CE 528 Cloud Computing

Lecture 4: Google File System
Fall 2026

Prof. Yigong Hu

Demo 1: Design Proposal

Wednesday 09/23, due 12:00 noon in branch demo-1

Next Monday (09/21) is a lab day. There is no lecture.

Criterion	Weight	What we look for
Design proposal	**50%**	Clear problem and proposed design, an architecture diagram, measurable success, and verifiable milestones for Demo 2, Demo 3, and the final. A working system is not required.
Slides	**25%**	Problem and motivation, design, evidence, clarity, and finishing within 10 minutes.
Challenge	**25%**	The technical difficulty and ambition of the proposed system.

Later progress grades use the milestones your team commits to here.

Presentation Grading

Rule	Policy
Time	10-minute presentation, followed by 5 minutes of Q&A
Presenters	At most two presenters per demo. Every team member presents at least once across Demo 1, Demo 2, and Demo 3.
Individual score	Presentation quality, clarity, engagement, finishing on time, and answering questions
Multiple demos	**If you present more than once, your highest presentation score counts.**
Course weight	10% of the course grade. The final presentation is graded separately.

AWS Credits: One Code per Account

AWS accepts only ONE code per account

- Redeeming all codes into one team account does not work

What to do instead

- Each teammate redeems one code in their own account
- Run the project in one account at a time
- When its credit runs low, move to the next teammate's account
- Create resources with scripts, so moving is easy
- After moving, delete everything in the old account

Details: Setup page on the course website

Why Storage?

Storage is the layer the rest of the stack stands on

- MapReduce, last lecture, reads its input and writes its output to GFS
- BigTable, Google's table store, keeps its files in GFS as well
- Three papers in three years: GFS 2003, MapReduce 2004, BigTable 2006

Get the storage right and the systems above it get simpler

- Your own project has to keep its state somewhere too

One Machine Is Not Enough

A crawled copy of the web in 2004: roughly 400 TB

- About 20 billion pages at around 20 KB each

One disk of that era read at 30-35 MB/s

- Reading 400 TB through a single disk takes about four months
- Storing it takes on the order of a thousand disks

So the data has to be spread over many machines

- Many disks read in parallel, and one file can outgrow any single disk

Then Everything Fails

At this scale, commodity machines are the only affordable option

- Far better performance per dollar than reliable, expensive hardware
- Expensive hardware fails less often, but it still fails

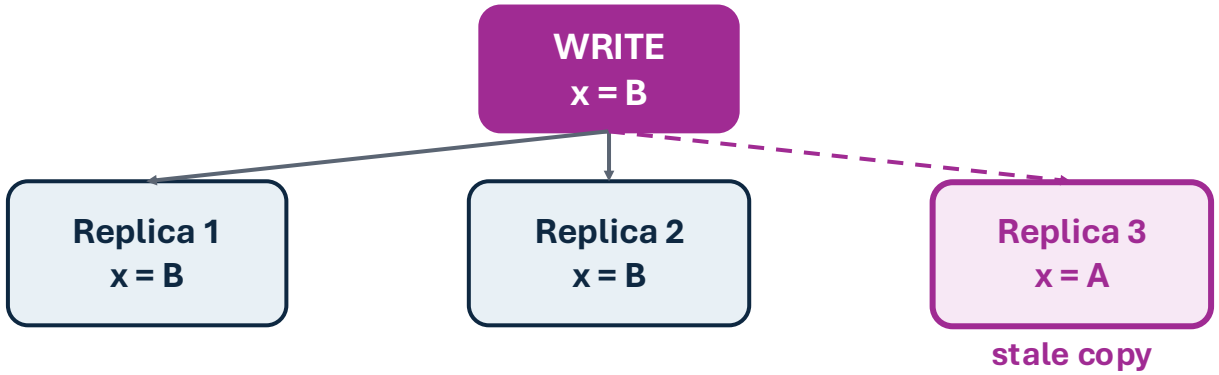
One Google cluster, first year of service

- Around 20 rack failures, each taking 40-80 machines away for hours
- About a thousand machine failures and thousands of disk failures
- Plus rewiring, router reloads and network blips

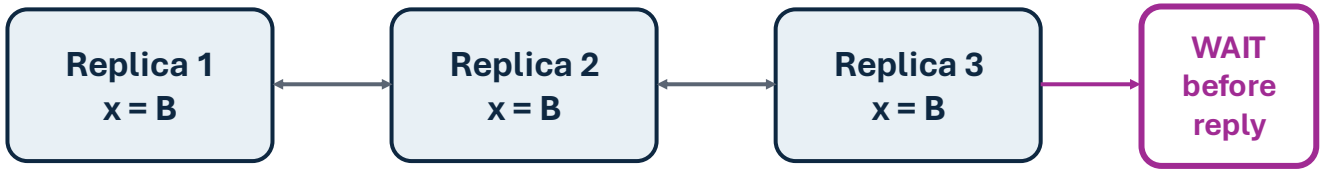
Reliability has to come from the software

Fault Tolerance Comes with a Cost

A partial write can make replicas disagree



Stronger consistency makes replicas coordinate



extra messages + waiting → lower performance

- Fault tolerance requires replication
- Replication risks inconsistency
- Consistency requires coordination
- Coordination lowers performance

Two Questions for This Lecture

How can data remain correct when it is spread across machines that fail?

What performance cost must we pay to make that happen?

What Would We Like for Consistency?

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

That one machine does one thing at a time

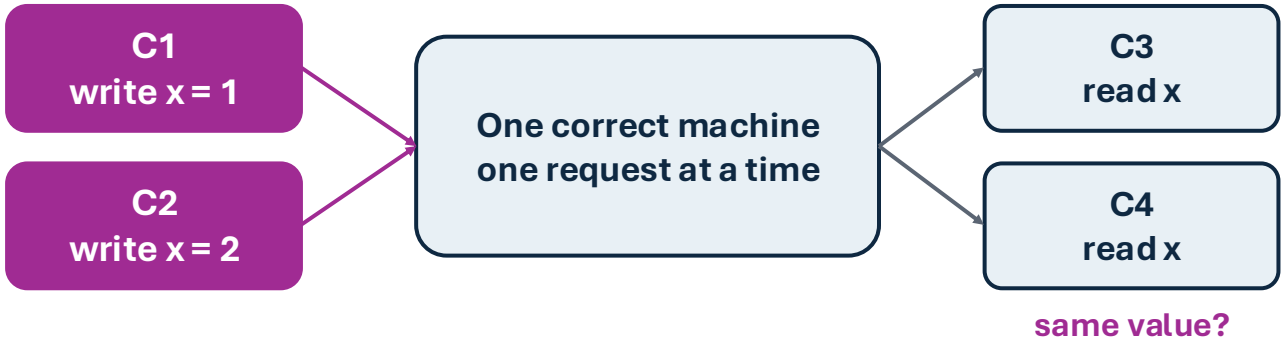
- Concurrent requests are put in some order, and the machine picks the order
- Each request finishes before the next one starts

So the test is: could one machine have produced what the clients saw?

- Not a particular answer, but an answer that fits one single order

What Should the Readers See?

C1 and C2 write at the same time, then C3 and C4 read



What are they allowed to see?

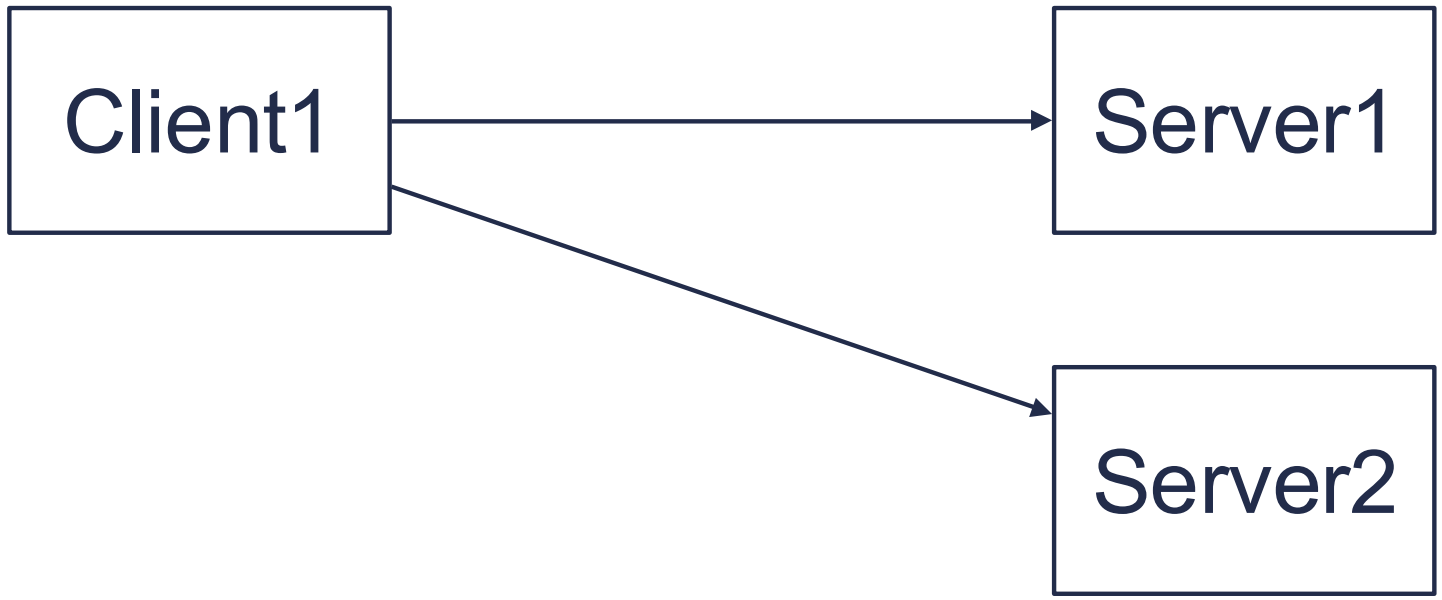
Either order is fine; disagreeing is not

- Suppose C1 and C2 write concurrently, and after the writes have completed, C3 and C4 read. what can they see?
- One correct machine would pick an order and stick to it
- Either order is acceptable
- The two readers disagreeing is not

Bad Replication Design

Client sends update to each replica chunk server

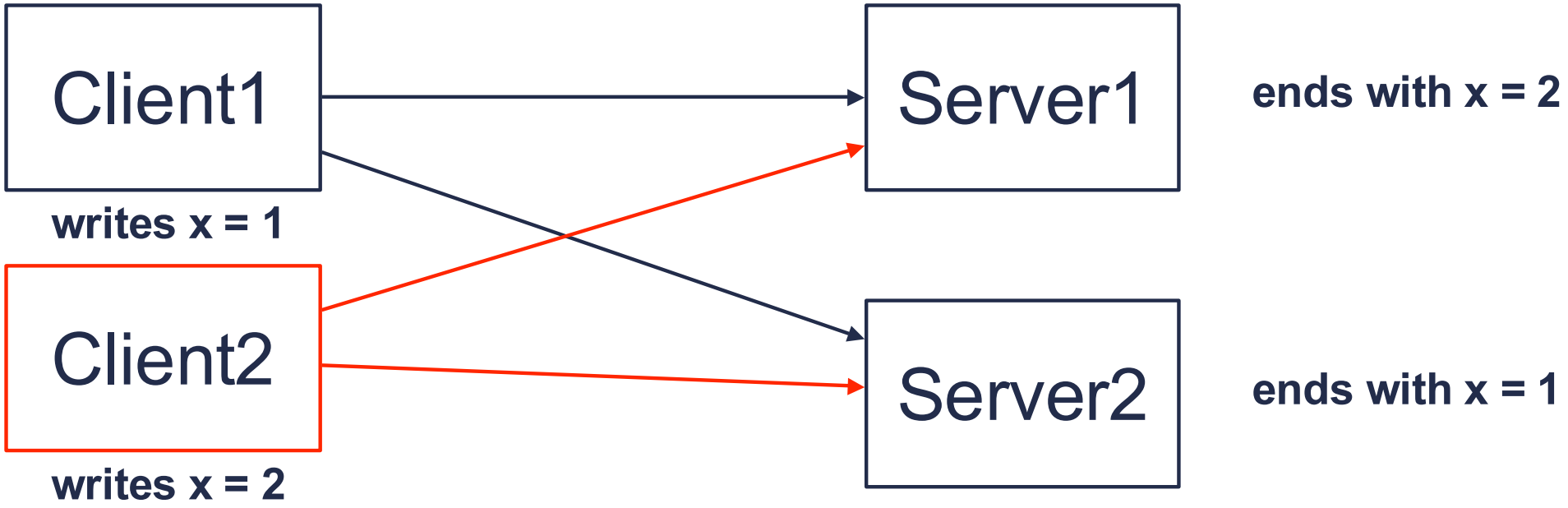
Each chunk server applies the update to its copy



What Can Go Wrong?

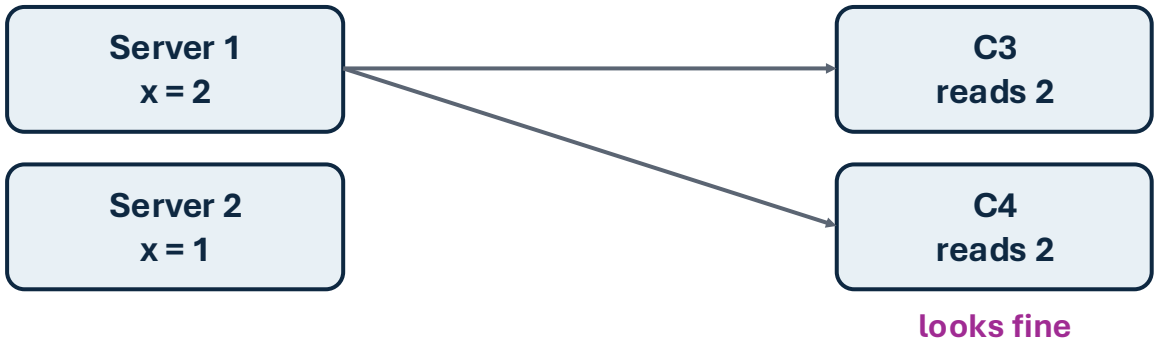
***Two* clients write the same data at the same time**

- i.e. "concurrent writes"
- Chunk servers may see the updates in different orders!
- Again, the risk is that, later, two clients may read different content

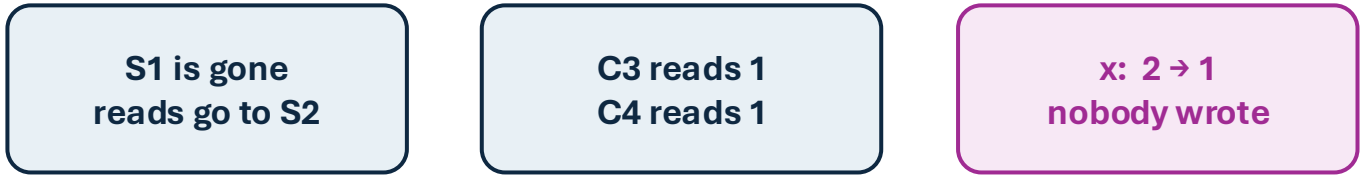


Can We Just Always Read from Server 1?

Rule: always read S1, fall back to S2



Then Server 1 fails



Nobody wrote, yet the value changed

- A natural patch: everyone reads Server 1, and falls back to Server 2 only when S1 is down
- While S1 is up every reader sees 2, so the disagreement is hidden, not repaired
- S1 dies, the readers fall back to S2, and the value goes back to 1
- Nobody wrote. One machine could never do that.

The Fix Has a Price

Both failures have one root: the replicas diverged

- A fix must prevent divergence, not pick who to read

Agreement is not free

- More messages, more complexity, longer waits

So there is a range of designs, not one answer

- Stronger consistency costs performance
- Weaker consistency costs the application

GFS picks a point on it, deliberately not perfect

The Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

SOSP 2003

Why Are We Reading This Paper?

GFS touches many themes of this course

- parallel performance, fault tolerance, replication, consistency

A good systems paper: applications down to the network

A successful real-world design, judged on production experience

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.

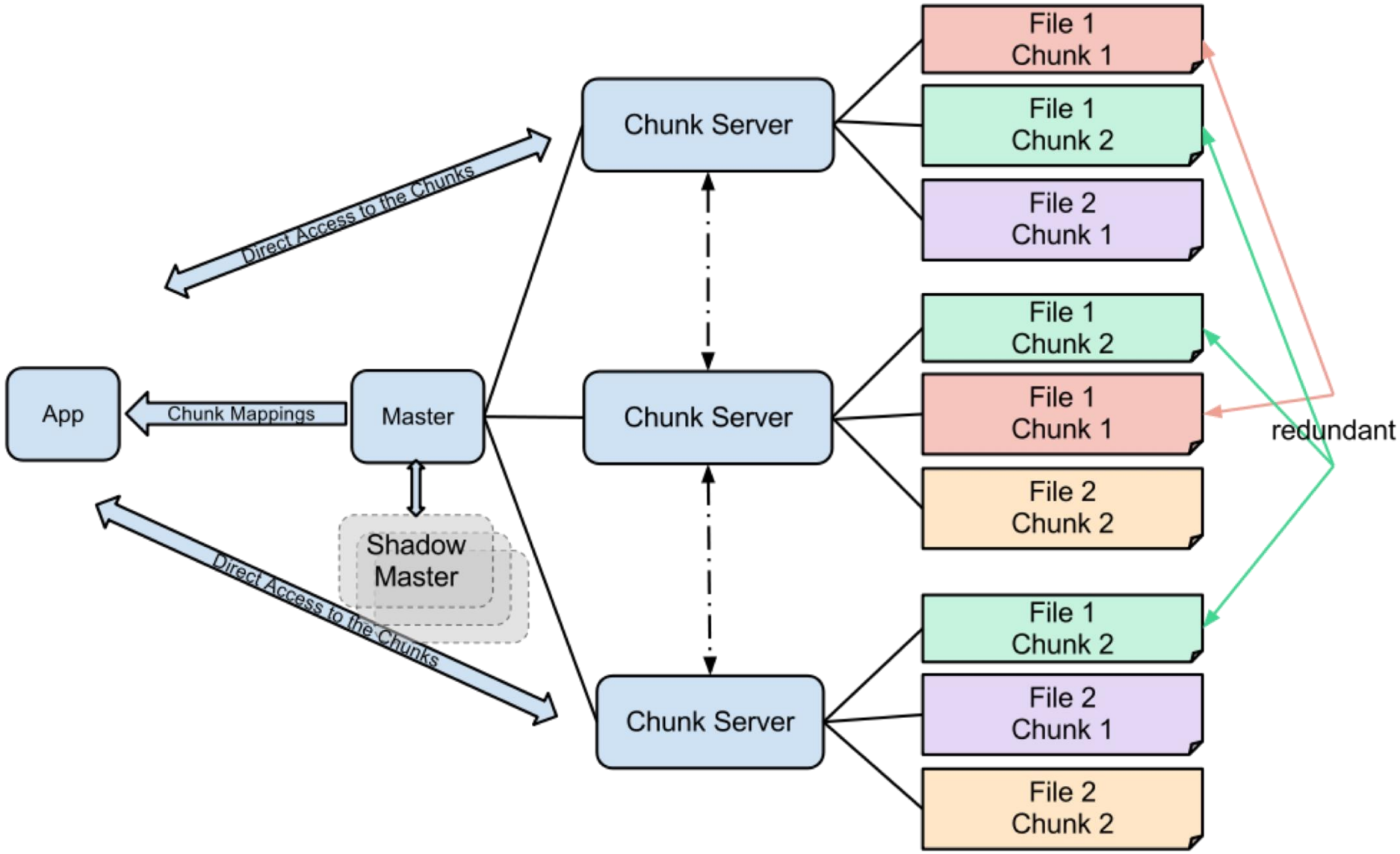
Used in industry, real-world experience.

Successful use of weak consistency.

- A missing search result goes unnoticed; ad billing does not
- Applications compensate: checksums and explicit record boundaries

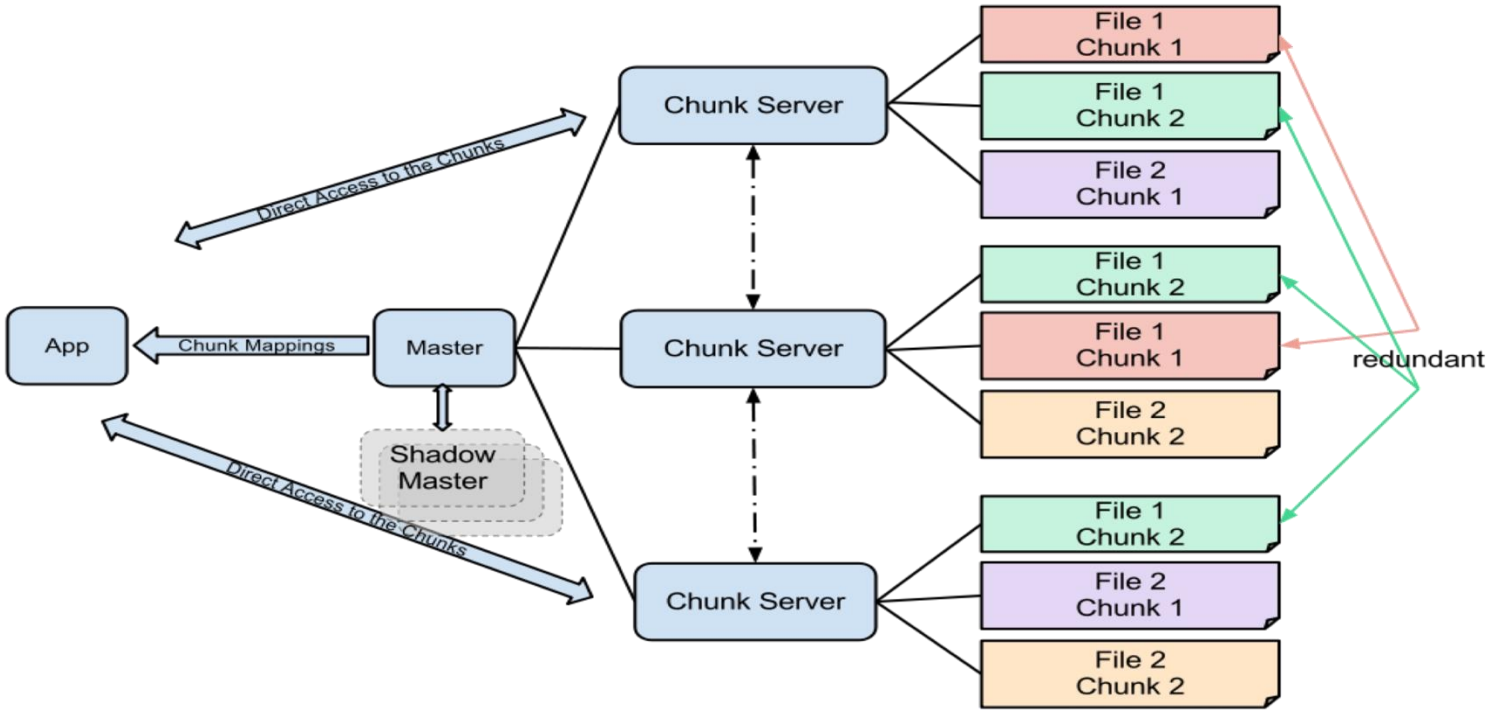
Successful use of single master.

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

One master: one place with a global view of the cluster

What it manages: the namespace, and every chunk

- Where they are: file -> chunks, chunk -> replicas
- Where to put new ones
- When to re-replicate: failure, load balancing
- When and what to delete: garbage collection

All metadata is kept in the master's memory, so lookups are fast

- Under 64 B per 64 MB chunk, so 640 TB of data needs under 640 MB

Periodic heartbeats: chunkserver state out, master instructions back

Master State: What Survives a Reboot?

In memory, so lookups are fast

- file name -> array of chunk handles
- chunk handle -> version number, replica list, primary, lease expiry

On disk: an operation log plus checkpoints

- Written: the file-to-chunk mapping, and the chunk version numbers
- Not written: replica list, primary, lease expiry - they are rebuilt

After a reboot

- Replay the log from last checkpoint, ask chunkservers what they hold
- Wait one lease time before naming any new primary

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

Demo 1: HDFS

HDFS is the open-source clone of GFS

- One NameNode: the master, names and metadata only
- Three DataNodes: the chunkservers, data only

What to watch

- A 100 MB file becomes 7 blocks, each stored on two servers
- The master knows block ids and locations, and never touches the data
- On a DataNode, a block is an ordinary file on local disk

Then one server dies and the replicas rebuild themselves

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

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

Record Append

An ordinary write names a file, an offset and a buffer

- Clients must coordinate

A record append names a file and a buffer, but no offset

- The primary picks the offset and tells every replica to use it
- Many clients can append to one file at once

Stale Replicas and Version Numbers

Every chunk carries a version number handed out by the master

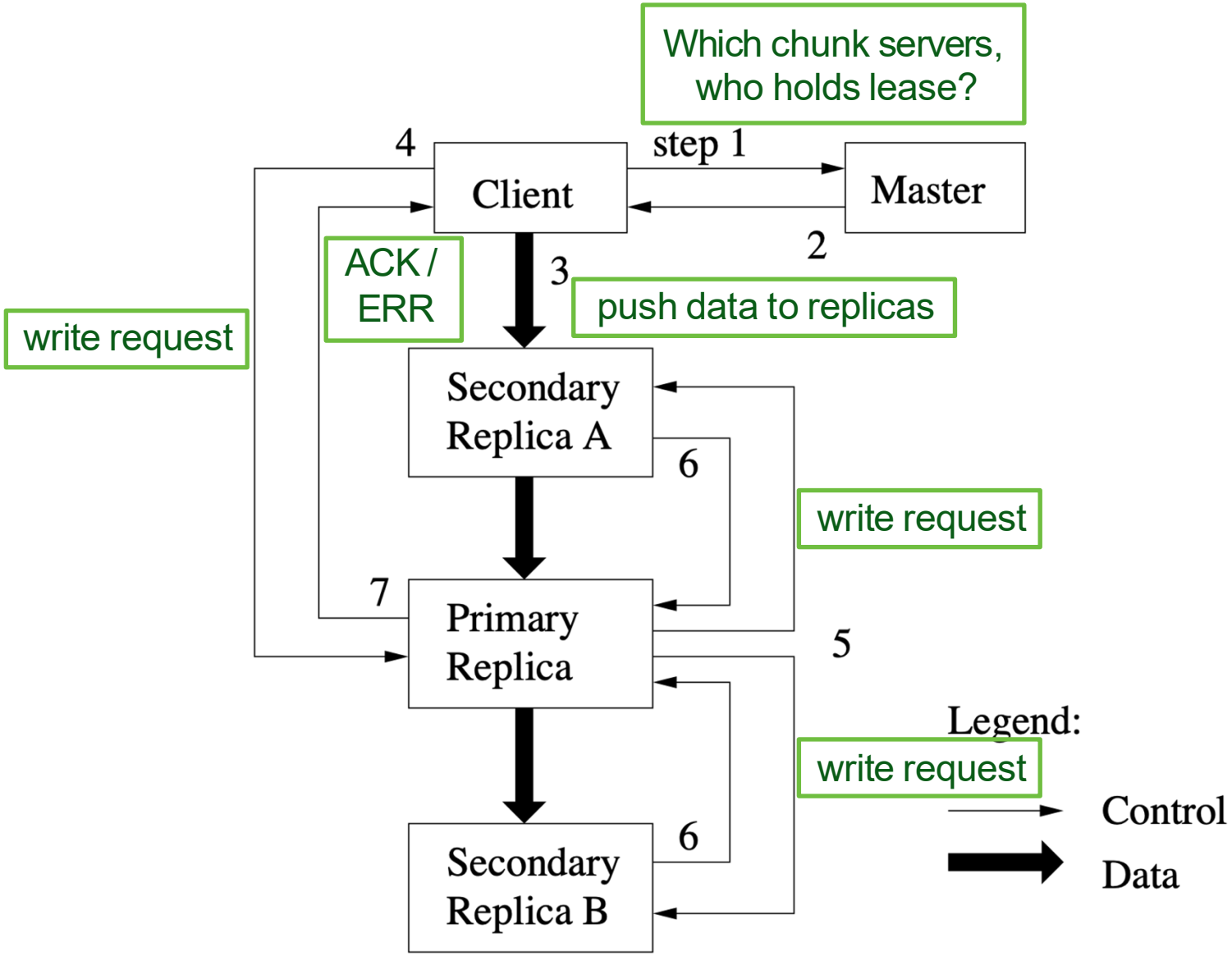
- The master bumps it whenever it grants a lease to a new primary
- It writes the new number to its own disk, then tells primary and secondaries
- Those replicas write the number to disk as well

A replica that missed an update keeps an old number

- The master never hands a stale replica to a client, and never makes it primary
- Stale replicas are garbage collected

After a power failure the master waits rather than serve stale data

Control and Data Flow



Demo 2: What a Read and a Write Actually Do

Read: ask the master once, then go straight to the data

- One getBlockLocations call covers the whole file
- Then one connection per block, to a server holding that block

Write: one addBlock per block, then a chain

- The master picks the replicas and returns them as a pipeline
- The client sends the bytes once, to the first server, which forwards them on

What the master recorded for 40 MB: a few log entries, no data

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" but "undefined"
- 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"

How Can Consistent but Undefined Arise?

Clients break big writes into multiple small writes,

- e.g. at chunk boundaries, and GFS may interleave
- them if concurrent client writes.

How Can Inconsistent Content Arise?

Primary P updated its own state.

But secondary S1 did not update (failed? slow? network problem?).

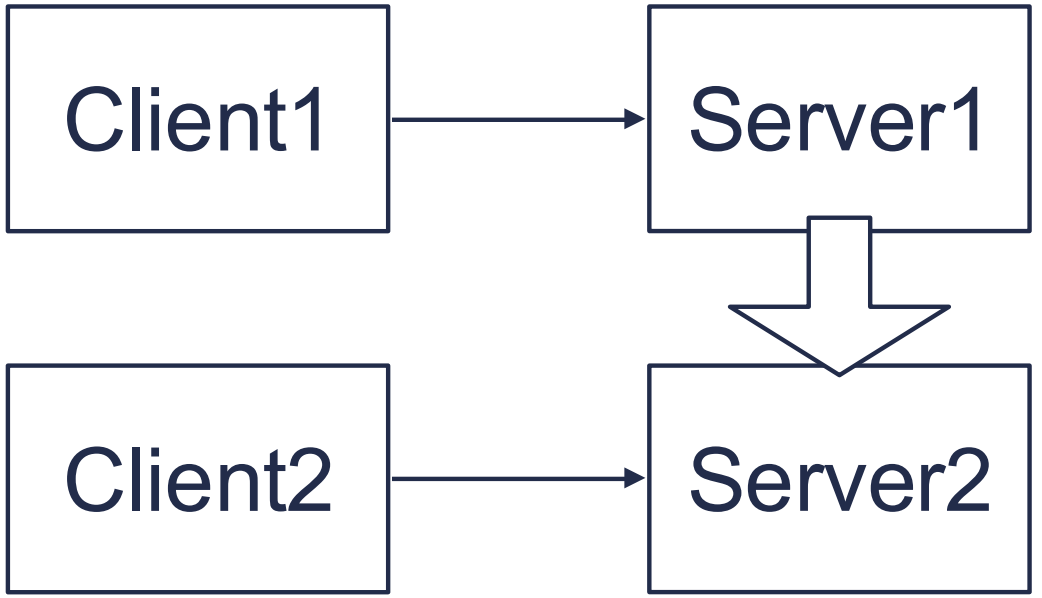
Client C1 reads from P; Client C2 reads from S1.

- they will see different results!

Such a departure from ideal behavior is an "anomaly".

But note that in this case the primary would have returned

- an error to the writing client.



Three Appends, Three Replicas

Append A: all three replicas write it, the client gets success

Append B: one replica misses it, so the client gets an error

Append C: the primary picks the next offset for everyone

The client retries B, and this time all three write it

- One replica now holds A, B, C, B; another holds A, padding, C, B
- Different readers see different orders, and B appears twice
- If the client dies before retrying, a record can live on some replicas only

Why Are These Anomalies OK?

GFS only had to serve Google's own applications

- Mostly one writer per file, or concurrent record appends
- Applications write checksums and record IDs into their own records
- Readers use them to skip padding, junk and duplicates
- MapReduce re-runs tasks anyway, so it already tolerates repeated output

Google engineers later said they should have made it more consistent

- Applications that needed order had to arrange it themselves

This is the point GFS chose on the range

- The Fix Has a Price promised a cost; these anomalies are it

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

What if the master 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.
- Master promises not to change primaries until after expiration.
- Separate lease per actively written chunk.

Leases help prevent split brain:

- Master 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.

What If the Master Crashes?

Every metadata change is logged before the master replies

- The log and the checkpoints are replicated to other machines
- A change counts as committed only once the replicas have it on disk

Shadow masters keep serving reads while the master is down

- They lag slightly behind and cannot accept mutations

In the 2003 design, replacing a dead master needed a human

- That took tens of minutes, and it is the weakest part of the paper

What Would Strong Consistency Take?

The primary would have to detect duplicate requests

A write must reach every replica, or none of them

- Keep it invisible until all replicas promise they can apply it
- Remove a secondary that cannot, instead of failing every client

A new primary must resynchronize with the secondaries before serving

Reads go to the primary, or secondaries need leases of their own

- All of this costs round trips: that is the price of the ideal model from the start of class

Takeaways

Good ideas

- One global file system as shared infrastructure for many applications
- Naming in the master, data in the chunkservers
- Sharding for throughput, huge chunks to keep metadata small
- A primary orders concurrent writes; leases stop split brain

Not so great

- One master ran out of RAM and CPU as the number of files grew
- Small files are inefficient; master failover was manual
- Weak consistency pushed work into every application

What came next: BigTable, then Colossus, which shards the master

Wanna Play With GFS Yourself?

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