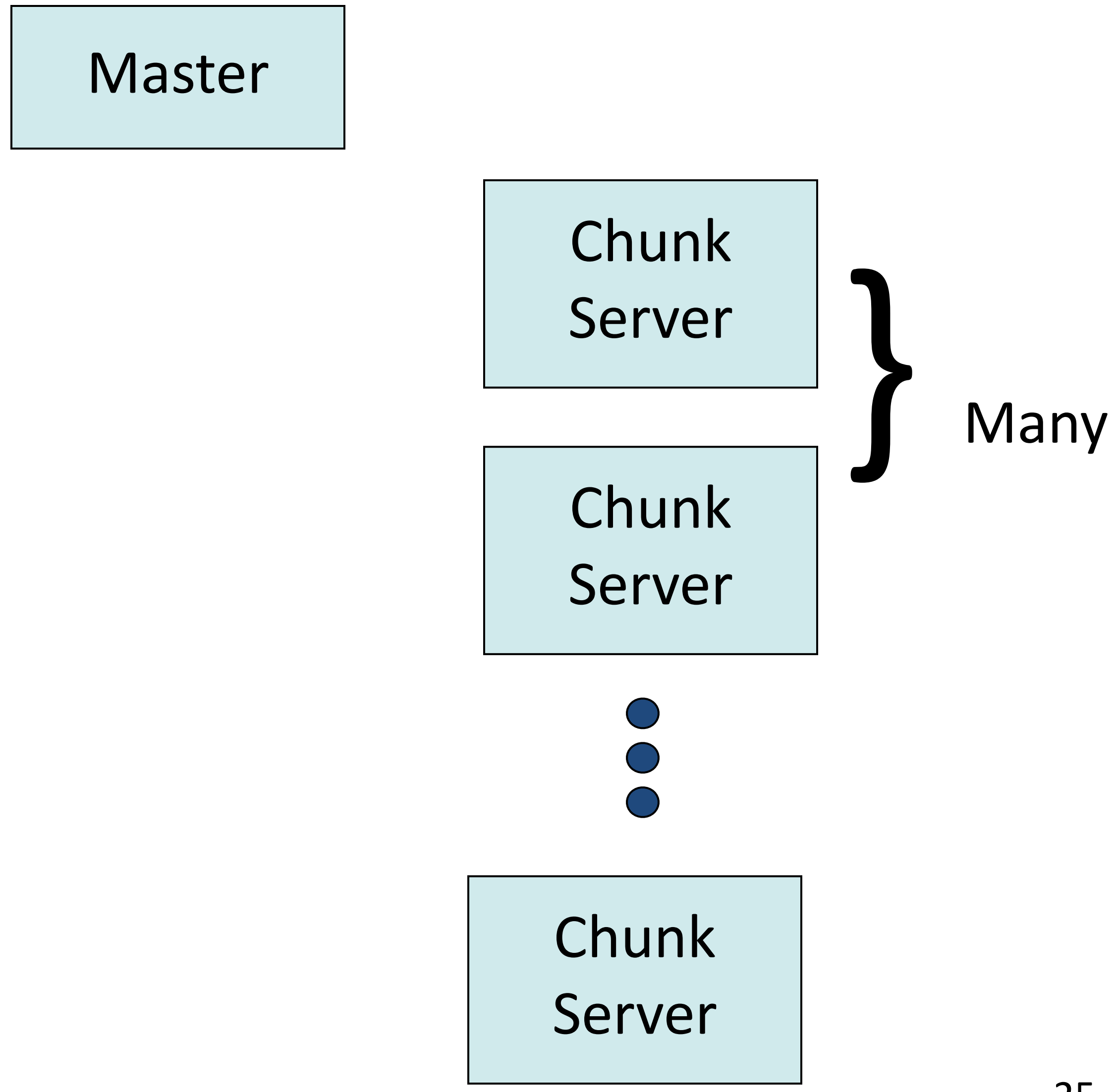
- Basic operations
  - Create, delete, open, close, read, write

- Additional operations
  - Snapshot: create a copy of a file or directory tree at low cost
  - Record append: allow multiple clients to append atomically without locking

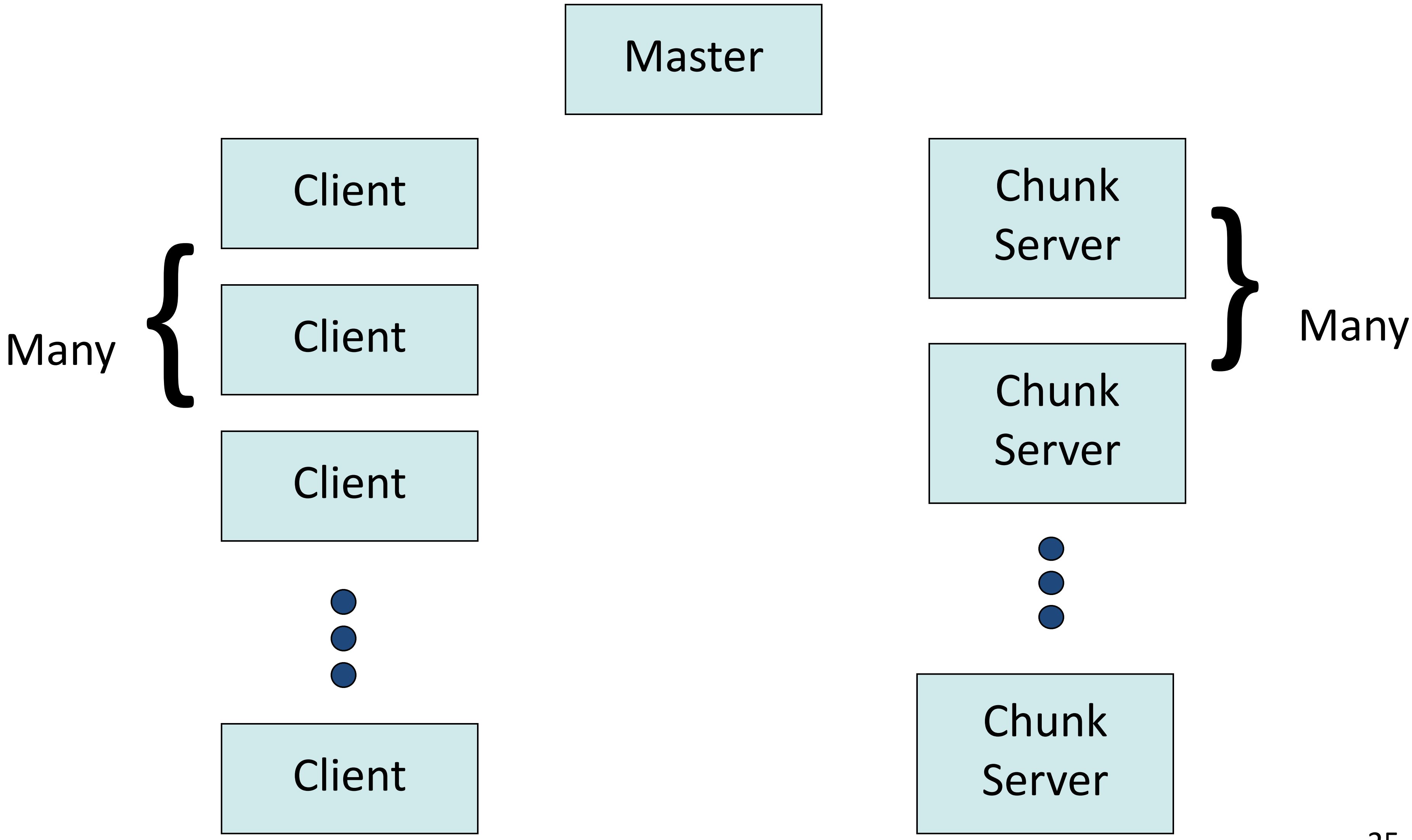
Architecture



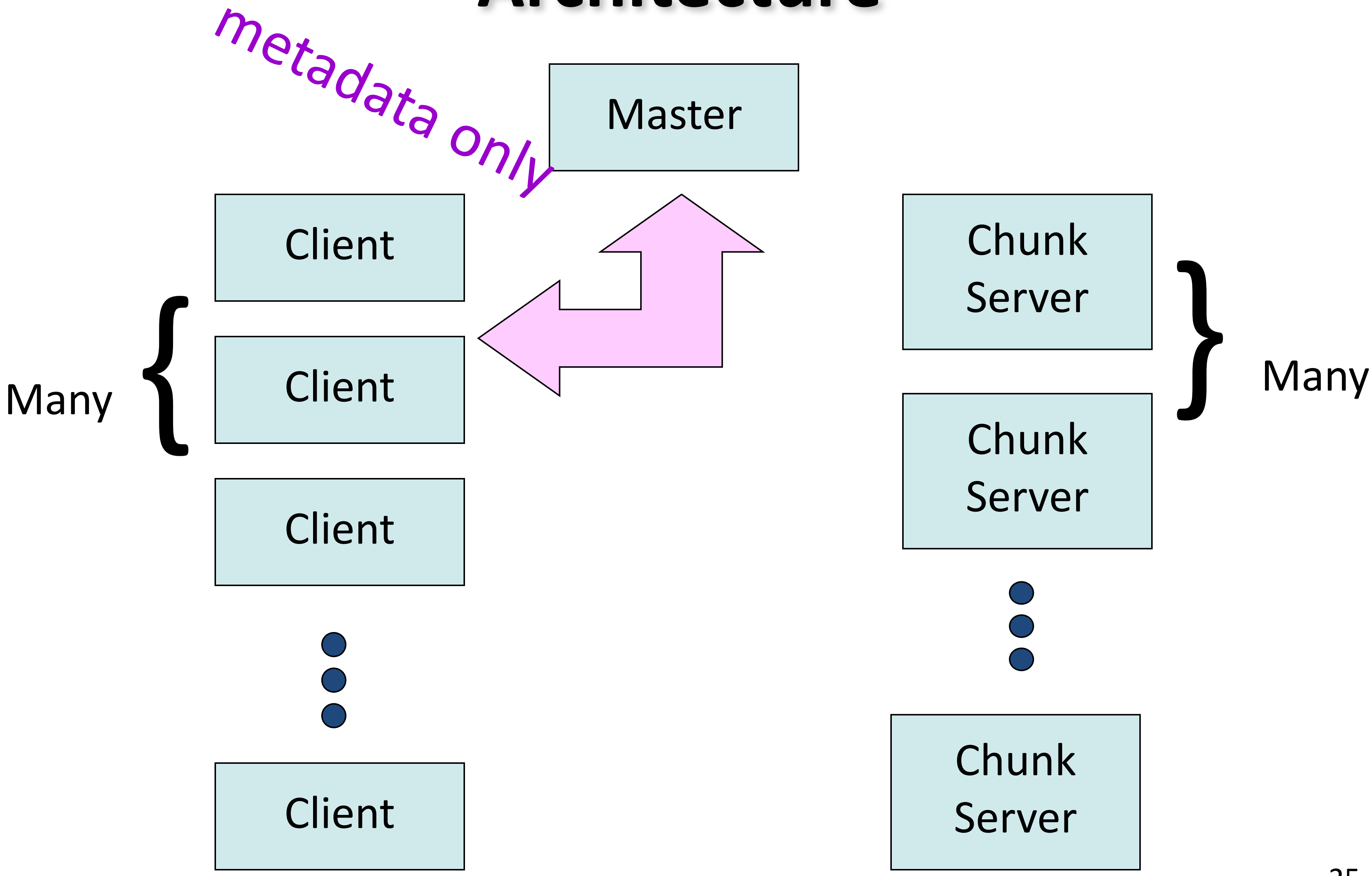
Architecture



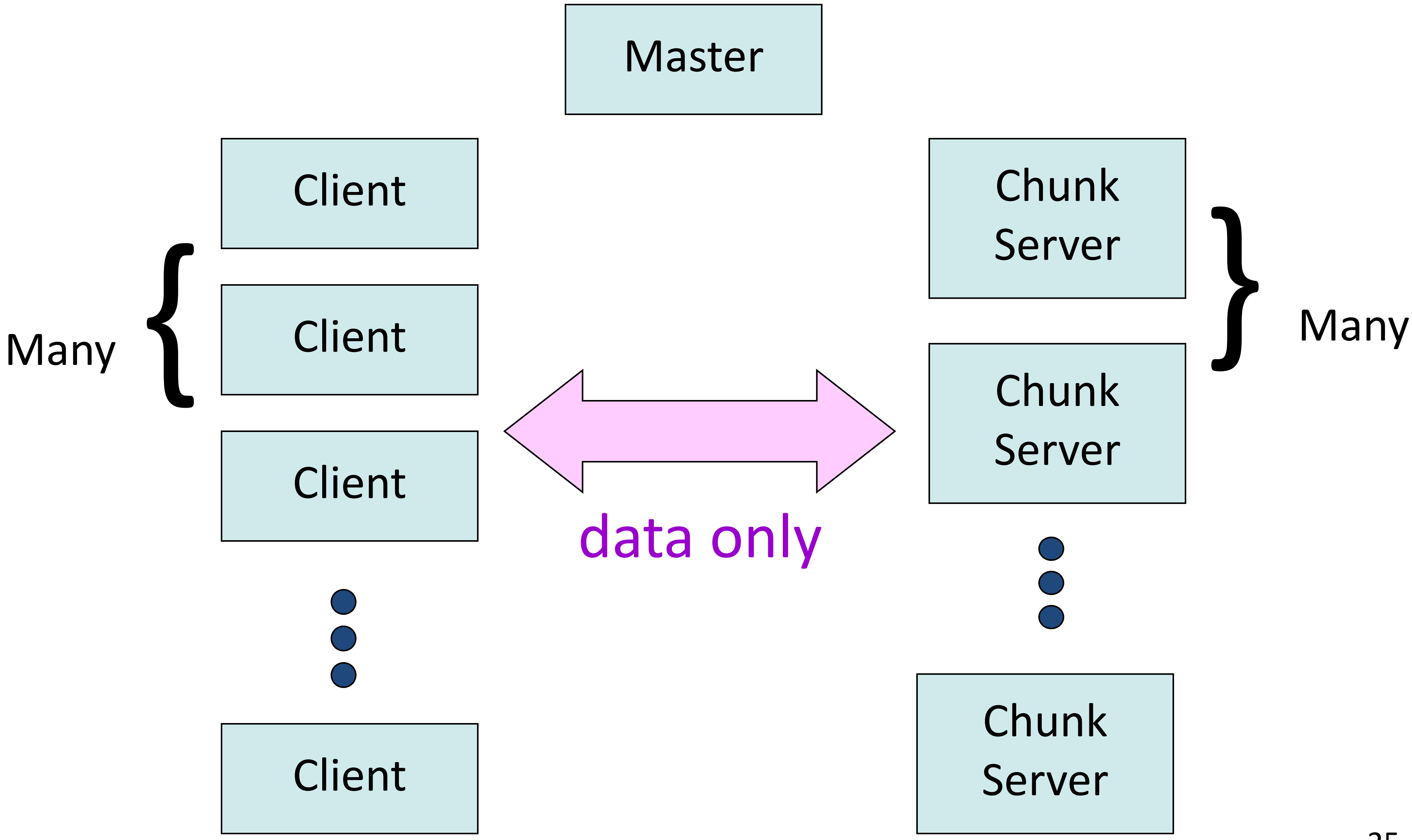
Architecture



Architecture



Architecture



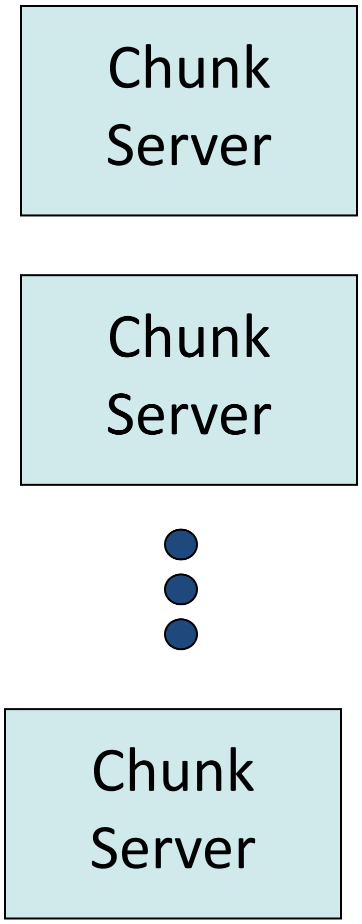
Architecture

Master

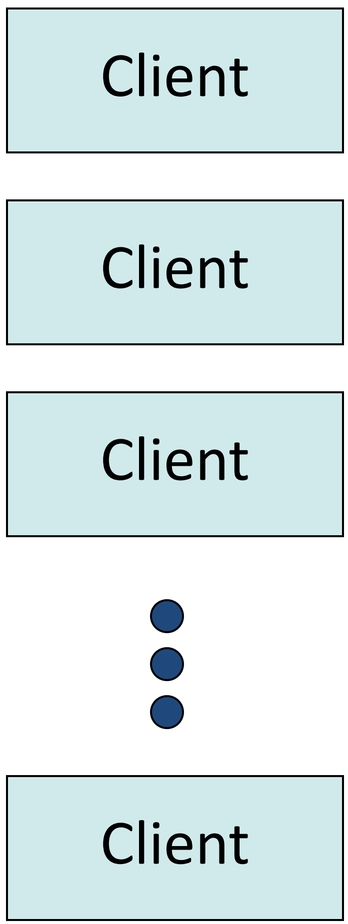
- Master: stores all metadata
  - Namespace
  - Access-control information
  - Chunk locations
  - ‘Lease’ management
- Heartbeats
  - Periodic communication with chunk servers
- Having one master → global knowledge
  - Allows better placement / replication
  - Simplifies design
  - Will it be a bottleneck?

Architecture

- Chunk servers: store all files
- Files are divided into fixed-size ***chunks***
- Each chunk
  - 64 MB
  - 64 bit unique handle
  - Triple redundancy
    - replicated three times across multiple chunk servers

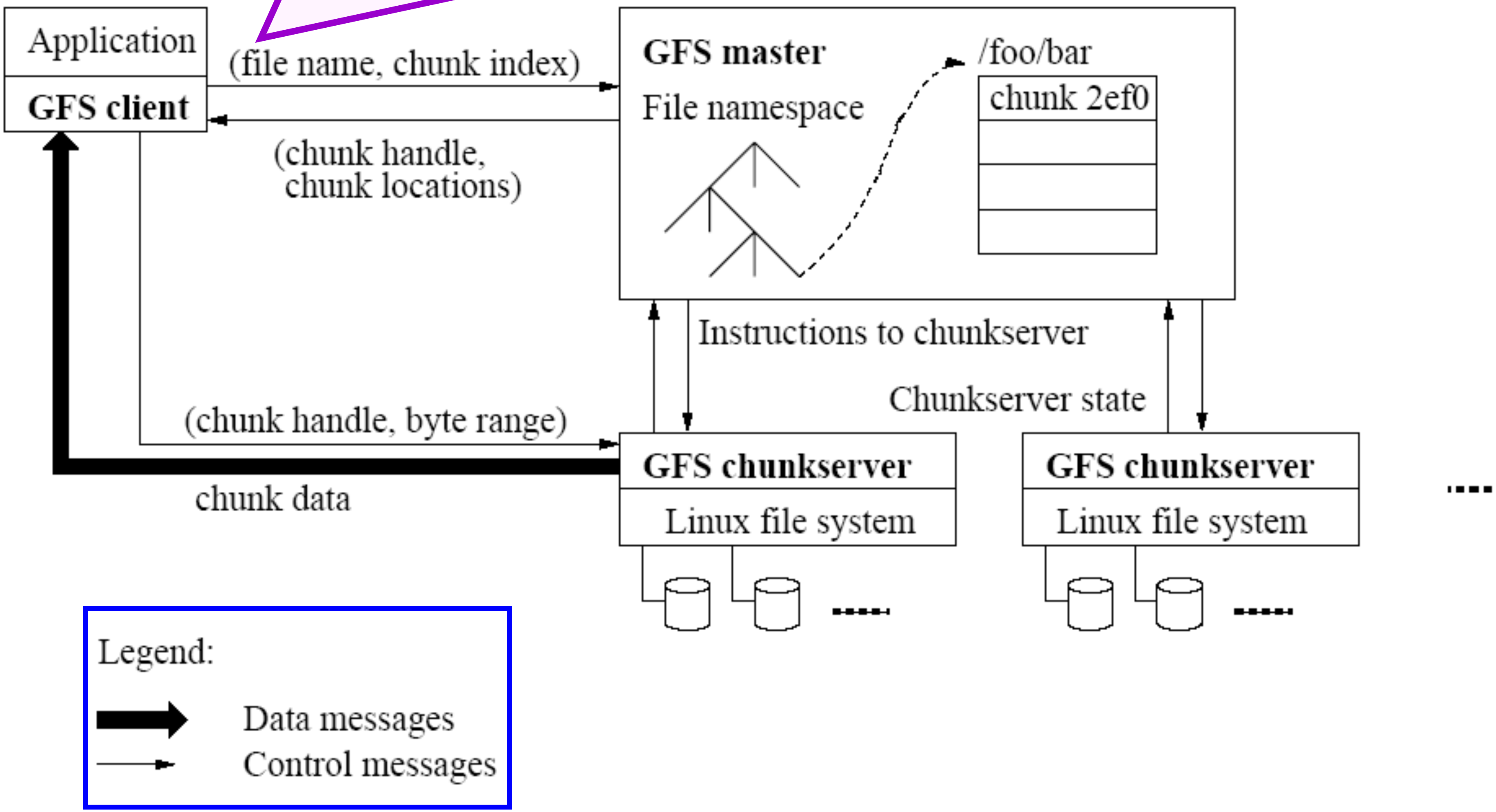


Architecture

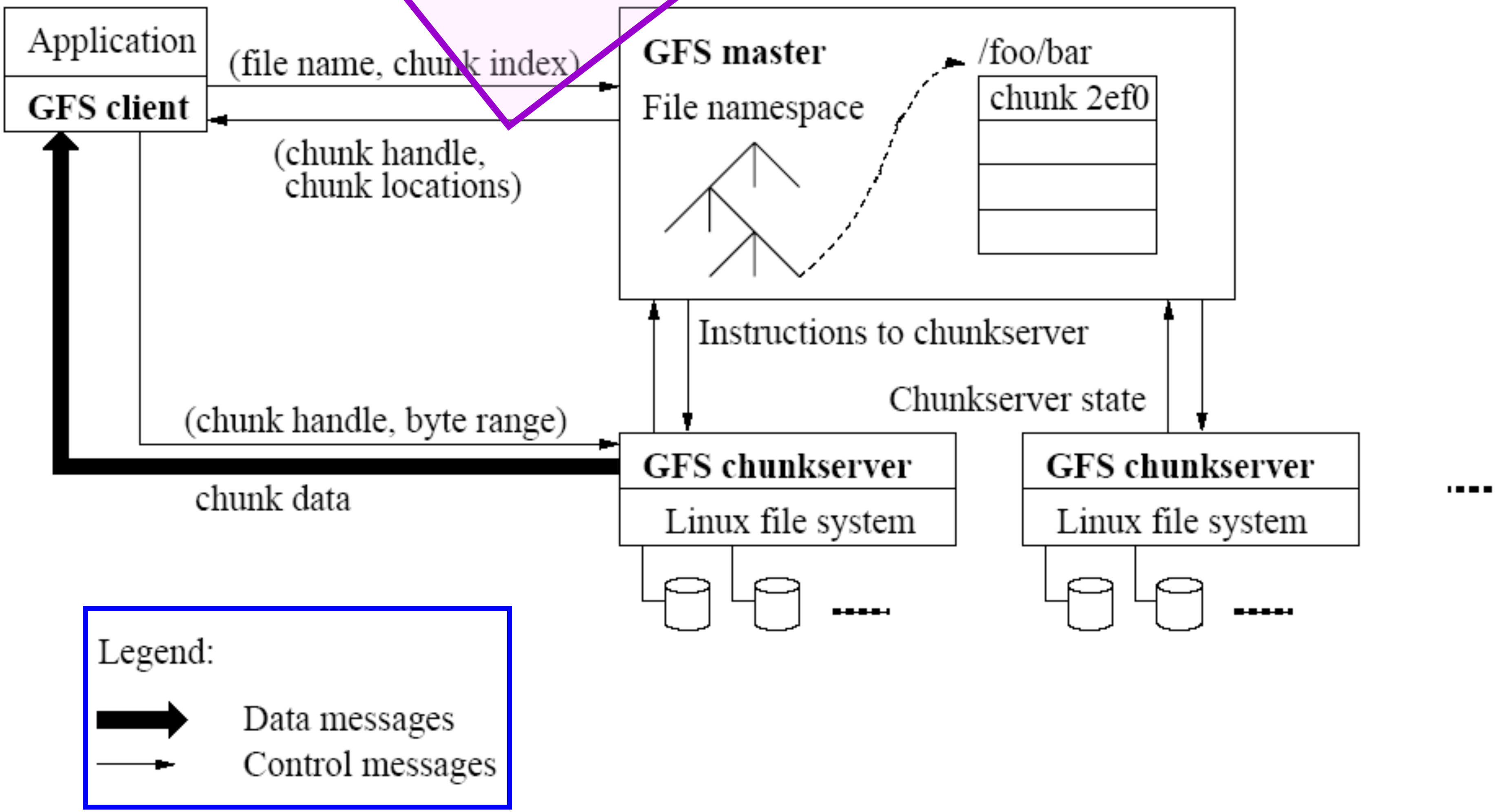


- Each app is linked with GFS client code
  - No OS-level API; user-level library
- Interacts with master for metadata
- Interacts with chunk servers for file data
  - All reads & writes go directly to chunk servers
- Clients cache only metadata
  - Neither clients nor chunk servers cache data

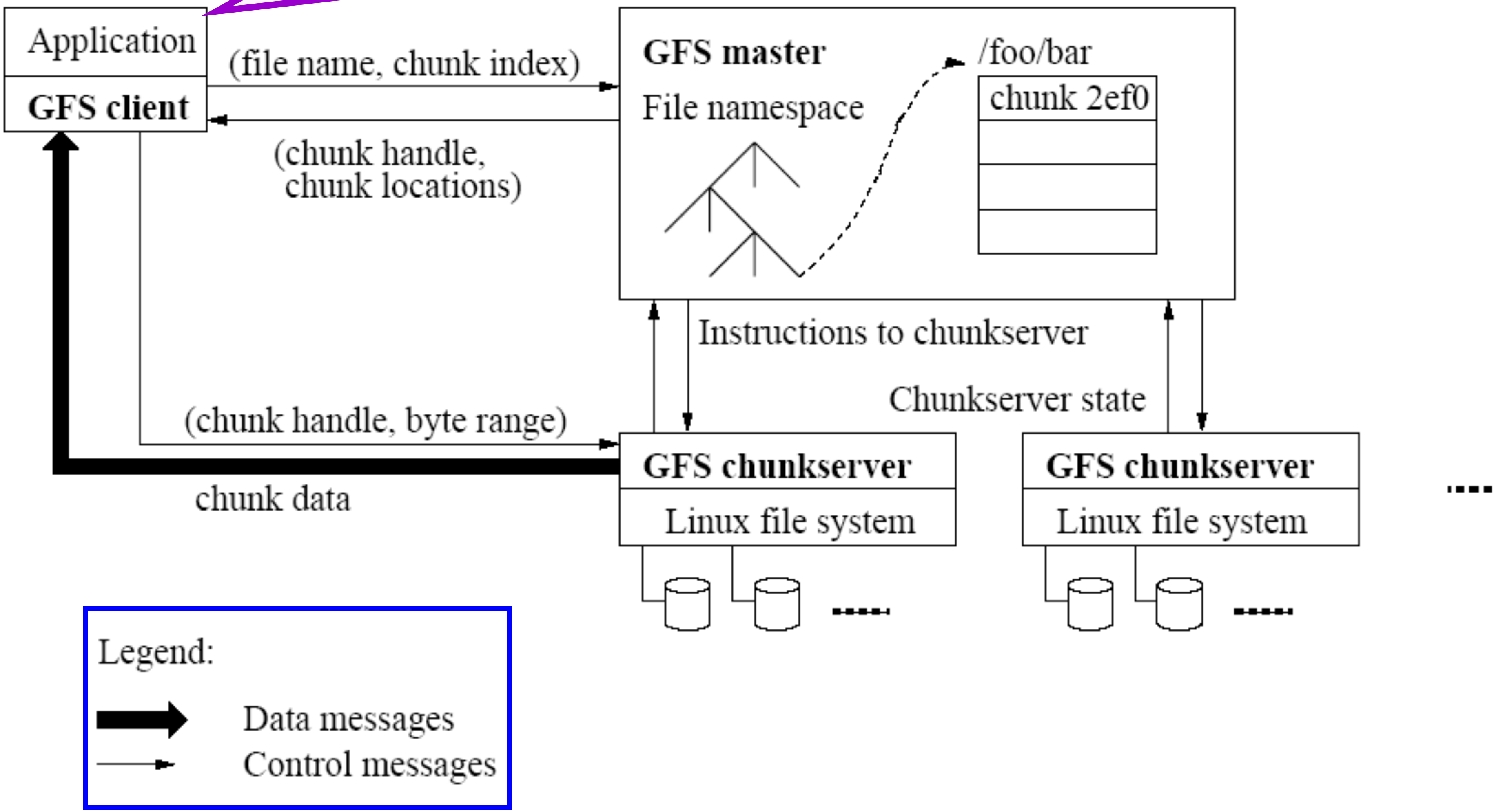
Using fixed chunk size, translate filename & byte offset to chunk index.
Send request to master

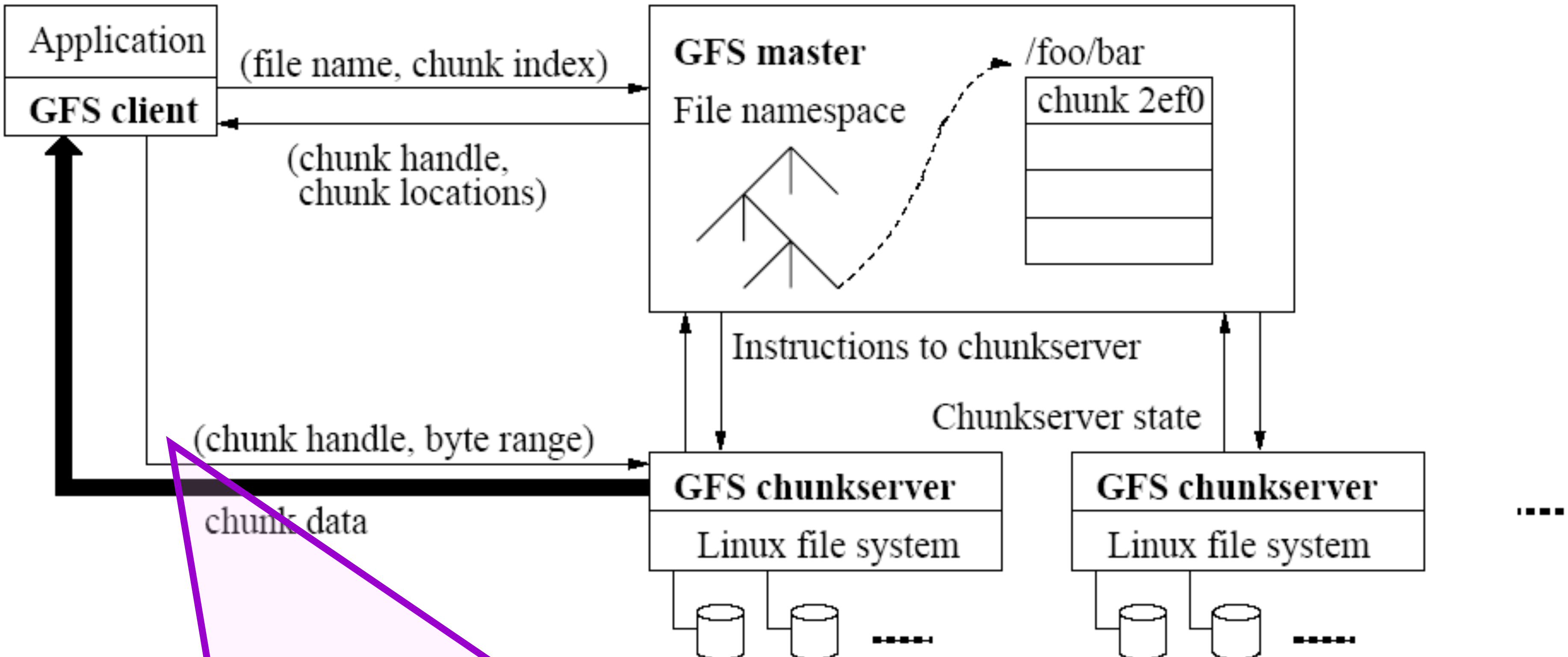


Replies with chunk handle & location of chunkserver replicas (including which is 'primary')



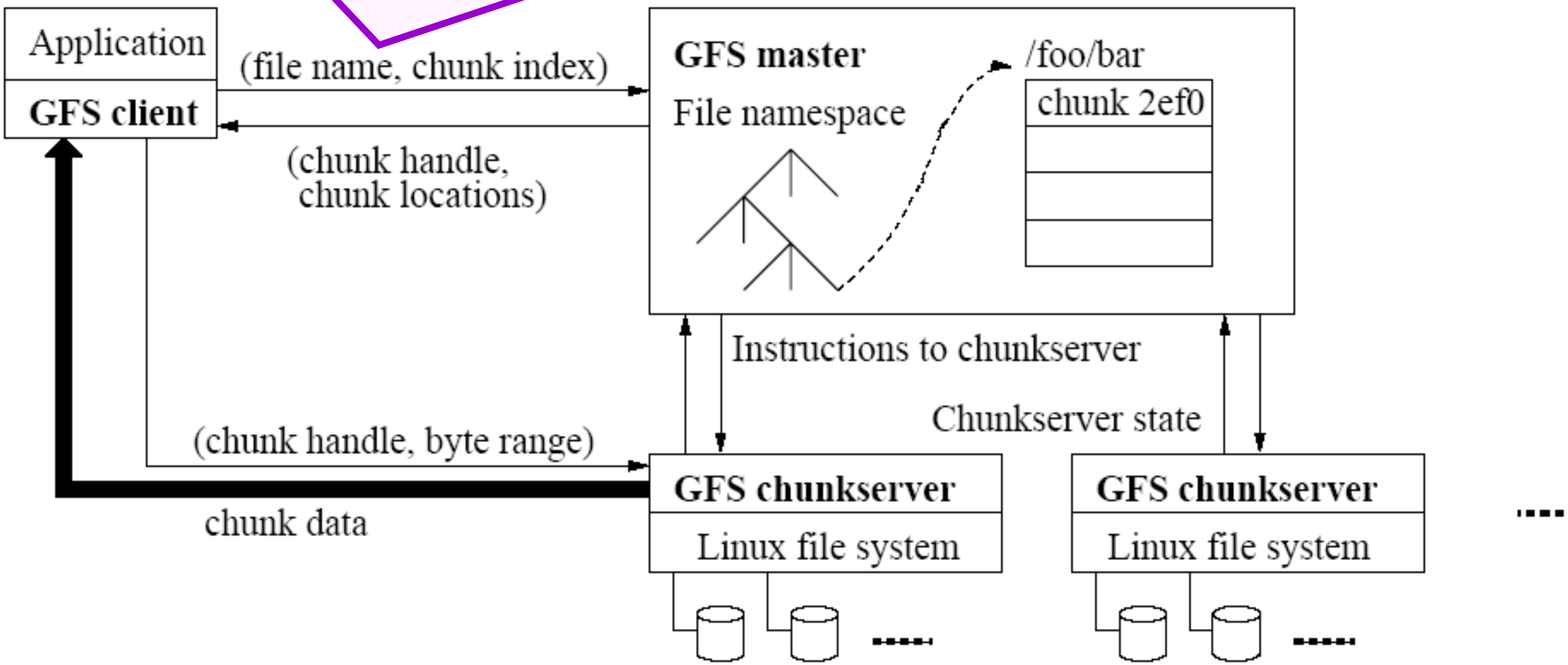
Cache metadata
using file name & chunk index as key





Request data from nearest chunkserver
"chunk handle & byte range"

No need to talk more about this 64MB chunk
Until cached info expires or file reopened



Namespace

- No per-directory data structure like most file systems
  - E.g., directory file contains names of all files in the directory
- No aliases (hard or symbolic links)
- Namespace is a single lookup table
  - Directly maps pathnames to metadata
  - No need for multiple lookups

Relaxed Consistency Model

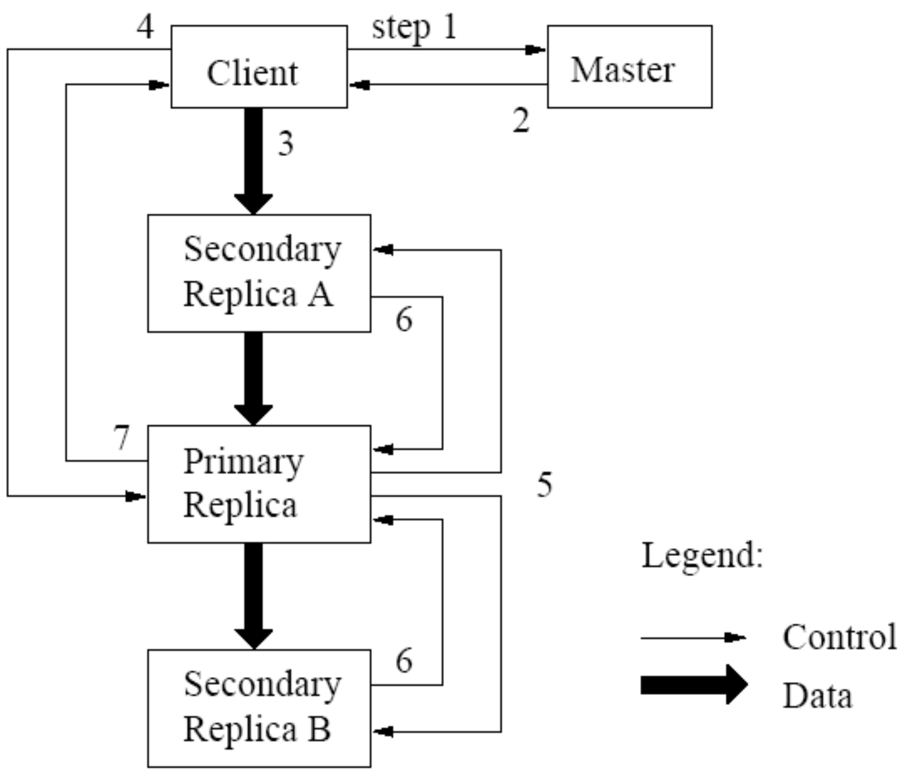
- Namespace mutations (e.g., file creation) are *atomic*
- State of file regions
  - ***Consistent***: all clients see the same data (but may not reflect all mutations)
  - ***Defined***: consistent + clients see the full effect of mutations

	Write	Record Append
Serial success	*defined*	*defined* interspersed with *inconsistent*
Concurrent successes	*consistent* but *undefined*	
Failure	*inconsistent*	

- GFS applies modification to a chunk in the same order on all its replicas
  - After a sequence of modifications, if successful, then modified file region is guaranteed to be ***defined***

Leases & Mutation Order

- Objective
  - Ensure data consistent & defined
  - Minimize load on master
- Master grants 'lease' to one replica
  - Called 'primary' chunkserver
- Primary *serializes* all mutation requests
  - Communicates order to replicas



Atomic Record Appends

- Traditional writes would need a distributed lock manager

- Record append
  - Follows the same control flow of mutation with extra logic
    - Difference: append may spill over

- Primary checks to see if append spills over into new chunk
  - If so, pads ***old*** chunk to full extent
  - Tells secondary chunk-servers to do the same
  - Tells client to try append again on ***next*** chunk
- Usually works because
  - max(append-size) < ¼ chunk-size [API rule]
  - (meanwhile other clients may be appending)

Master Replication

- For fault tolerance
- Master log & checkpoints replicated
- Outside monitor watches master livelihood
  - Starts new master process as needed
- Shadow masters
  - Provide read-access when primary is down
  - Lag state of true master

Conclusions

- De-coupling of data and control flows
- Single-master design
  - Many advantages
  - Single point of failure?
- Focusing on the core use cases of the file system (e.g. atomic appends) can lead to the right abstractions
- Eventual consistency