CE 528 Cloud Computing

Lecture 4: Google File System
Spring 2025

Prof. Yigong Hu

BOSTON UNIVERSITY

Slides courtesy of Chang Lou

Administrivia

Next Monday is the lab day

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

SOSP 2003

Why Are We Reading This Paper?

Distributed storage is a key abstraction

GFS paper touches on many themes of this course

- parallel performance, fault tolerance, replication,

Consistency

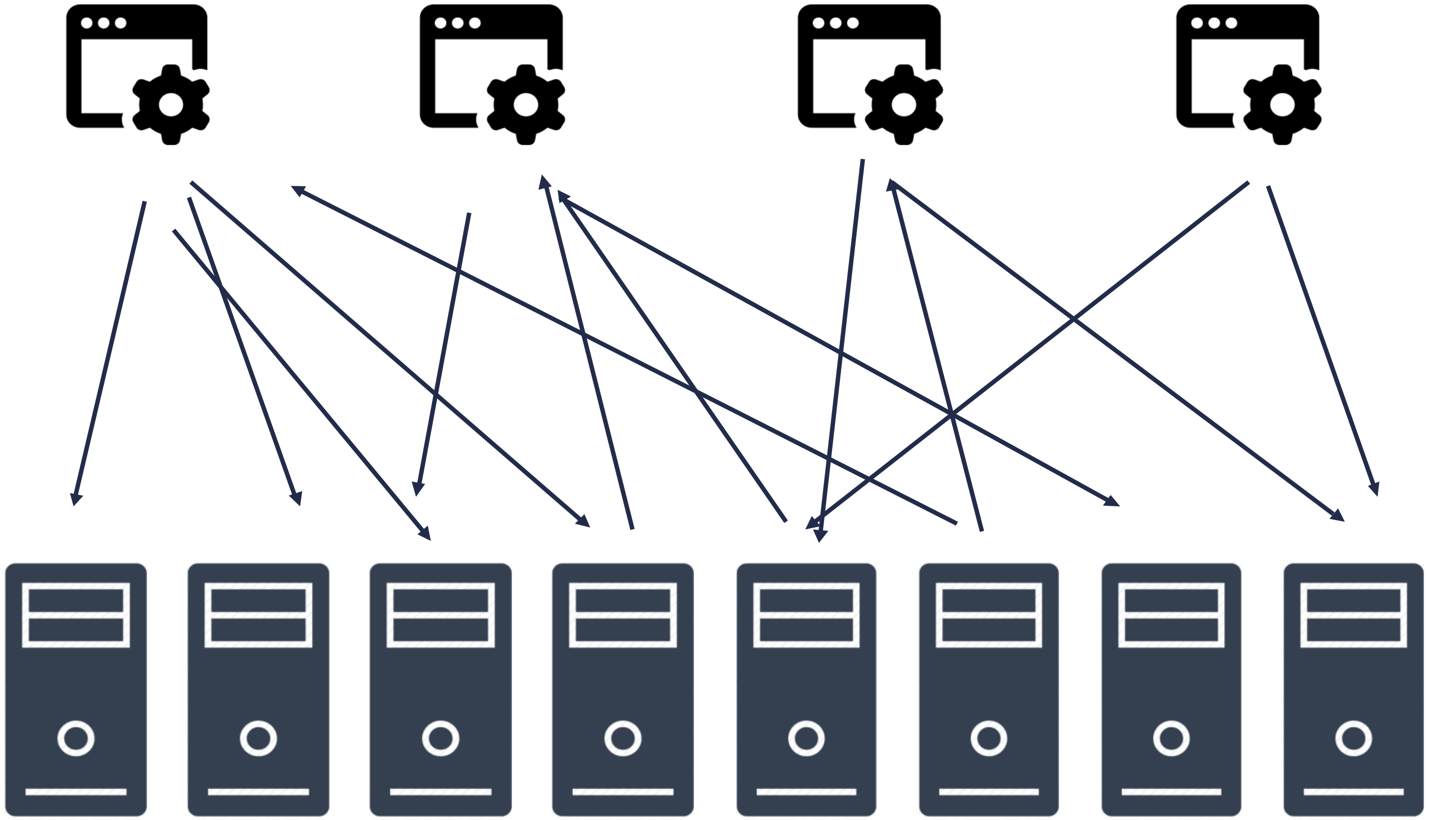
- good systems paper -- details from apps all the way to network

successful real-world design

How to Read a Research Paper

- What are the **motivations** for this work?
- What is the proposed **solution**?
- What is the work's **evaluation** of the proposed solution?
- What is your **analysis** of the identified problem, idea and evaluation?
- What are the **contributions**?
- What are **future directions** for this research?
- What **questions** are you left with?
- What is your **take-away message** from this paper?

Motivation



- Performance
- Many servers
- Fault tolerance
- Replication
- Better Consistency

Goal of GFS

Many Google services needed a big fast unified storage system

- Mapreduce, Youtube

Global (over a single data center)

- Allows sharing of data among applications

Automatic

- For parallel performance
- To increase space available
- recovery from failures

Assumptions of GFS

Just one data center per deployment

Internal Google applications/users

Workload (e.g., crawling -> indexing -> PR -> ...)

- Multiple clients
- Large streaming reads, Small random writes
- Concurrent appends to the same file

Aimed at sequential access to huge files: read or append

- I.e. not a low-latency DB for small items
- High Throughput > Low Latency

What Are the Contribution of the Paper?

Not the basic ideas of distribution, sharding, fault-tolerance.

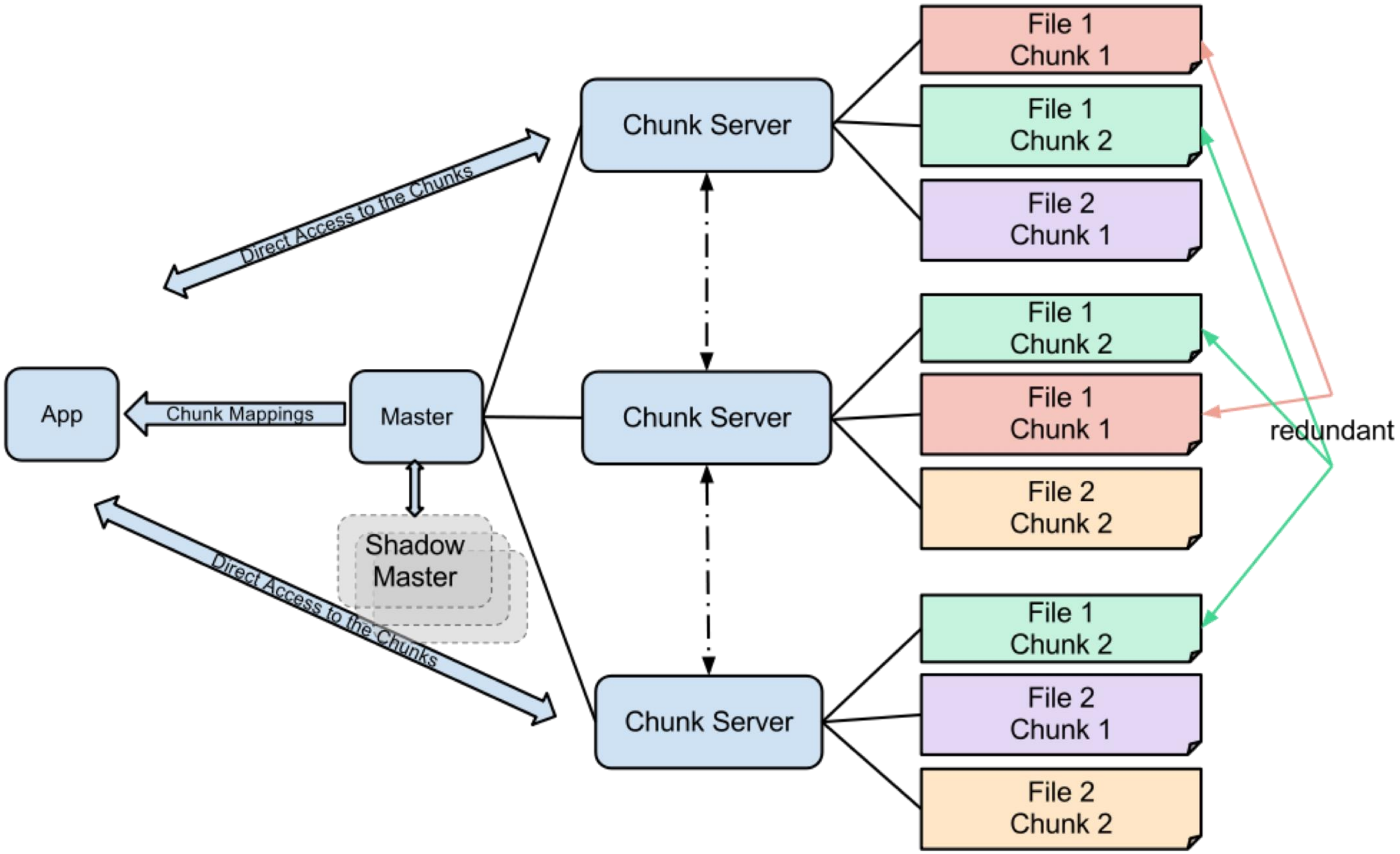
Huge scale.

Used in industry, real-world experience.

Successful use of weak consistency.

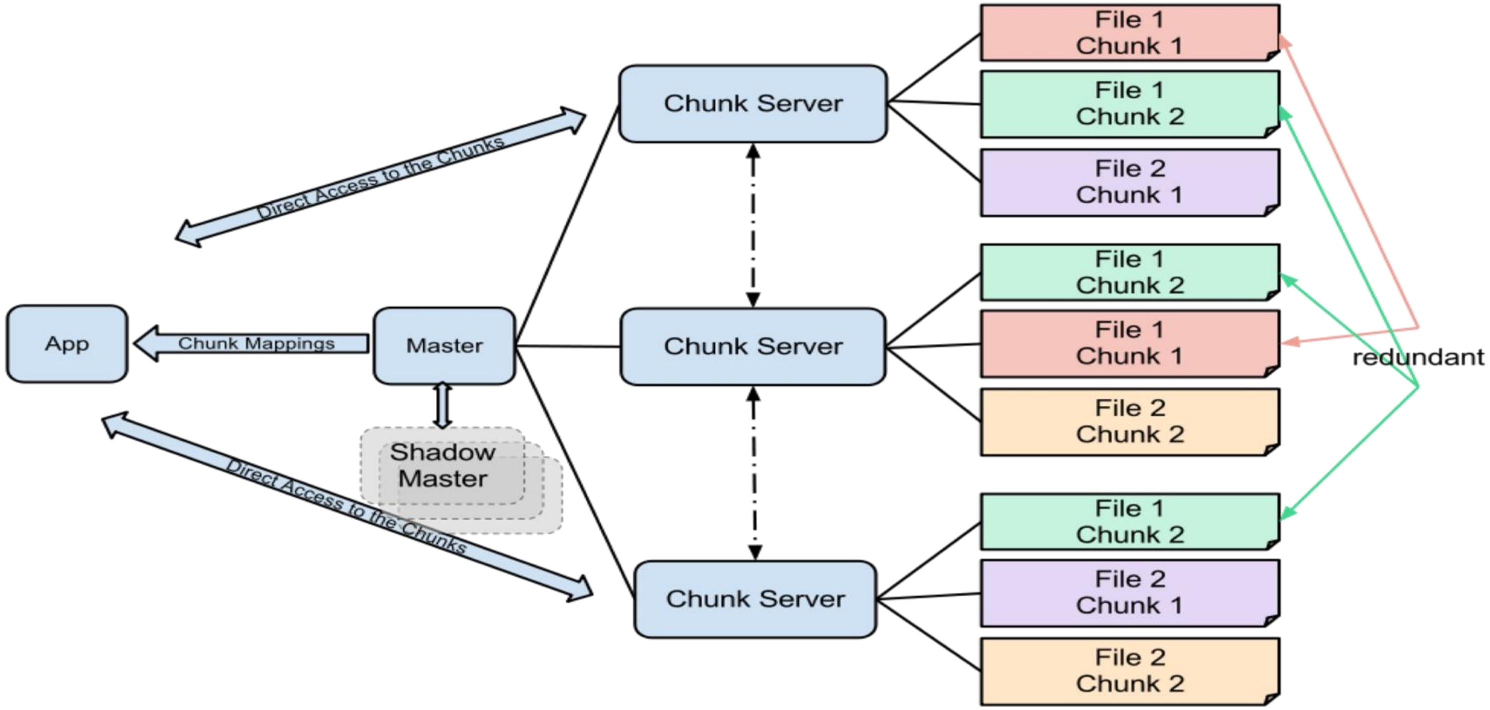
Successful use of single master.

GFS's Architecture



Overall Architecture

- User-level process running on commodity Linux machines
- Files broken into chunks (typically 64 MB),
- 3x redundancy
- Data transfers happen directly between clients and Chunk Servers
- Single Master and master replicas
- Division of Master Server and Chunk Servers



Master Node

Centralization for simplicity & global knowledge for chunk placement

Namespace and metadata management

Managing chunks

- Where they are (file<-chunks, replicas)
- Where to put new
- When to re-replicate (failure, load-balancing)
- When and what to delete (garbage collection)

Fault tolerance

- Shadow masters
- Monitoring infrastructure outside of GFS
- Periodic snapshots
- Mirrored operations log

Master Node

All Metadata is kept in Master's memory – it's fast!

- A 64 MB chunk needs less than 64B metadata => for 640 TB less than 640MB

Master learns ChunkServer-to-chunk mapping from Chunk Servers when

- Master starts
- A Chunk Server joins the cluster

Master exchanges periodic heartbeat with Chunk Servers

- state monitoring & instructions

Operation log to keep file-to-chunk mapping persistent

- Is used for serialization of concurrent operations
- Replicated in master's disk and on remote machines
- Respond to client only when log is flushed locally and remotely

Chunk Servers

64MB chunks as Linux files

- Reduce size of the master data structures
- Reduce client-master interaction
- Internal fragmentation => allocate space lazily

Fault tolerance

- Heart-beat to the master
- Something wrong => master inits replication

Basic Ops (Read, Write)

When Client Wants to Read A File?

1. C sends filename and offset to master M (if not cached)
    - M has a filename -> array-of-chunk handle table and a chunk handle -> list-of-chunk servers table

2. M finds chunk handle for that offset

3. M replies with chunk handle + list of chunk servers

4. C caches handle + chunk server list

5. C sends request to nearest chunk server chunk handle, offset

6. chunk server reads from chunk file on disk, returns to client

When Client Wants to Read a File

Clients only ask master where to find a file's chunks

- clients cache name -> chunk handle info
- coordinator does not handle data, so (hopefully) not heavily loaded

What about writes?

- Client knows which chunk servers hold replicas that must be updated.
- How should we manage updating of replicas of a chunk

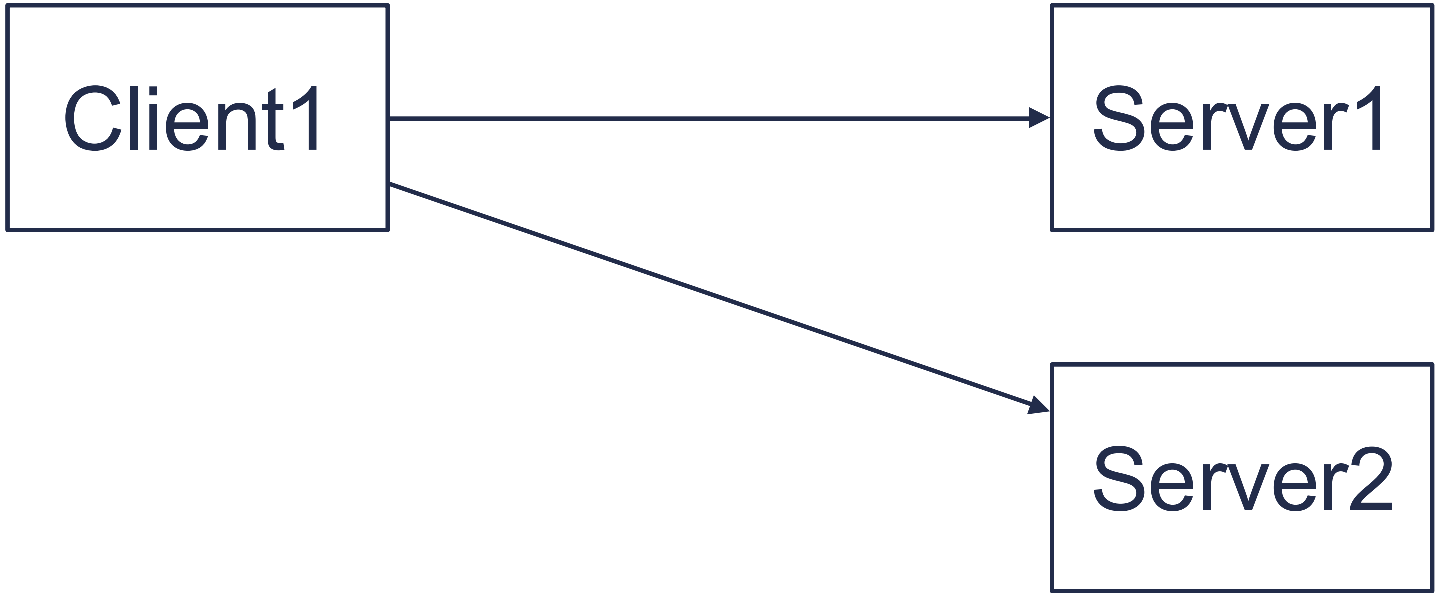
What Would We Like for Consistency?

Goal: Distributed systems try to create an illusion that users are using one single powerful machine

Suppose C1 and C2 write concurrently, and after the writes have completed, C3 and C4 read. what can they see?

Bad Replication Design

- Client sends update to each replica chunk server
- Each chunk server applies the update to its copy



Solution: Primary/Secondary Replication

For each chunk, designate one server as "primary".

Clients send write requests just to the primary.

- The primary alone manages interactions with secondary servers.
- (Some designs send reads just to primary, some also to secondaries)

The primary chooses the order for all client writes.

- Tells the secondaries -- with sequence numbers -- so all replicas
- apply writes in the same order, even for concurrent client writes.

When Client Wants to Write

1. C asks M about file's chunk @ offset

2. M tells C the primary and secondaries

3. C sends data to all (just temporary...), waits for all replies (?)

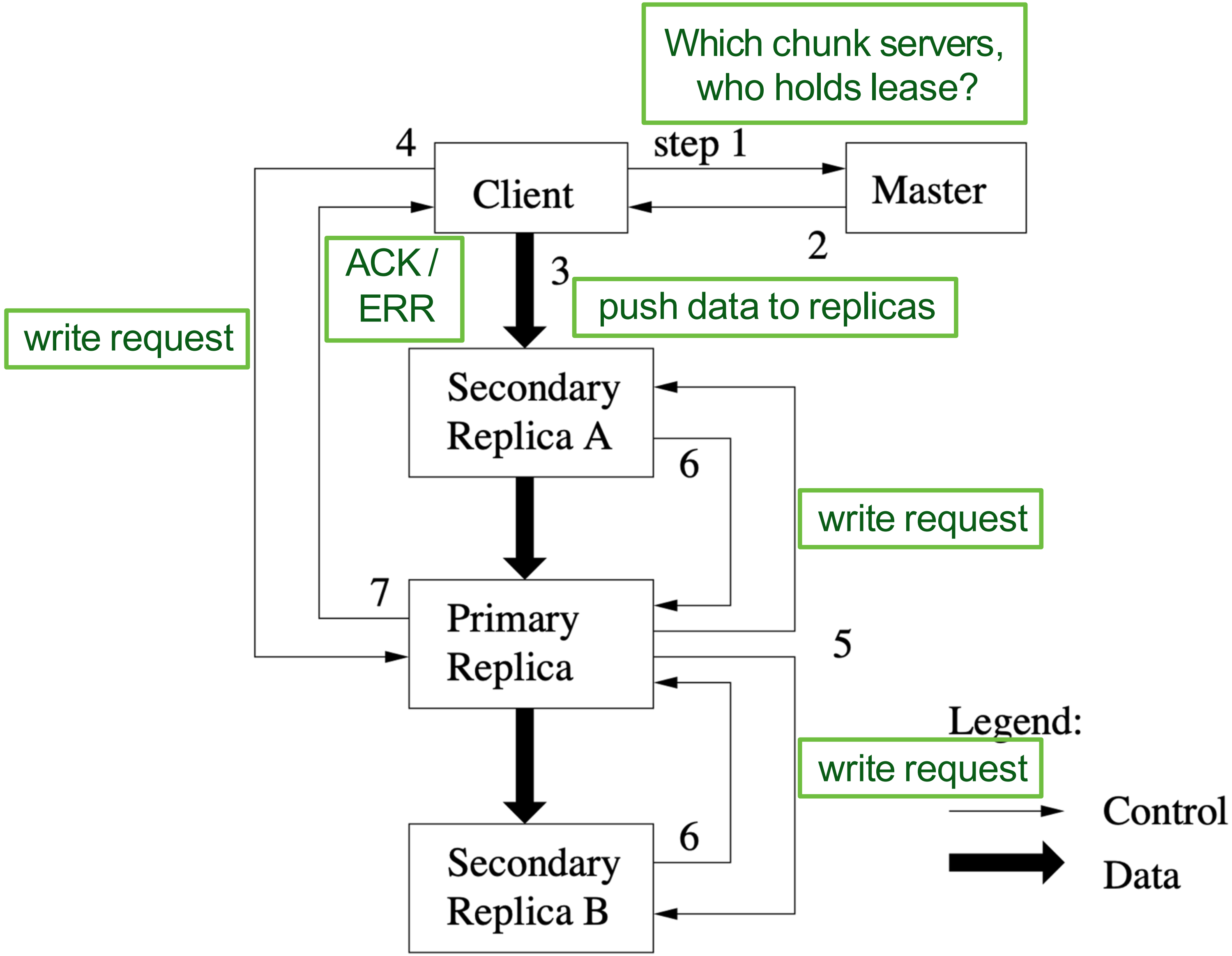
4. C asks P to write
   - P checks that lease (?) hasn't expired
   - P writes its own chunk file (a Linux file)

5. P tells each secondary to write
   - (copy temporary into chunk file)

6. P waits for all secondaries to reply, or timeout
   - secondary can reply "error" e.g. out of disk space

7. P tells C "ok" or "error"
   - C retries from start if error

Control and Data Flow



GFS Consistency Guarantees

somewhat complex!

if primary tells client that a write succeeded,

- and no other client is writing the same part of the file,
- all readers will see the write.
- "defined”

if successful concurrent writes to the same part of a file,

- and they all succeed, all readers will see the same content,
- but maybe it will be a mix of the writes.
- "consistent"
- E.g. C1 writes "ab", C2 writes "xy", everyone might see "xb".

If primary doesn't tell the client that the write succeeded,

- different readers may see different content, or none.
- "inconsistent"

A Client Crashes While Writing?

Either it got as far as asking primary to write, or not.

A Secondary Crashes Just As The Primary Asks It to Write?

1. **Primary may retry a few times, if secondary revives quickly**
    - with disk intact, it may execute the primary's request
    - and all is well.

2. **Primary gives up, and returns an error to the client**.
    - Client can retry -- but why would the write work the second
    - time around?

3. **Coordinator notices that a chunkserver is down.**
    - Periodically pings all chunk servers.
    - Removes the failed chunkserver from all chunkhandle lists.
    - Perhaps re-replicates, to maintain 3 replicas.
    - Tells primary the new secondary list.

A Secondary Crashes Just As The Primary Asks It to Write?

Re-replication after a chunkserver failure may take a **long time**

- Since a chunkserver failure requires re-replication of all its chunks. 80 GB disk, 10 MB/s network -> an hour or two for full copy.
- So the primary probably re-tries for a while,
- and the master lets the system operate with a missing
- chunk replica, before declaring the chunkserver permanently dead.
- How long to wait before re-replicating?
- Too short: wasted copying work if chunkserver comes back to life. Too long: more failures might destroy all copies of data.

What If a Primary Crashes?

The master must be able to designate a new primary if the present primary fails.

But the coordinator cannot distinguish "primary has failed" from "primary is still alive but the network has a problem."

What if the coordinator designates a new primary while old one is active?

- two active primaries!
- C1 writes to P1, C2 reads from P2, doesn't seen C1's write!
- called "split brain" -- a disaster

What If a Primary Crashes?

Solution: Lease

- Permission to act as primary for a given time (60 seconds).
- Primary promises to stop acting as primary before lease expires.
- Coordinator promises not to change primaries until after expiration.
- Separate lease per actively written chunk.

Leases help prevent split brain:

- Coordinator won't designate new primary until the current one is
- guaranteed to have stopped acting as primary.

What If a Primary Crashes?

Remove that chunk server from all chunk handle lists.

For each chunk for which it was primary,

- wait for lease to expire,
- grant lease to another chunk server holding that chunk.

Wanna Play With GFS Yourself?

Try HDFS (an open-source clone inspired by GFS)!