Distributed Systems

15-440/640

Fall 2018

15 – Cluster File Systems: The Google File System

Readings: “The Google File System” Sections 2.3-2.6, 3.1, 3.3, 5.1, 5.2

Daniel S. Berger

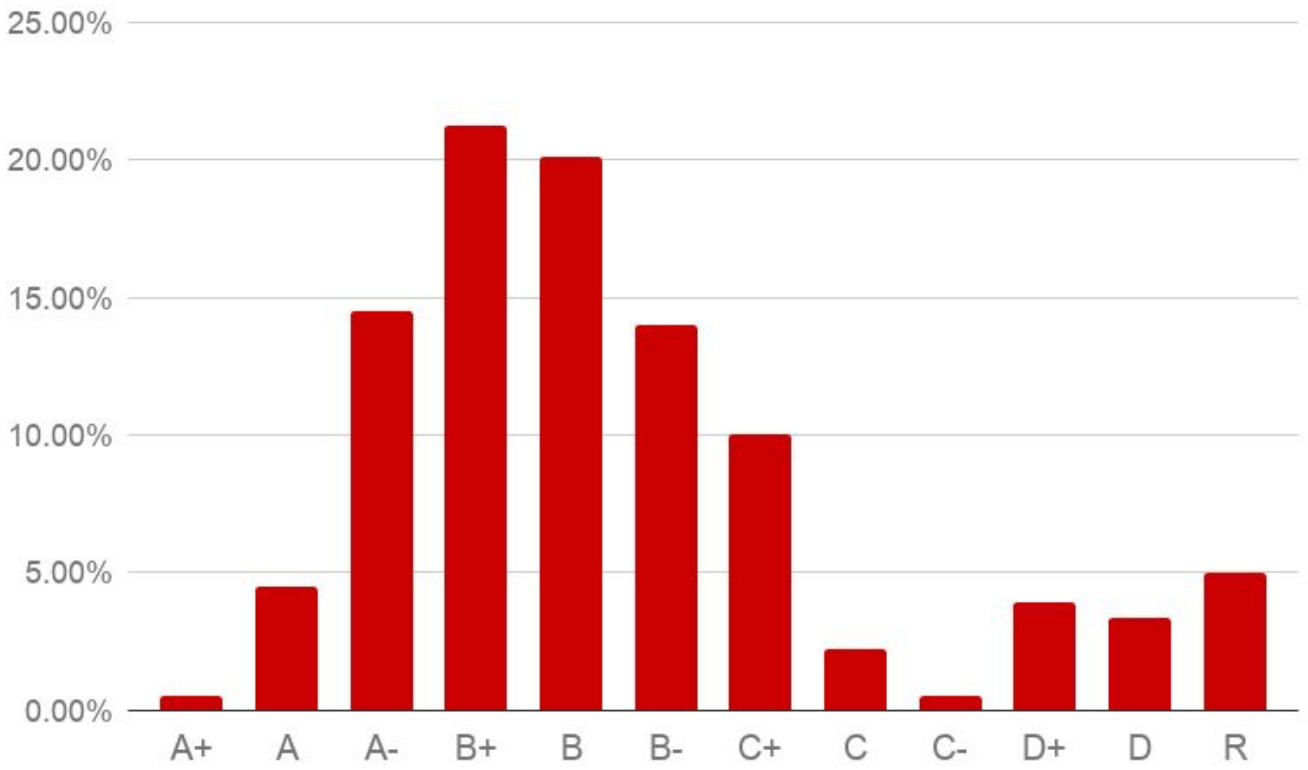
Midterm Results

Solutions on website

	Avg Pts	Topic
Q1	11/15	True/False (DSF, WAL, LSP)
Q2	14/20	Short Answers (Communication, Time Sync, Replication)
Q3	8/9	Communication and RPC
Q4	9/12	DFS (CAP, High availability, caching)
Q5	5/10	Logging and Failure Recovery (Checkpointing)
Q6	13/16	BergerNet (RPCs, 2PC, 3PC, Reliability/Availability)
Q7	9/16	Hepp Dean's Hierarchical Mutual Exclusion (Mutexes, Performance)
Q8	2/2	Feedback

Max (97), Mean (70), Median (71), Std-dev (12)

Midsemester Letter Grades



No curving (regular 90, 80, 70, 60 cutoffs).
Need to pass all parts independently.

Instructor OH & Regrade Requests

Today (Daniel):

- after class to 2.30pm
- in GHC 4124

Thursday (Yuvraj + Daniel)

- after class to 1pm
- in GHC 4124

Thursday (Yuvraj)

- from 1pm to 2pm
- in Wean 5313

Idea: focus on small group / individual meetings. Put your name into list on our door.

Midterm exam pickup (Laura and Jenni, see Piazza)
⇒ Regrade requests in writing by Nov 2

Project 2 and Schedule is Out!

Topic: RAFT distributed consensus

- Released yesterday
- **Recitation tomorrow, Wean 7500**
- Checkpoint on 11/5
- Final deadline on 11/12

HW3 on 10/28

Expect P3 right after P2

Updated schedule of classes, projects, homeworks

Lecture Today: Cluster FS Case Study

Task for today: “design a new distributed filesystem for large clusters”

What would you do? How would you start?

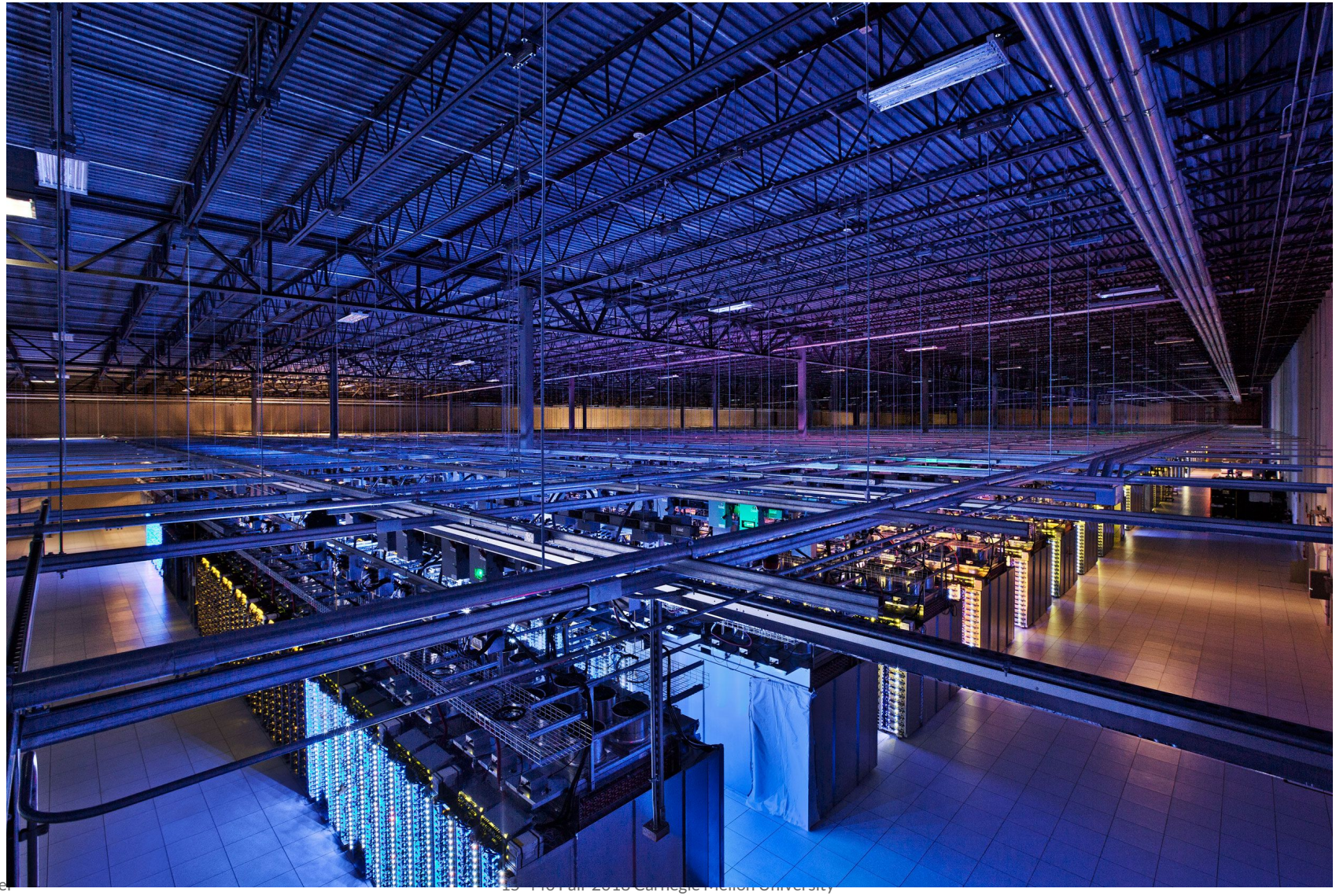
What if you were Google: “your most important workload is a search engine/ Spanner backend”

What would change in your design?

GFS: Google’s distributed fault-tolerant file system

- WAL + Checksums + Primary-Backup + new tricks
- Focus on maintainability & data center environment
- Very different from DFS, DDB seen so far

GFS Operation Environment



GFS Operation Environment

- Hundreds of thousands of commodity servers
- Millions of commodity disks
- Failures are **normal (expected)**:
  - App bugs, OS bugs
  - Human error
  - Disk failures
  - Memory failures
  - Network failures
  - Power supply failures

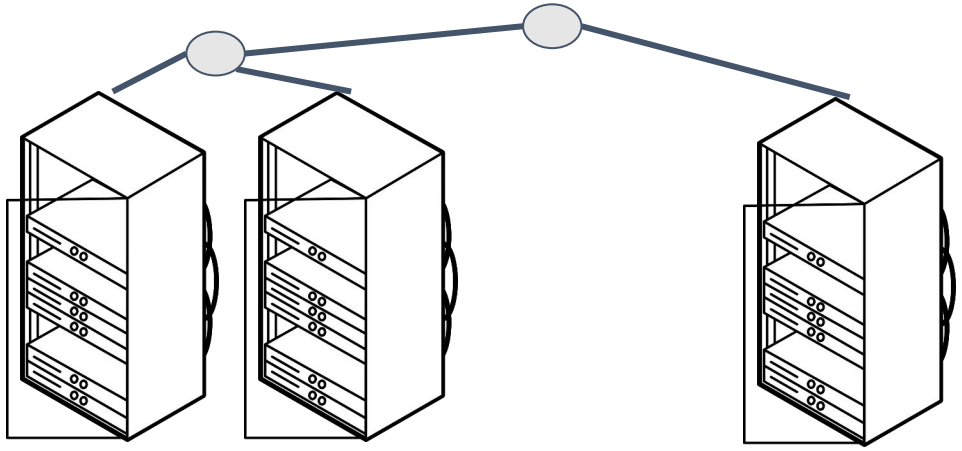
GFS: Workload Assumptions

- Large files, >= 100 MB in size
- Large, streaming reads (>= 1 MB in size)
  - Read once
- Large, sequential writes that append
  - Write once
- Concurrent appends by multiple clients (e.g., producer-consumer queues)
  - Want atomicity for appends without synchronization overhead among clients

GFS Design Goals

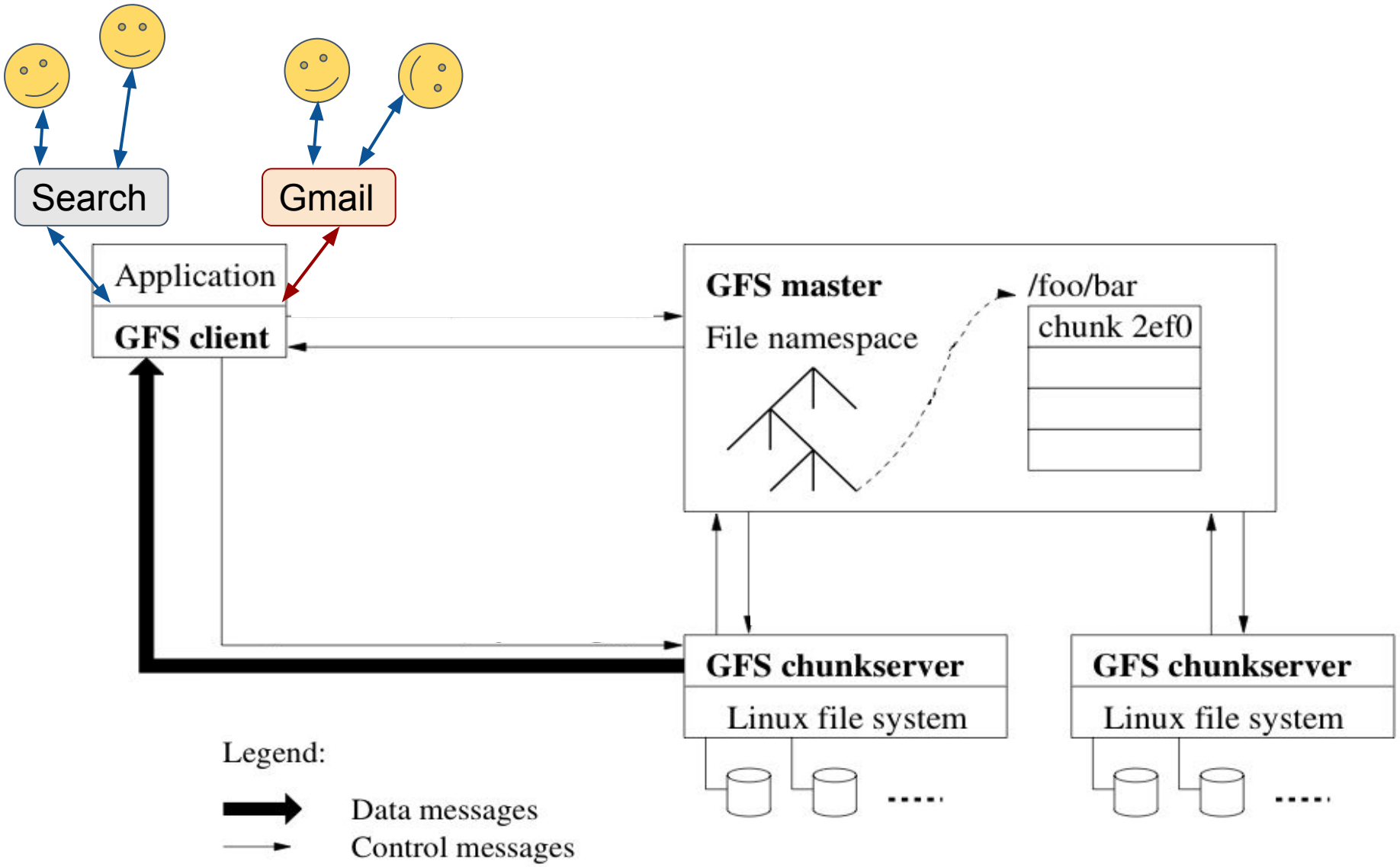
- Maintain high data and system availability
- Handle failures gracefully and transparently
- Low synchronization overhead between entities of GFS
- Exploit parallelism of numerous disks/servers
- Choose high sustained throughput over low latency for individual reads / writes
- Co-design filesystem and applications (GFS client library)

GFS Architecture



- **One master server**
  - State replicated on backups
- Many chunk servers (100s – 1000s)
  - Chunk: 64 MB portion of file, identified by 64-bit, globally unique ID
  - Spread across racks; intra-rack b/w greater than inter-rack
- Many clients accessing different files stored on same cluster

High-Level Picture of GFS Architecture



GFS Architecture Master Server

Holds all metadata in RAM; very fast operations on file system metadata

- Metadata:
  - Namespace (directory hierarchy)
  - Access control information (per-file)
  - Mapping from files to chunks
  - Current locations of chunks (chunkservers)
- Delegates consistency management
- Garbage collects orphaned chunks
- Migrates chunks between chunkservers
  - Why is migration needed?

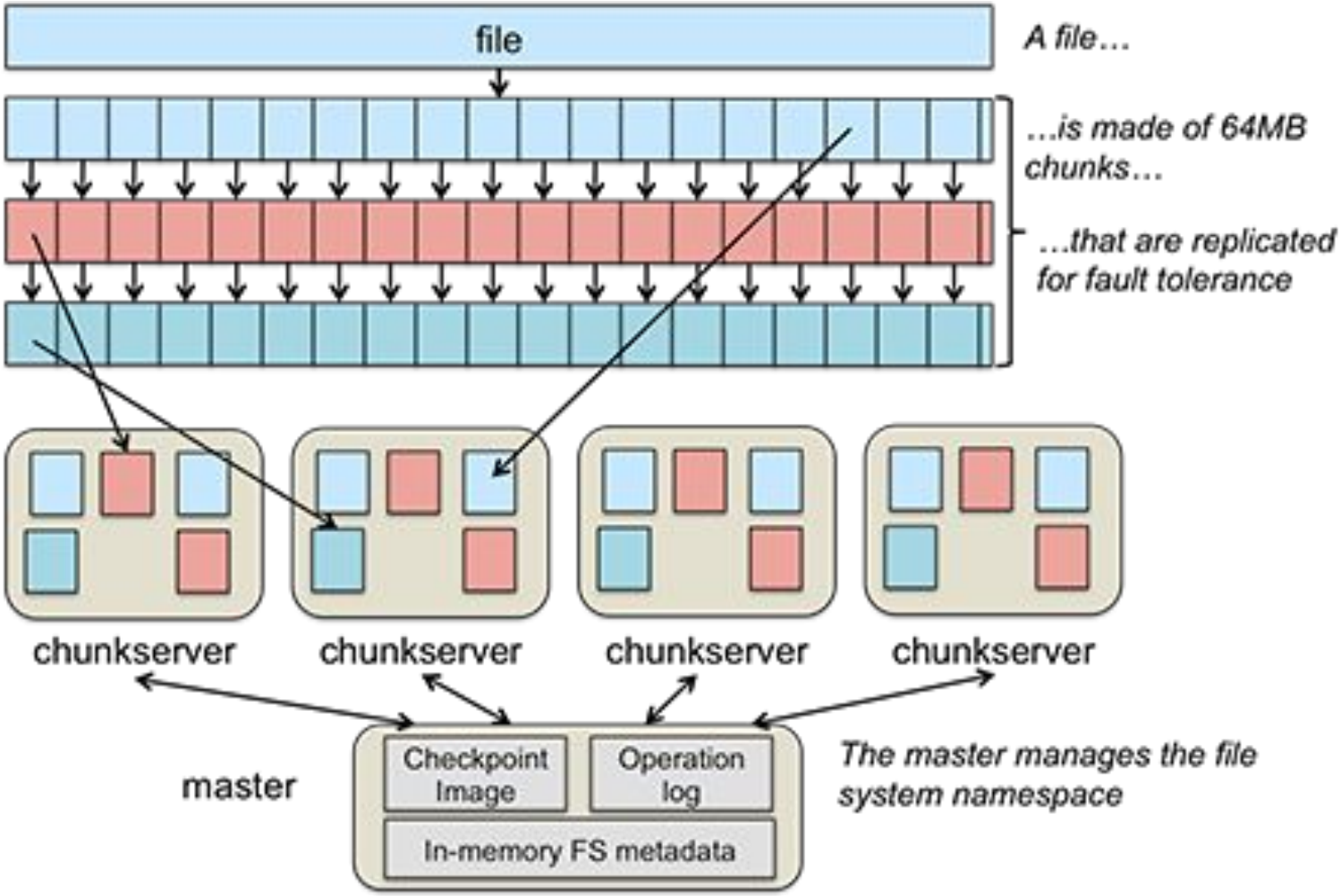
GFS Architecture Chunkserver

- Stores 64 MB file chunks on local disk using standard Linux filesystem (like Ext4), each with version number and checksum

⇒ GFS overhead per chunk

- No understanding of overall distributed file system (just deals with chunks)
- Read/write requests specify chunk handle and byte range
- Chunks replicated on configurable number of chunkservers (default: 3)
- No caching of file data (beyond standard Linux buffer cache)
- Send periodic heartbeats to Master

Master/Chunkservers



GFS Architecture Client

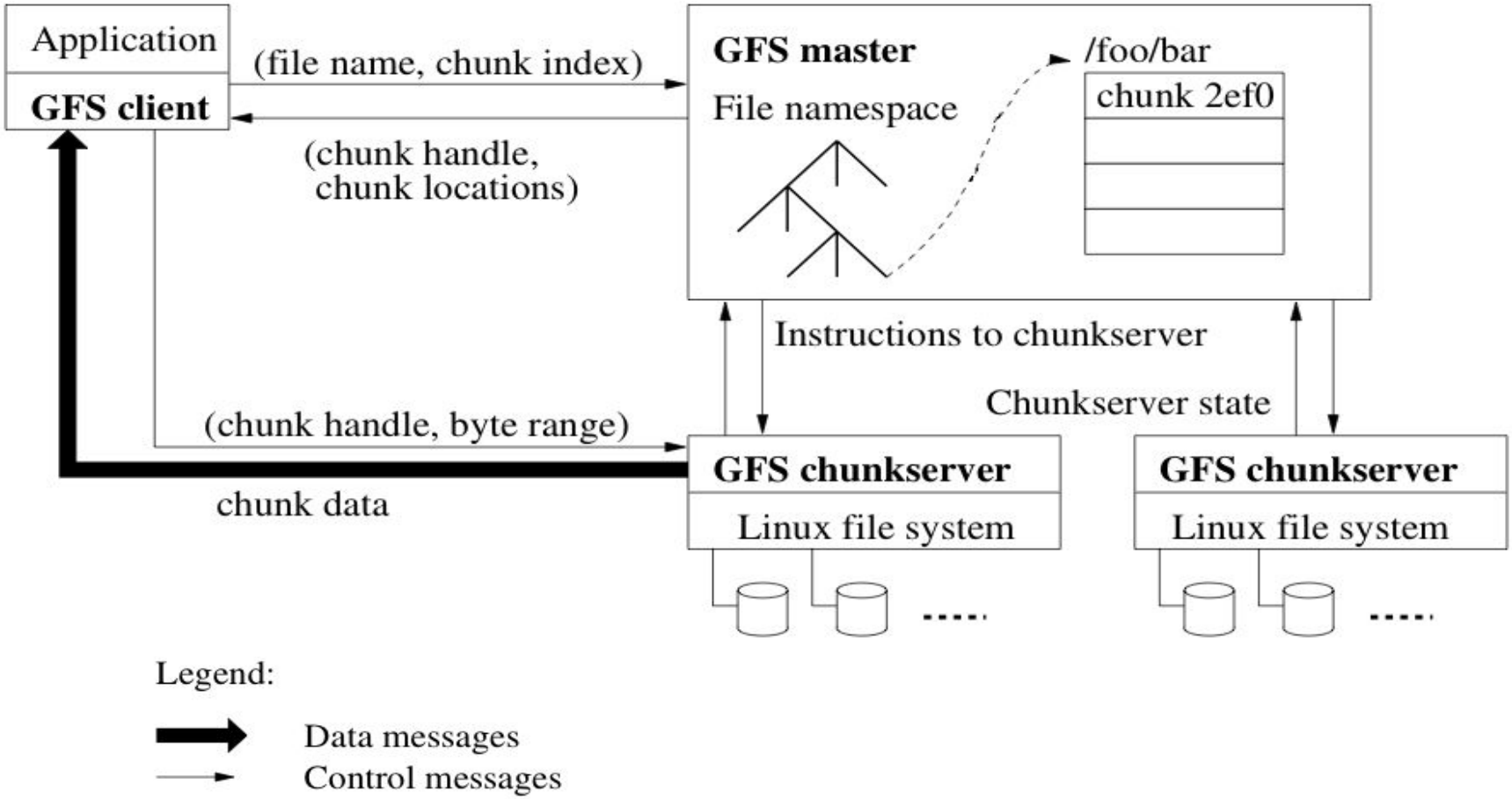
- Issues control (metadata) requests to master server
- Issues data requests directly to chunkservers
  - This exploits parallelism and reduces master bottleneck
- **Caches metadata**
- **No caching of data**
  - No consistency difficulties among clients
  - Streaming reads (read once) and append writes (write once) don't benefit much from caching at client

GFS Architecture Client

- No file system interface at the operating-system level (e.g., VFS layer)
  - User-level API is provided
  - Does not support all the features of POSIX file system access – but looks familiar (i.e. open, close, read…)
- Two special operations are supported.
  - **Snapshot**: efficient way to copy an instance of a file or directory tree
  - **Append**: append data to file as an atomic operation **without having to lock a file**

    ⇒ Multiple processes can append to the same file concurrently without fear of overwriting one another's data

GFS Architecture



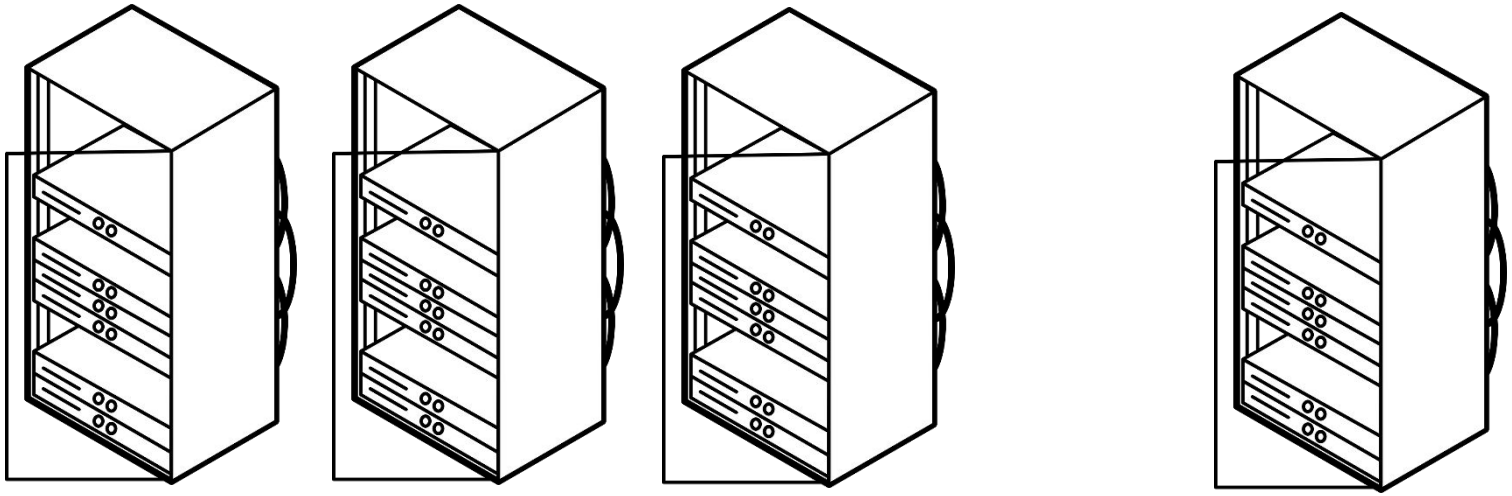
GFS Client Read Operation

- Client sends master:
  - **read(file name, chunk index)**
- Master's reply:
  - chunk ID, chunk version number, locations of replicas
- Client sends to "closest" chunkserver with replica:
  - **read(chunk ID, byte range)**
  - "Closest" determined by IP address on simple rack-based network topology
- Chunkserver replies with data

GFS Client Write Operation I

- 3 replicas for each block → must write to all
- When block created, Master decides placements
  - Two within single rack
  - Third on a different rack
  - Access time / safety tradeoff

How to ensure consistent writes to all replicas?



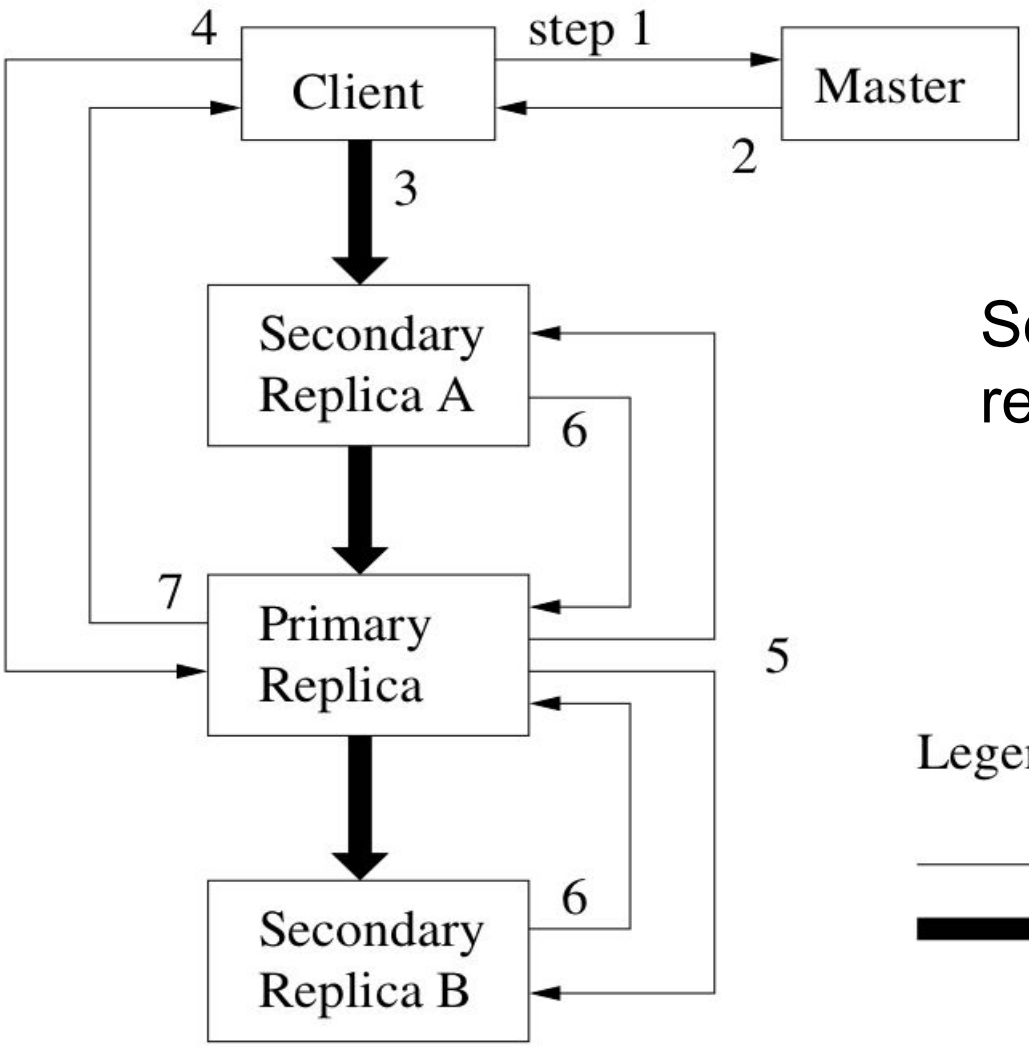
GFS Client Write Operation II

- Some chunkserver is primary for each chunk
  - Master grants lease to primary (typically for 60 sec.)
  - Leases renewed using periodic heartbeat messages between master and chunkservers
- Client asks master for primary and secondary replicas for each chunk

How to efficiently send write data to all three replicas?

- Client sends data to replicas in daisy chain
  - Pipelined: each replica forwards as it receives
  - Takes advantage of full-duplex Ethernet links

GFS Client Write Operation III



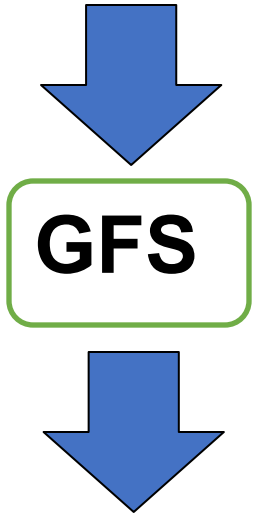
Send to closest replica first

GFS Client Write Operation IV

- All replicas acknowledge data write to client
  - Don't write to file → just get the data
- Client sends write request to primary (commit phase)
- Primary assigns serial number to write request, providing ordering
- Primary forwards write request with same serial number to secondary replicas
- Secondary replicas all reply to primary after completing writes **in the same order**
- Primary replies to client

GFS Record Append Operation

- Google uses large files as queues between multiple producers and consumers
- Variant of GFS write step



Why not use a regular GFS write (client offset)?

- Client pushes data to last chunk's replicas
- Client sends request to primary
- Common case: request fits in last chunk:
  - Primary appends data to own chunk replica
  - Primary tells secondaries to do same at same byte offset in their chunk replicas
  - Primary replies with success to client

GFS Append if Chunk is Full

- When data won't fit in last chunk:
  - Primary fills current chunk with padding
  - Primary instructs other replicas to do same
  - Primary replies to client, "retry on next chunk"
- If record append fails at any replica, client retries operation

What guarantee does GFS provide after reporting success of append to application?

- Replicas of same chunk may contain different data—even duplicates of all or part of record data
- Data written **at least once** in atomic unit

  ⇒ due to GFS client retries until success

GFS File Deletion

- When client deletes file:
  - Master records deletion in its log
  - File renamed to hidden name including deletion timestamp
- Master scans file namespace in background:
  - Removes files with such names if deleted for longer than 3 days (configurable)
  - In-memory metadata erased
- Master scans chunk namespace in background:
  - Removes unreferenced chunks from chunkservers

GFS Logging at Master

What if GFS loses the master?

- Master has all metadata information
  - Lose it, and you've lost the filesystem!
- Master logs all client requests to disk sequentially (→ WAL)
- Replicates log entries to remote backup servers (→ Primary-Backup Replication)
- Only replies to client after log entries safe on disk on self and backups!
- Logs cannot be too long – why?
- Periodic checkpoints as on-disk Btree

GFS Chunk Leases and Version Numbers

- If no outstanding lease (→ chunk primary), when client requests write, master grants new one
- Chunks have version numbers
  - Stored on disk at master and chunkservers
  - Each time master grants new lease, increments version, informs all replicas

Why does GFS need leases and version numbers?

- Network partitioned chunkservers, primaries
- Master can revoke leases
  - e.g., when client requests rename or snapshot a file
- Detect outdated chunkserver with version #

GFS Consistency Model (Metadata)

- Changes to namespace (i.e., metadata) are atomic
  - Done by single master server!
  - Master uses WAL to define global total order of namespace-changing operations

GFS Consistency Model (Data)

- Changes to data are ordered as chosen by a primary
  - But multiple writes from the same client may be interleaved or overwritten by concurrent operations from other clients
- Record append completes at least once, at offset of GFS's choosing
  - Applications must cope with possible duplicates
- Failures can cause inconsistency
  - E.g., different data across chunk servers (failed append)
  - Behavior is worse for writes than appends

GFS Fault Tolerance (Master)

- Replays log from disk
  - Recovers namespace (directory) information
  - Recovers file-to-chunk-ID mapping (but not location of chunks)
- Asks chunkservers which chunks they hold
  - Recovers chunk-ID-to-chunkserver mapping
- If chunk server has older chunk, it's stale
  - Chunk server down at lease renewal
- If chunk server has newer chunk, adopt its version number
  - Master may have failed while granting lease

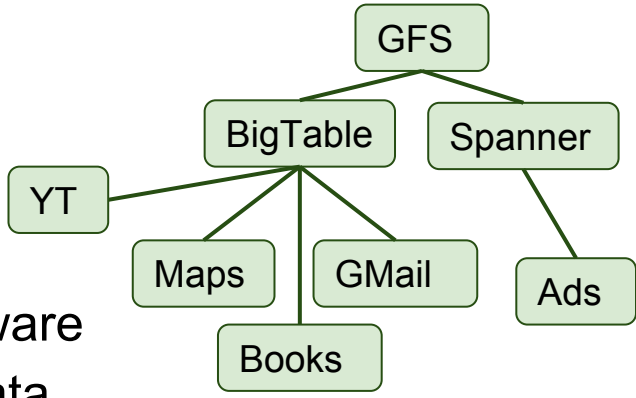
GFS Fault Tolerance (Chunkserver)

- Master notices missing heartbeats
- Master decrements count of replicas for all chunks on dead chunkserver
- Master re-replicates chunks missing replicas in background
  - Highest priority for chunks missing greatest number of replicas

GFS Limitations

- Does not mask all forms of data corruption
  - Requires application-level checksum
- Master biggest impediment to scaling
  - Performance and availability bottleneck
  - Takes long time to rebuild metadata
  - Solution:
    - Multiple master nodes, all sharing set of chunk servers. Not a uniform name space.
- Large chunk size
  - Can’t afford to make smaller
- Security?
  - Trusted environment, but users can interfere

GFS Summary



- Success: used actively by Google
  - Availability and recoverability on cheap hardware
  - High throughput by decoupling control and data
  - Supports massive data sets and concurrent appends
- Semantics not transparent to apps
  - Must verify file contents to avoid inconsistent regions, repeated appends (at-least-once semantics)
- Performance not good for all apps
  - Assumes read-once, write-once workload (no client caching!)

- Successor: Colossus
  - Eliminates master node as single point of failure
  - Storage efficiency: Reed-Solomon (1.5x) instead of Replicas (3x)
  - Reduces block size to be between 1~8 MB
  - Few details public ☹