15-440/640 Distributed Systems

Cluster Filesystems

The Google File System

Announcements

- P2 released. Dates as on course website. **Start early!**
- For everyone’s safety:
  - Please do not congregate after the class for Q/A -- ask questions during the lecture or make use of Piazza and OH
  - If you are sick, please watch the lectures remotely
  - Wear your mask properly **covering your nose and mouth entirely at all times during the lecture**
- For any private communication, use course staff email < ds-staff-f21-private@lists.andrew.cmu.edu>. Not individual instructor email addresses.

Introduction

Deep dive into a distributed filesystem for large clusters (developed by Google).

Google File System (GFS)

Unique design choices

- Markedly different from traditional file systems
- Tradeoffs driven by specific characteristics of the operation environment and workload
- Influenced several cluster file systems design

Open source version:
Apache Hadoop Distributed File System (HDFS)

- Widely successful and deployed in hundreds of companies

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, “The Google File System”, SOSP 2013.

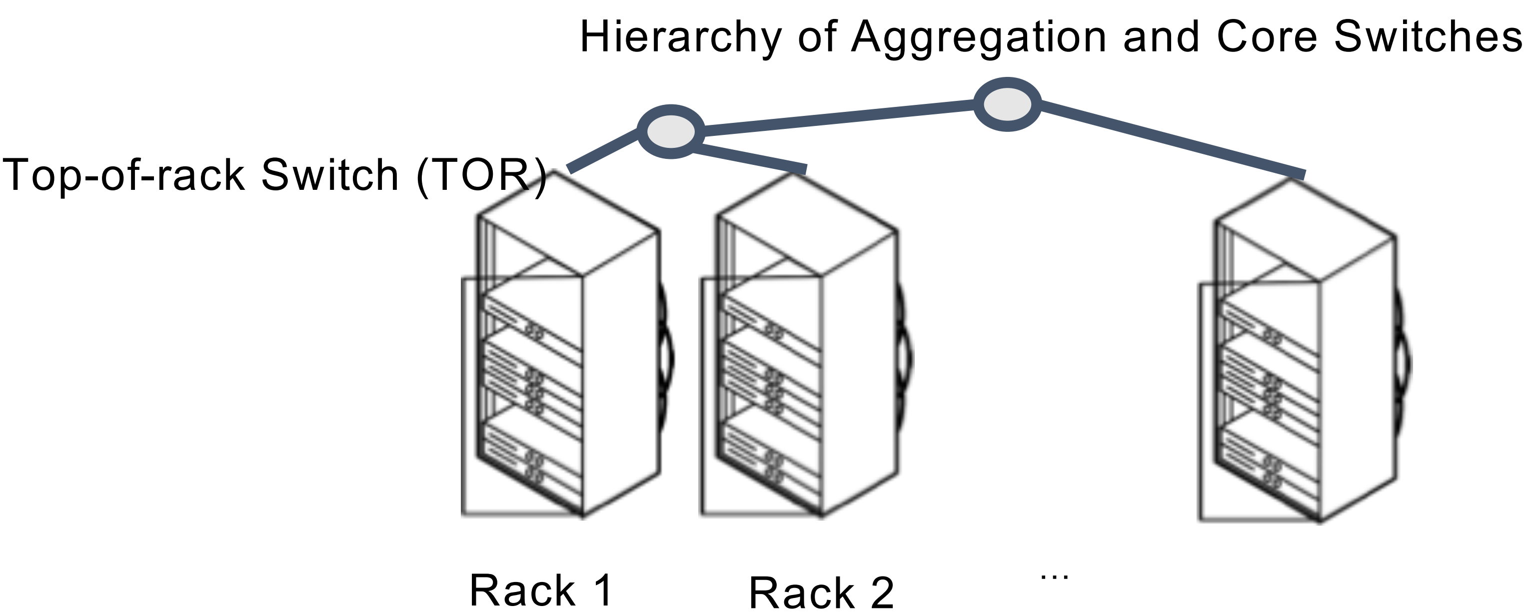
Outline: GFS

- Motivation and design goals
- Architecture
- Client Operations
- Fault tolerance
- Consistency model
- Post-GFS

GFS Operation Environment: Data center

Warehouse scale computer built out of
large number of interconnected commodity servers

GFS Operation Environment: Data center



- Communicating within a rack
  - low latency, high bandwidth, less contention for bandwidth
- Communicating across racks
  - higher latency, limited available bandwidth, more contention

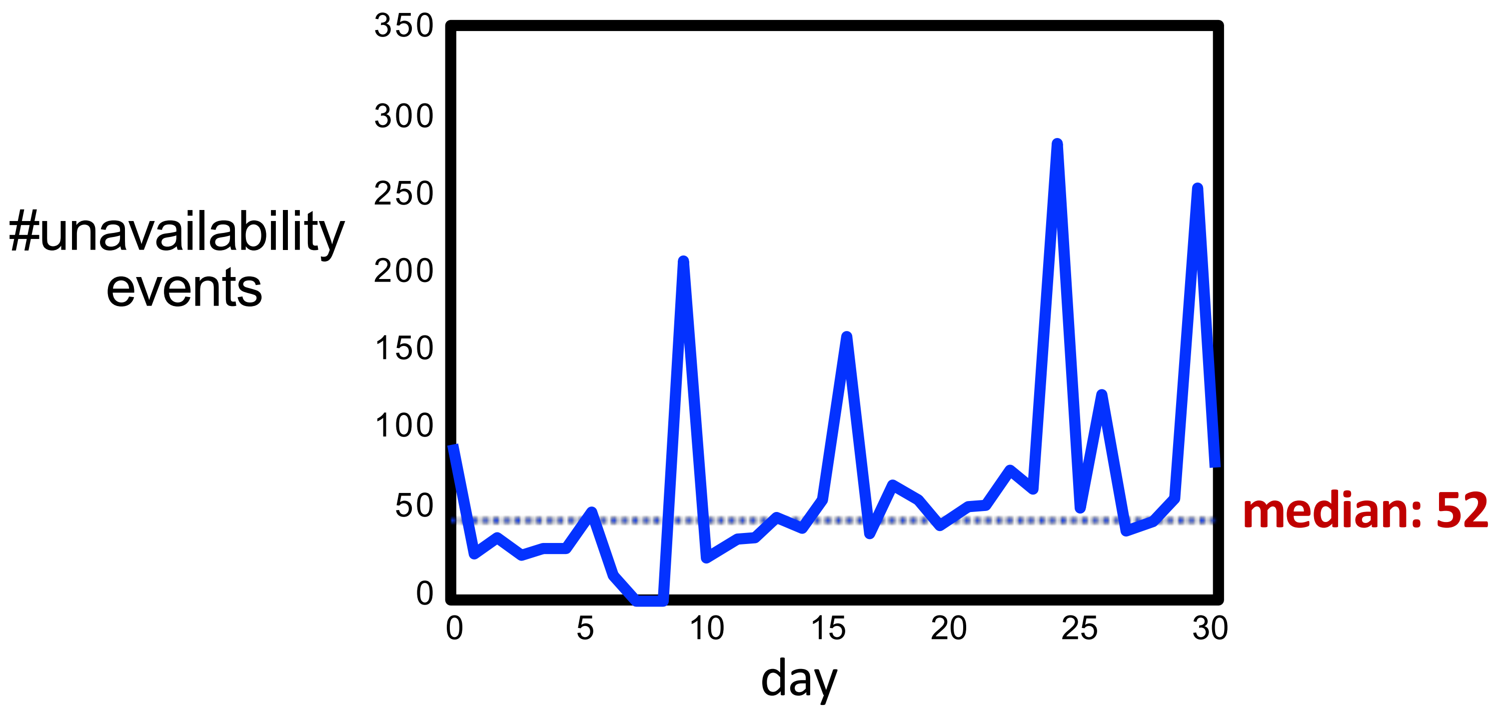
GFS Operation Environment

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

"Failures/unavailabilities are the norm rather than the exception"

Unavailability statistics (from a Facebook cluster)

- Multiple thousands of servers
- Unavailability event: server unresponsive for > 15 min



Daily server unavailability events = 0.5 - 1%

Source: Rashmi et. al., "A Solution to the Network Challenges of Data Recovery in Erasure-coded Distributed Storage Systems: A Study on the Facebook Warehouse Cluster", USENIX HotStorage 2013, ACM SIGCOMM 2014

GFS: Workload Assumptions

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

GFS Design Goals

- Maintain high data and system availability
- Handle failures transparently (i.e., automatically)

- Low synchronization overhead between entities of GFS
- Exploit parallelism of numerous disks/servers
- Choose high sustained throughput for individual reads / writes
  - High throughput more important than low latency

- Co-design filesystem and applications

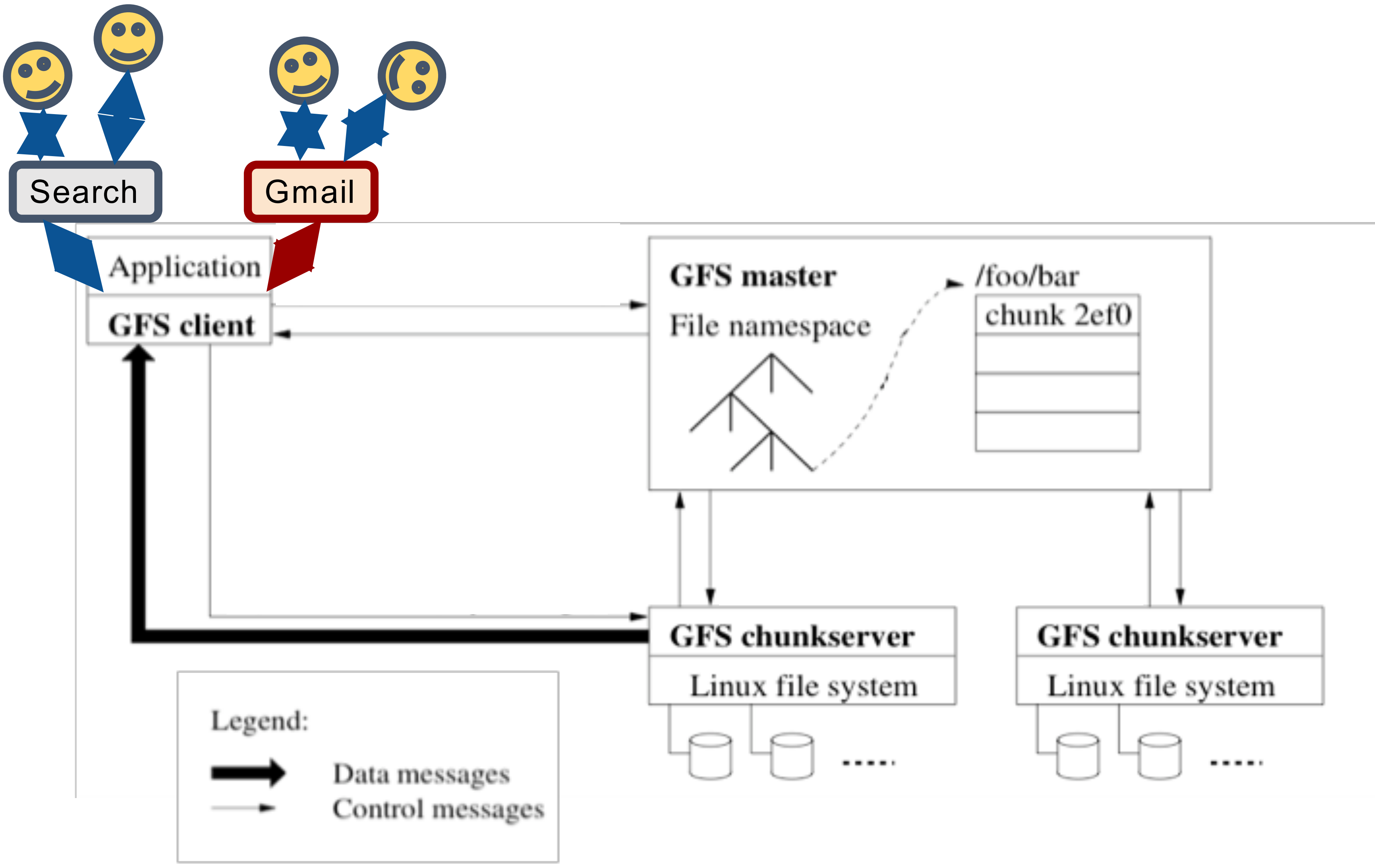
Outline: GFS

- Motivation and design goals
- Architecture
- Client Operations
- Fault tolerance
- Consistency model
- Post-GFS

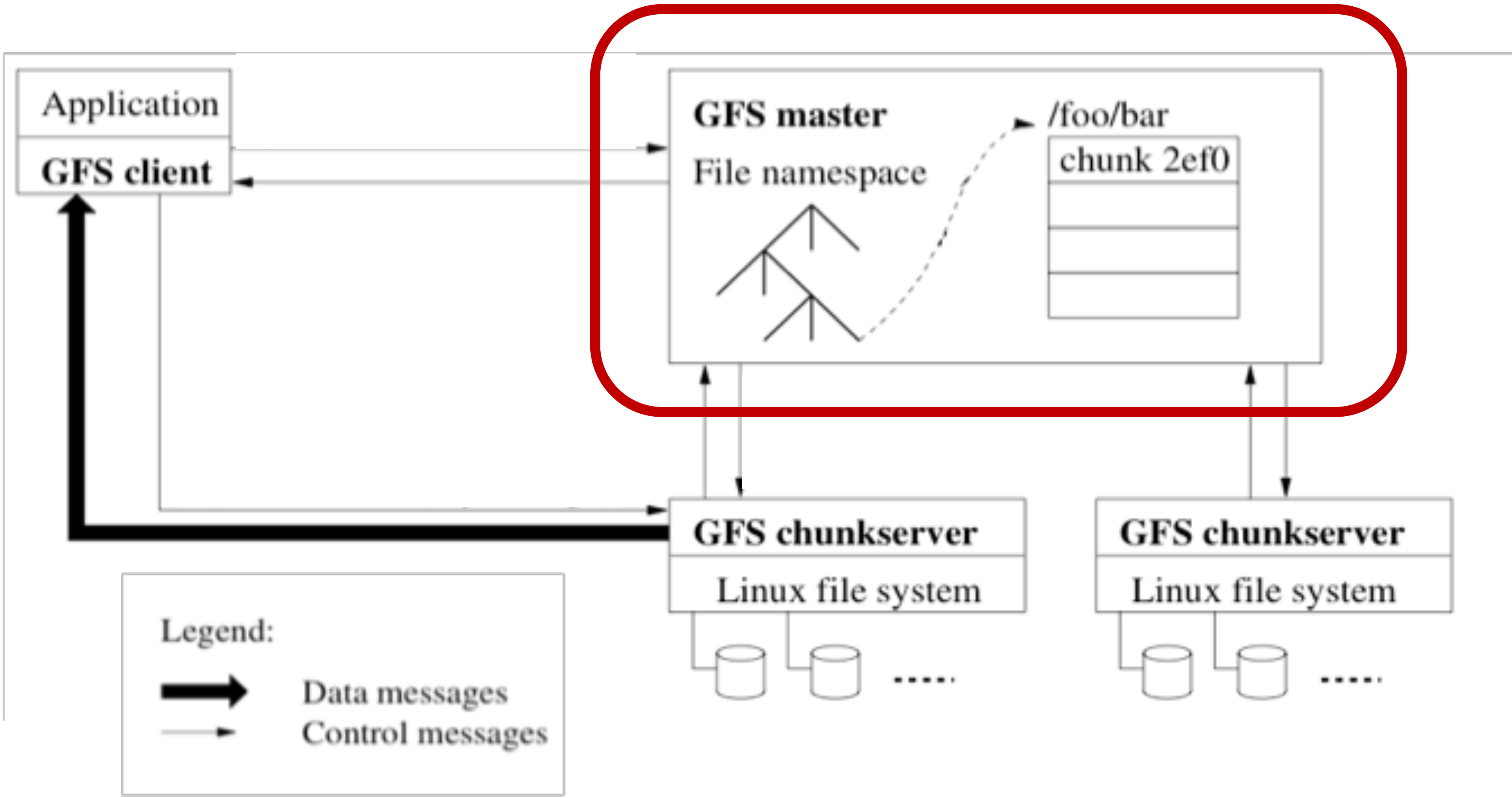
GFS Architecture

- One master server
- Many chunk servers (1000s)
  - Chunk: fixed size (e.g., 64 MB) portion of file, identified by 64-bit globally unique ID
- Many clients accessing different files stored on same cluster

High-Level Picture of GFS Architecture



High-Level Picture of GFS Architecture

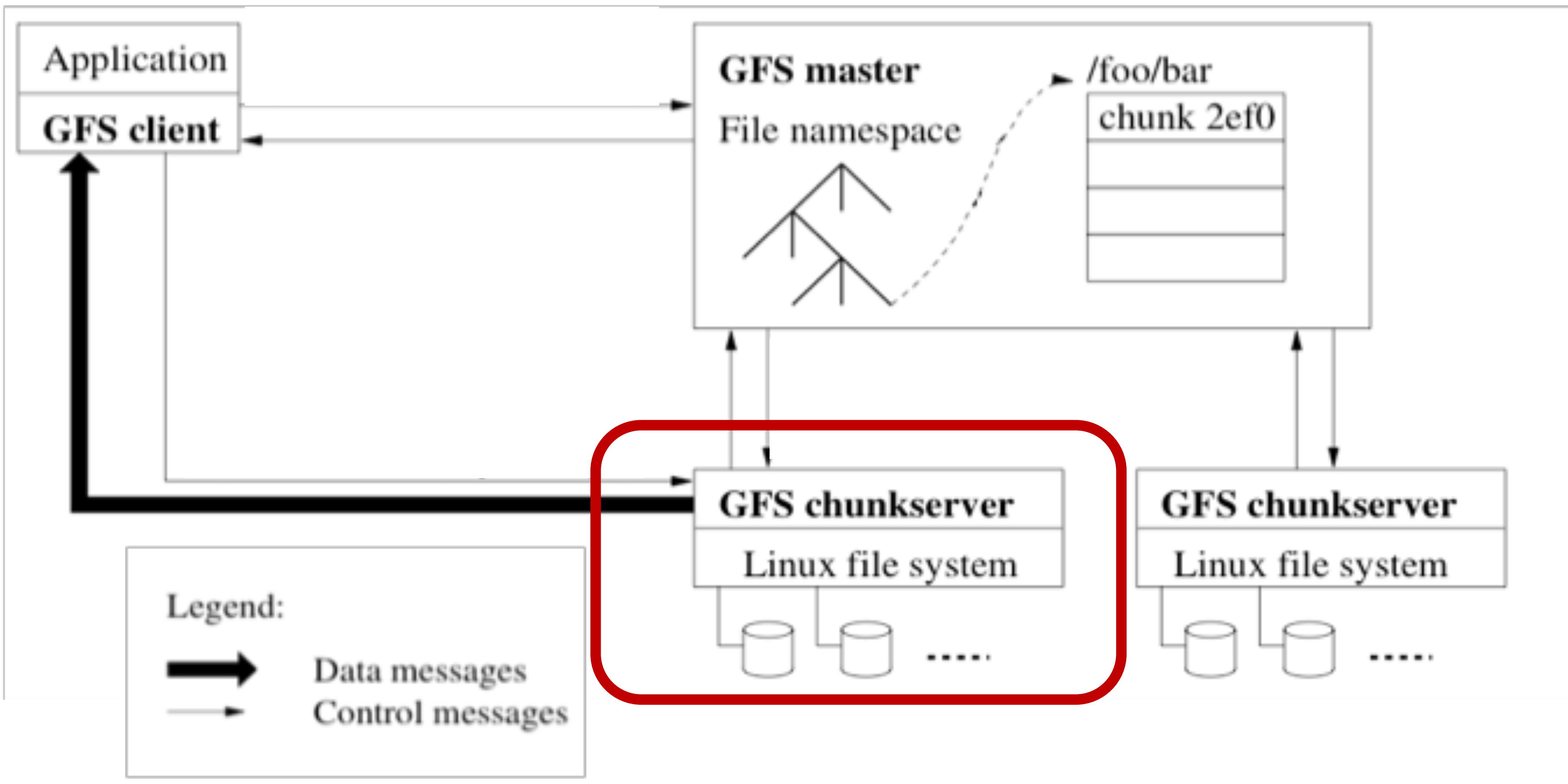


GFS Architecture: Master Server

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)
- Migrates chunks between chunkservers
  - Why is migration needed?
- Controls consistency management
- Garbage collects orphaned chunks

High-Level Picture of GFS Architecture



GFS Architecture: Chunkserver

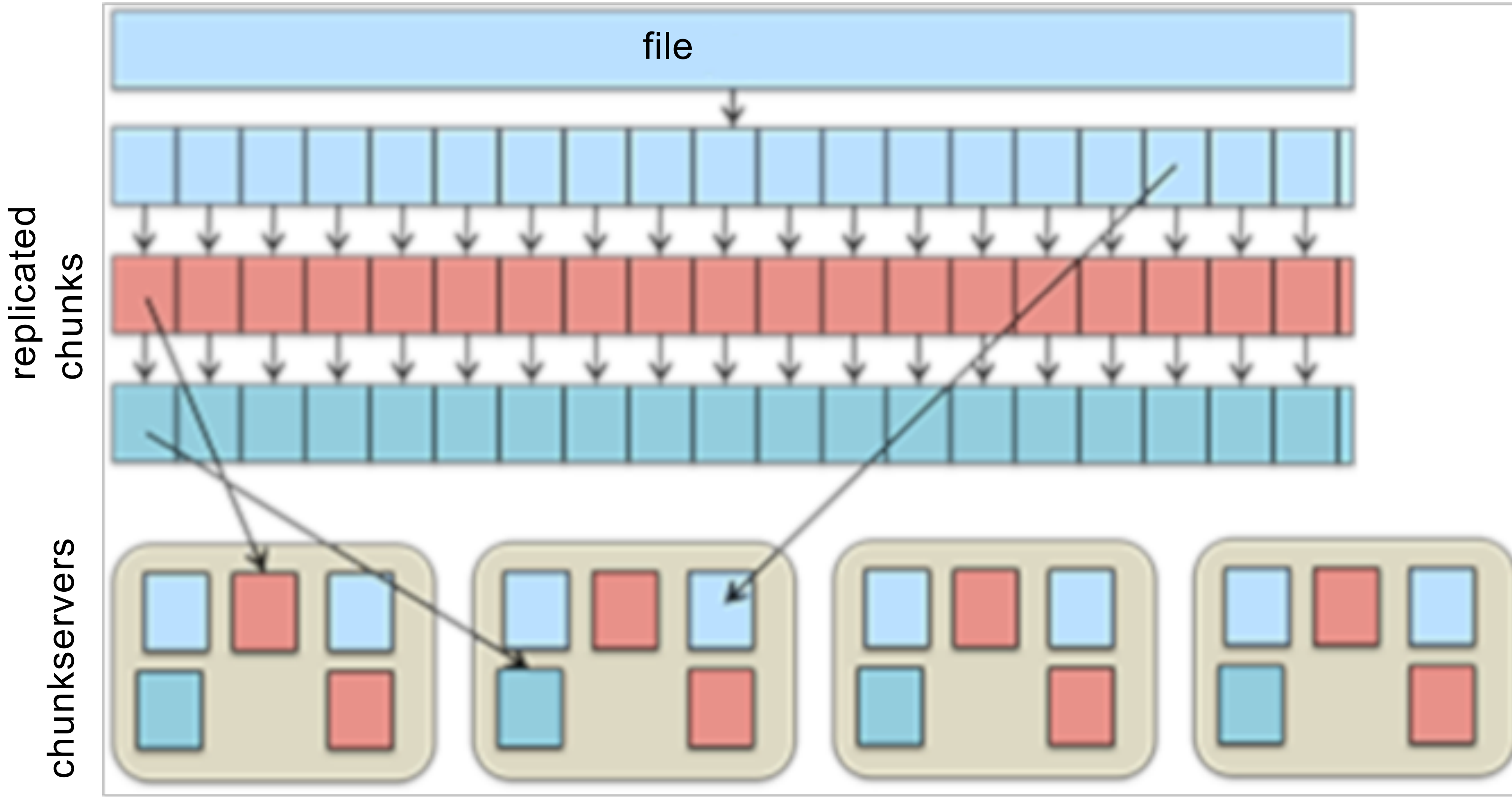
- Stores file chunks (e.g., 64 MB in size) on local disk using standard Linux filesystem (like Ext4), each with version number and checksums

> Why 64MB, and not traditional block size?

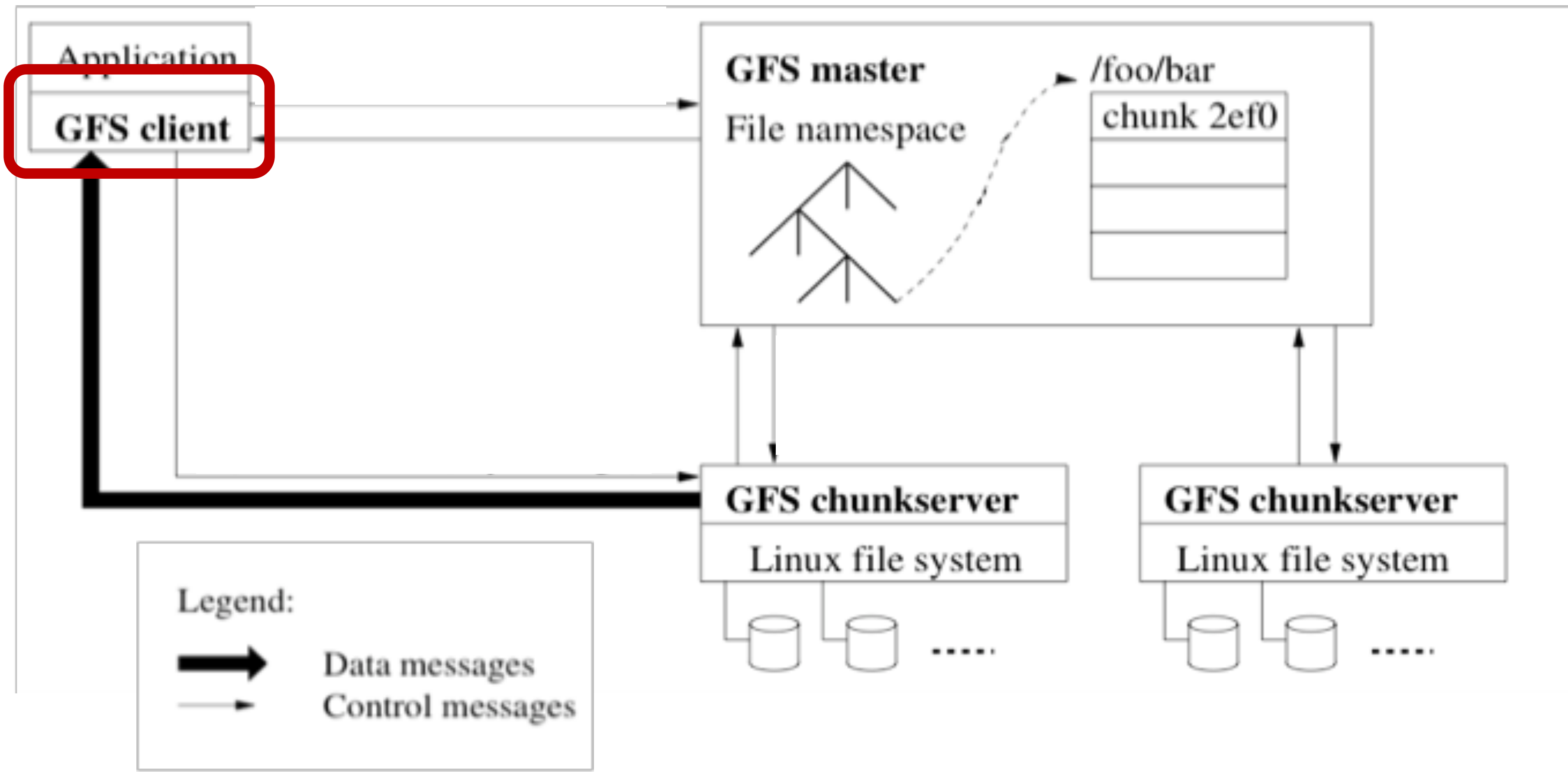
⇒ To reduce GFS overhead per chunk

- No understanding of overall distributed file system (just deal 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



High-Level Picture of GFS Architecture



GFS Architecture: Client

- Issues control (metadata) requests to master server
- Issues data requests directly to chunkservers

- **No caching of data**
  - Streaming reads (read once) and append writes (write once) don't benefit much from caching at client
  - Simplifies client and overall system: No cache coherence issues
- **Caches metadata**
  - E.g., Chunkserver associated to a chunk

GFS Architecture: Client

- No file system interface at the operating-system level
  - Not a traditional in-kernel file system
  - 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
  - **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

  - **Snapshot:** creates a copy of a file or directory at low cost

Outline: GFS

- Motivation and design goals
- Architecture
- Client operations
- Fault tolerance
- Consistency model
- Post-GFS

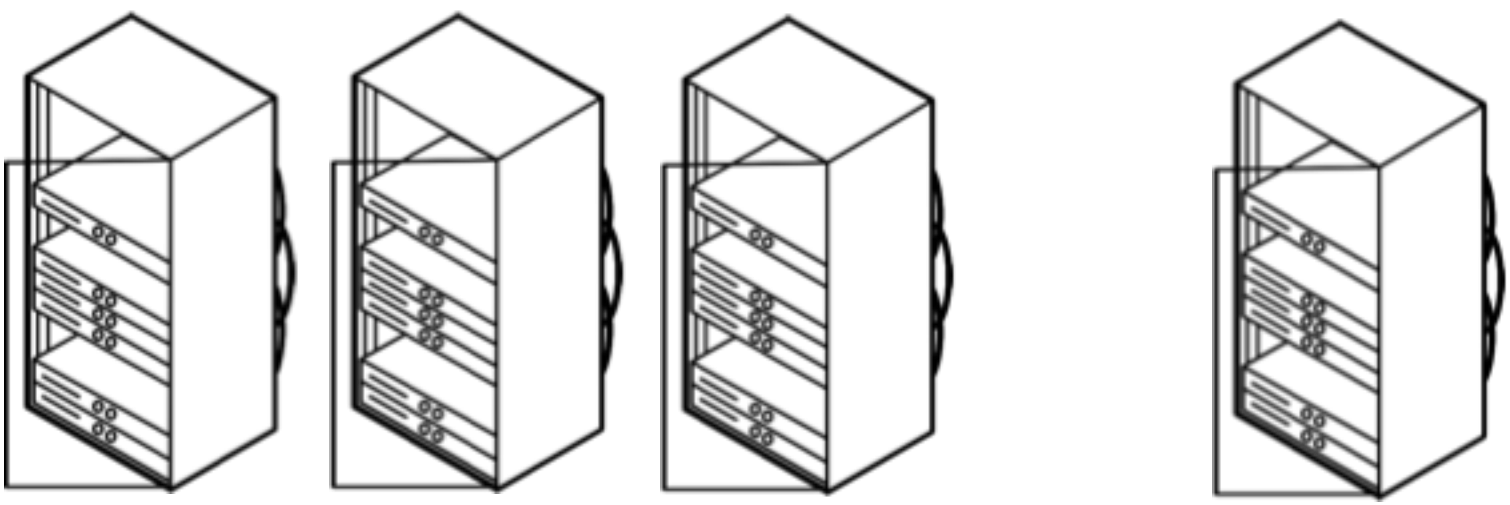
GFS Client: Read Operation

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

GFS Client: Write Operation

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

How to ensure consistent writes to all replicas?



GFS Client: Write Operation

- Some chunkserver is primary for each chunk
  - Managed via leases
  - Master grants lease to primary (typically for 60 sec.)
  - Leases renewed via piggybacking over periodic heartbeat messages between master and chunkservers
- Client asks master for primary and secondary replicas for each chunk
  - Response cached at client

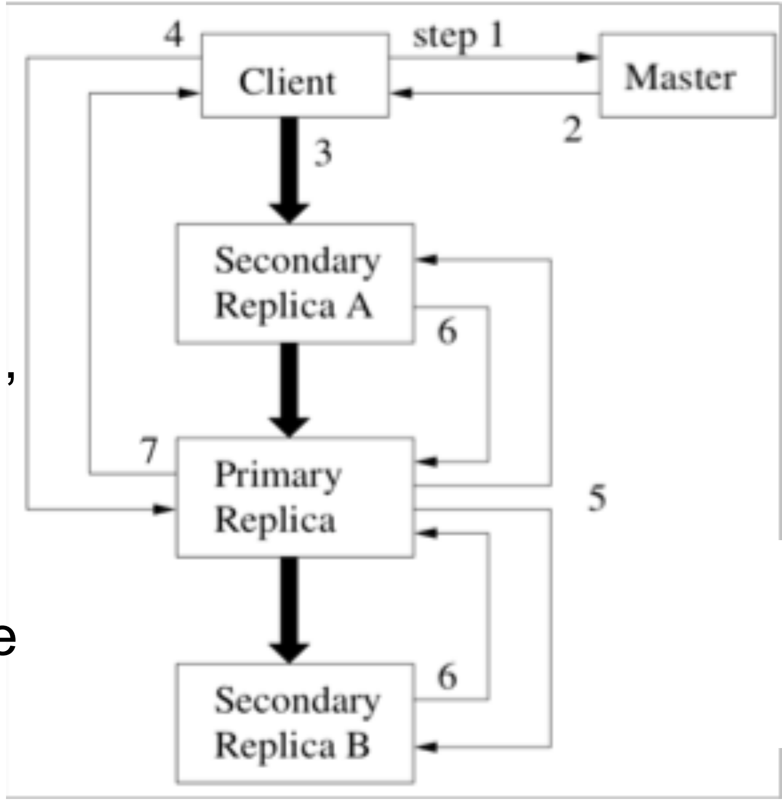
How to efficiently send write data to all three replicas?

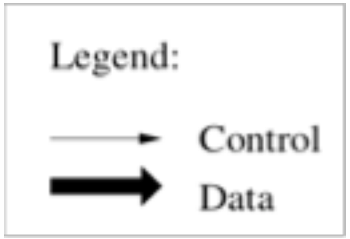
- Client sends data to replicas in daisy chain
  - Pipelined: each replica forwards as it receives

GFS Client: Write Operation

- Clients get metadata, daisy-chains data
- All replicas acknowledge receiving data write to client
- Doesn't write to file → just buffers data

- Client sends write request (chunk handle, offset) to primary → commit phase
- Primary assigns **serial numbers** to write requests, 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 Client: Write Operation

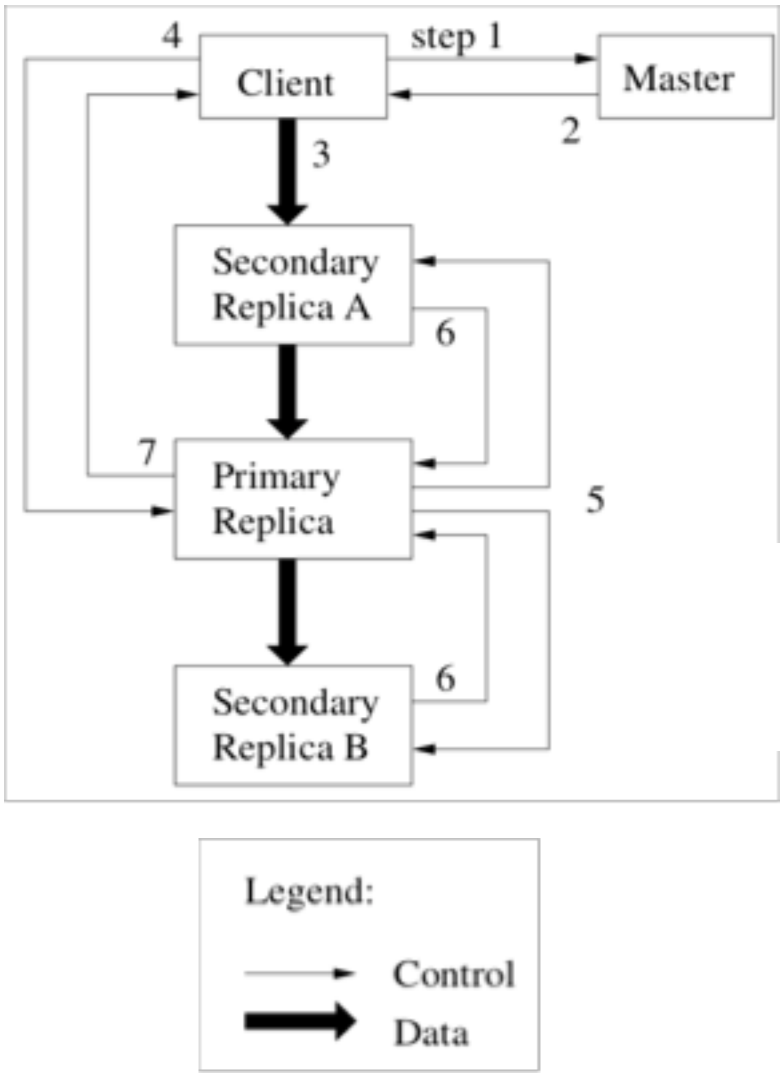
Key points:

- Data pushed linearly along a chain
- Flow of data decoupled from flow of control

Why?

Helps to

- fully utilize each machine's network bandwidth
- avoid network bottlenecks and high-latency links
- minimize the latency to push through all the data.



GFS Client: Record Append Operation

- Large files used as queues between multiple producers and consumers
  - Need atomic append operation

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

⇒ multiple clients might use GFS write (client offset) operation to write records to the same region

⇒ Avoid using complex and expensive synchronization among clients (e.g., distributed lock manager)

- Client pushes data to last chunk's replicas; sends append request to primary without specifying byte offset

GFS Client: Record Append Operation

- 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
- 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

GFS Client: Record Append Operation

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

- Replicas of same chunk may contain different data
  - Can contain duplicates of all or part of record data
  - Some regions of a chunk consistent and some not

- Semantics?
- Data written **at least once** in atomic unit

  ⇒ GFS client retries until success

Outline: GFS

- Motivation and design goals
- Architecture
- Client Operations
- Fault tolerance
- Consistency model
- Post-GFS

GFS Fault Tolerance

High Availability

- Chunk replication
  - Each chunk is replicated on multiple chunkservers

- Master (i.e., state of the master) replication
  - Operation log and checkpoints replicated on multiple machines

Data Integrity

- Checksum checks
  - Each chunk has checksums
  - Checksum verified for every read and write
  - Checksum also verified periodically for inactive chunks

GFS Fault Tolerance: Chunkserver

Chunkservers can be temporarily down or fail

Insufficient chunk replicas

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

Stale chunks

- Chunks have version numbers
  - Stored on disk at master and chunkservers
  - Each time master grants new lease to primary, increments version, informs all replicas
- Detect outdated chunks with version number
  - Outdated chunks are ignored and garbage collected

GFS Fault Tolerance: Master

What if GFS loses the master?

- Master has all metadata information
  - Lose master = lose the filesystem
- Master logs metadata updates to disk sequentially (→ WAL)
- Replicates log entries to remote backup servers
- Only replies to client after log entries safe on disk on self and backups

GFS Fault Tolerance: Master

- Replays log from disk
  - Recovers namespace (directory) and 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 newer chunk, adopt its version number
  - Master may have failed while granting lease

- Logs cannot be too long – why?
  - Master uses log to rebuild the filesystem state at startup
- How to avoid too long logs?
  - Periodic checkpoints taken to keep log short

Outline: GFS

- Motivation and design goals
- Architecture
- Client Operations
- Fault tolerance
- Consistency model
- Post-GFS

GFS Consistency Model

- Changes to namespace (i.e., metadata) are atomic
  - E.g., file creation
  - Due to: namespace locking (granular) + operation log

- Changes to data are ordered by a primary
  - Concurrent writes can be overwritten
  - Record appends complete at least once, at offset of GFS's choosing
    → Applications must cope with possible duplicates

GFS Consistency Model

- Failed operations can cause inconsistency
  - E.g., different data across chunk servers (failed append)
- Concurrent successful writes (to the same region) results in an “undefined” region
- Behavior is worse for writes than appends (why?)

GFS applications designed to accommodate the relaxed consistency model

- Co-design of applications and the file system

Outline: GFS

- Motivation and design goals
- Architecture
- Client Operations
- Fault tolerance
- Consistency model
- Post-GFS

Post GFS

Open-source Implementation:

- Apache Hadoop Distributed File System (HDFS)
- Widely deployed in industry (esp. as underlying filesystem for data analytics clusters)

Successor at Google: Colossus

- Some of the key differences
  - Eliminates master node as single point of failure: Multiple/distributed masters
  - Improved storage efficiency: Employs **erasure coding** instead of replicas

Replication vs. erasure codes

Two data chunks to be stored: a and b

Tolerate any 2 failures

~~chunk 1 a~~
~~chunk 2 a~~
chunk 3 a
chunk 4 b
chunk 5 b
chunk 6 b

3-replication

Storage overhead = 3x

~~chunk 1 a~~
~~chunk 2 b~~
chunk 3 a+b
chunk 4 a+2b

"parity chunks"

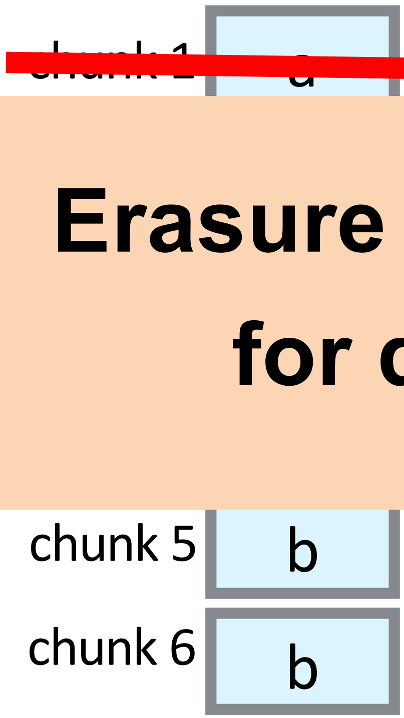
Erasure code

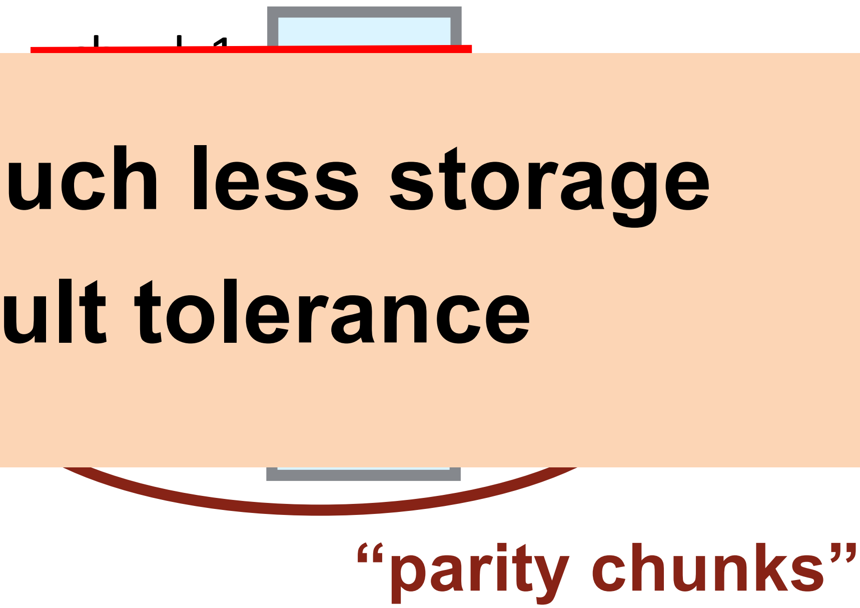
Storage overhead = 2x

Replication vs. erasure codes

Two data chunks to be stored: a and b

Tolerate any 2 failures





Erasure codes: much less storage for desired fault tolerance

3-replication

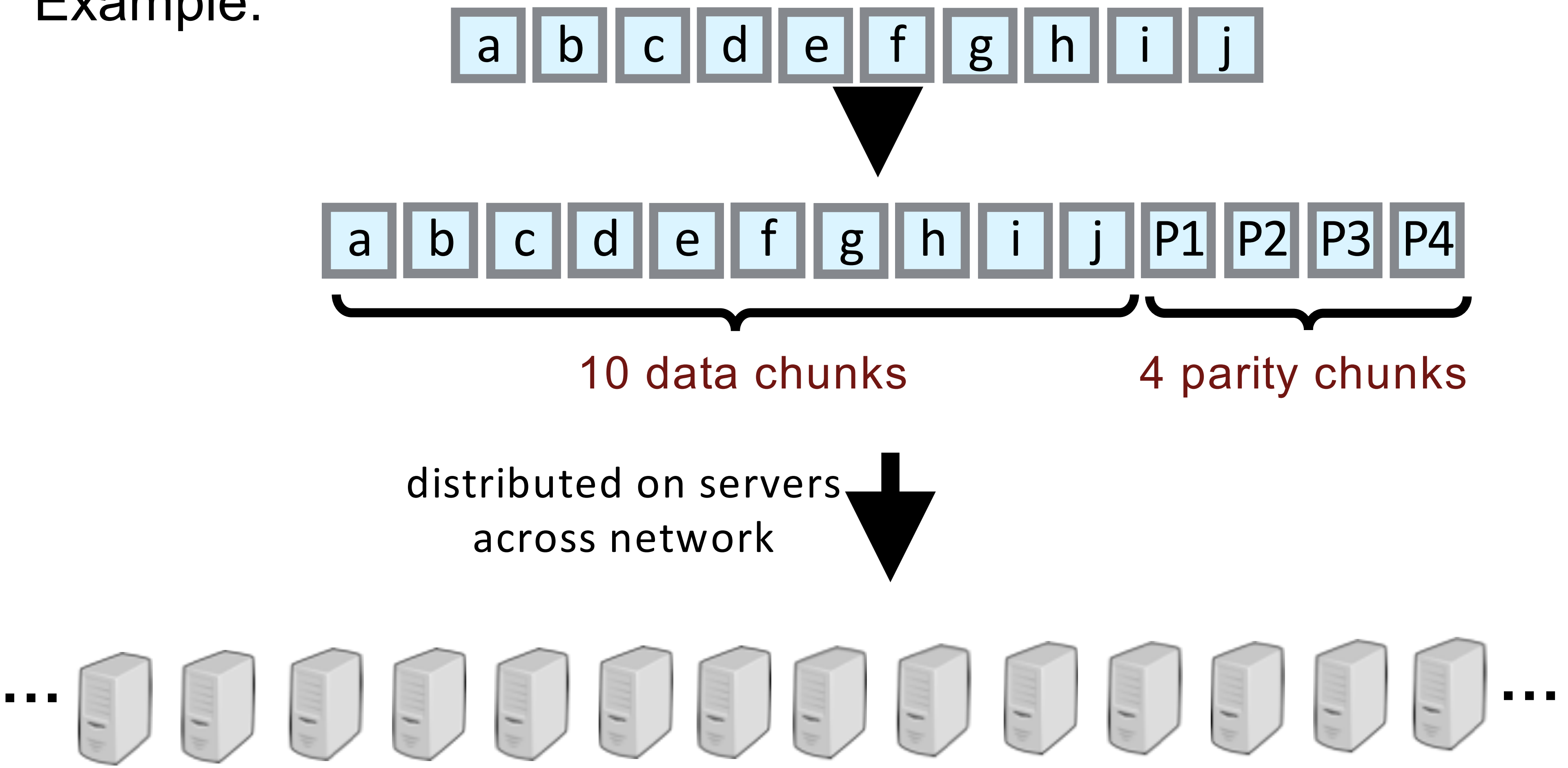
Storage overhead = 3x

Erasure code

Storage overhead = 2x

Erasure codes: how are they used in distributed storage systems?

Example:



Most large-scale storage systems use erasure codes

Facebook, Google, Amazon, Microsoft...

"Considering trends in data growth & datacenter hardware, we foresee HDFS **erasure coding** being an **important feature in years to come**"

- Cloudera Engineering (September, 2016)

Research on erasure codes for storage clusters

Mathematical structure of parities decide degree of reliability and overhead

- Traditional erasure code: Reed-Solomon code
- Recent research on erasure codes for distributed storage
  - Apache Hadoop Distributed File System (HDFS) v3.0
    - "A Piggybacking Design Framework for Read-and Download-efficient Distributed Storage Codes", IEEE ISIT 2013, IEEE Transactions on Information Theory, 2017.
    - "A "Hitchhiker's" Guide to Fast and Efficient Data Reconstruction in Erasure-coded Data Centers", ACM SIGCOMM 2014.
  - Microsoft Azure
    - "Erasure Coding in Windows Azure Storage", USENIX ATC, 2012.
    - “On the locality of codeword symbols”, Transactions on Information Theory, 2012.