Advanced Cloud Computing

Distributed Storage System

Wei Wang
CSE@HKUST
Spring 2022

THE DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING
計算機科學及工程學系


How should the big data be stored, before we crunch them?
Source: Google

Can we just have a BIG disk?

Motivating app: Web search

Crawl the whole web

Store it all on “one BIG disk”

Process users’ searches on “one BIG CPU”

Does it scale?

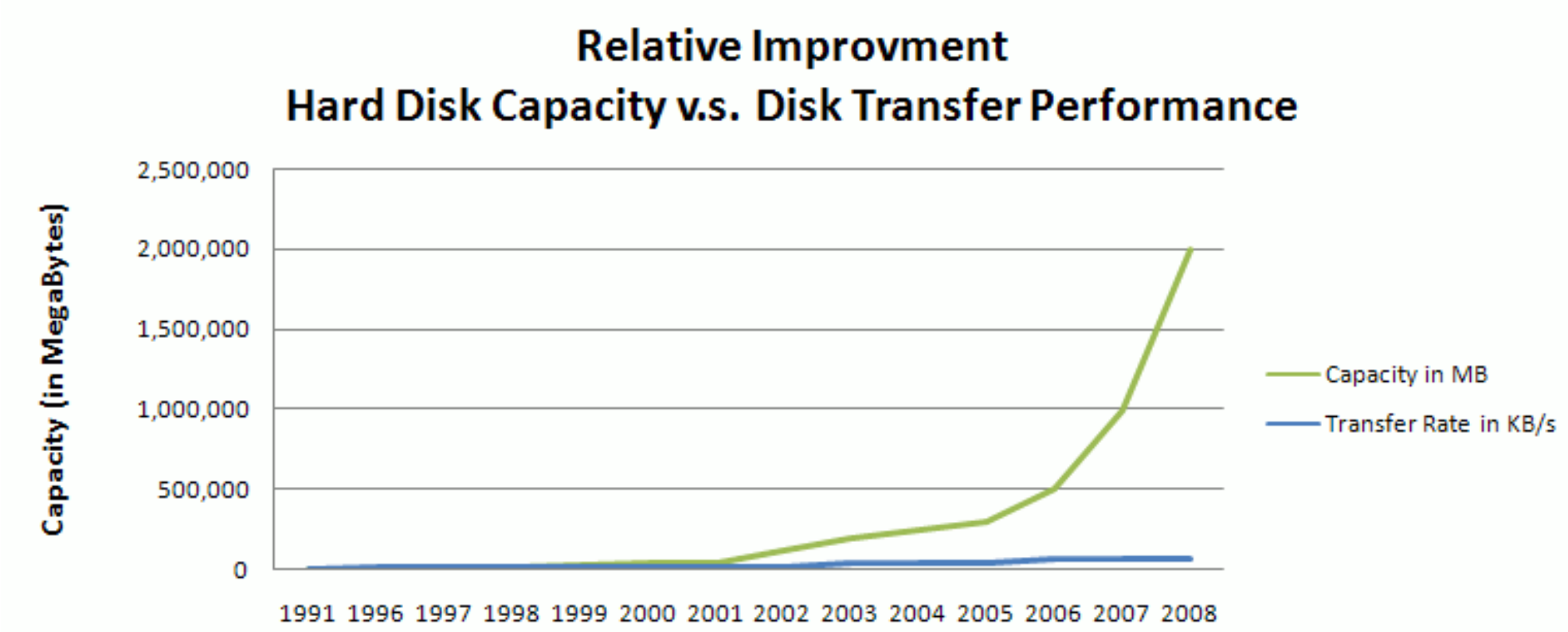
I/O is a bottleneck

Suppose we have crawled 100 Tb worth of data

State-of-the-art 7,200 rpm SATA drive has 3 Gbps I/O

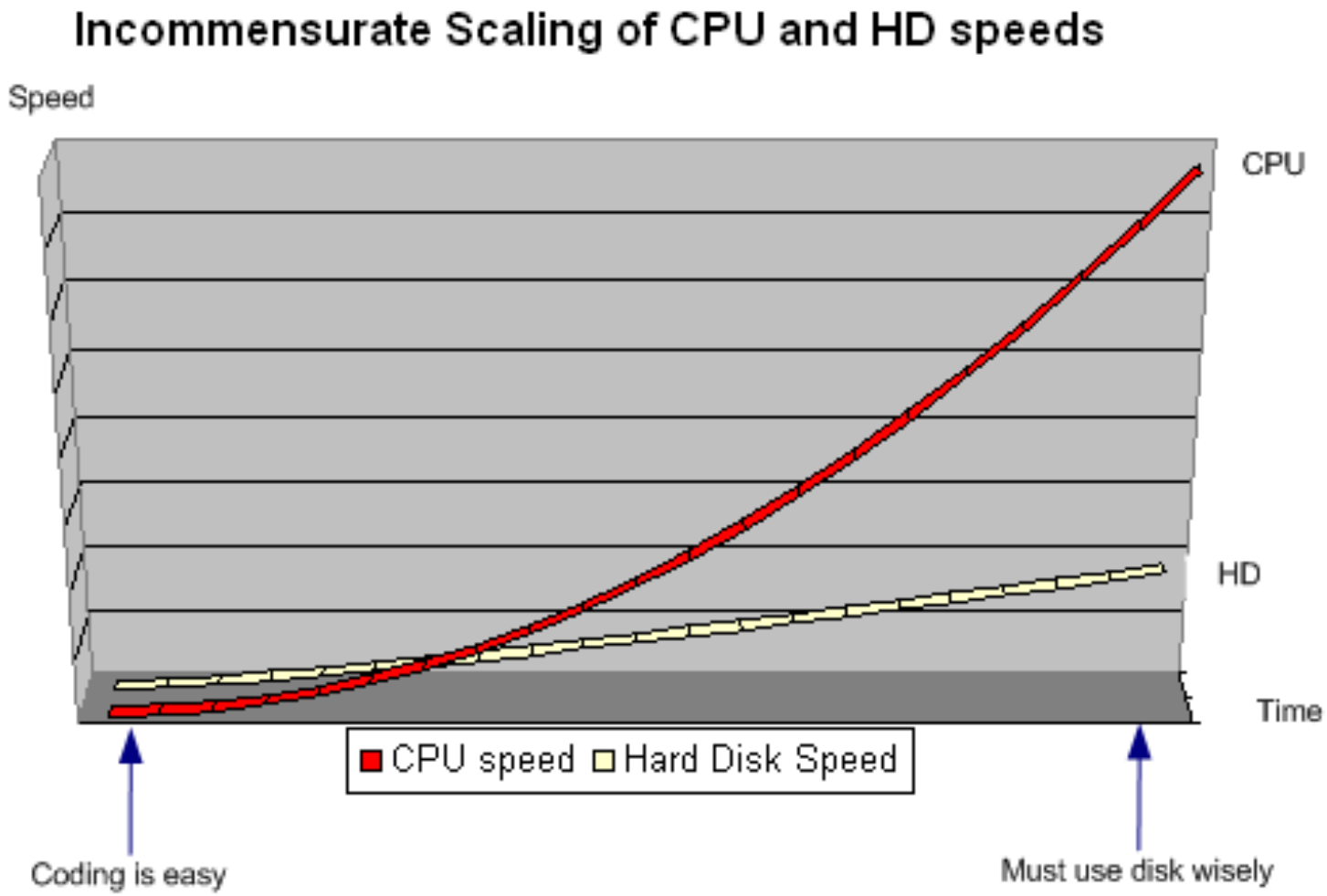
It takes 100 Tb / 3 Gbps = **9.3 hours** to scan through the entire data!

I/O lags far behind



Source: R1Soft, http://wiki.r1soft.com/pages/viewpage.action?pageId=3016608

I/O lags far behind



Source: R1Soft, http://wiki.r1soft.com/pages/viewpage.action?pageId=3016608

Other issues

Building a high-end supercomputer is much, much costly

Storing all data in one place adds the risk of hardware failures

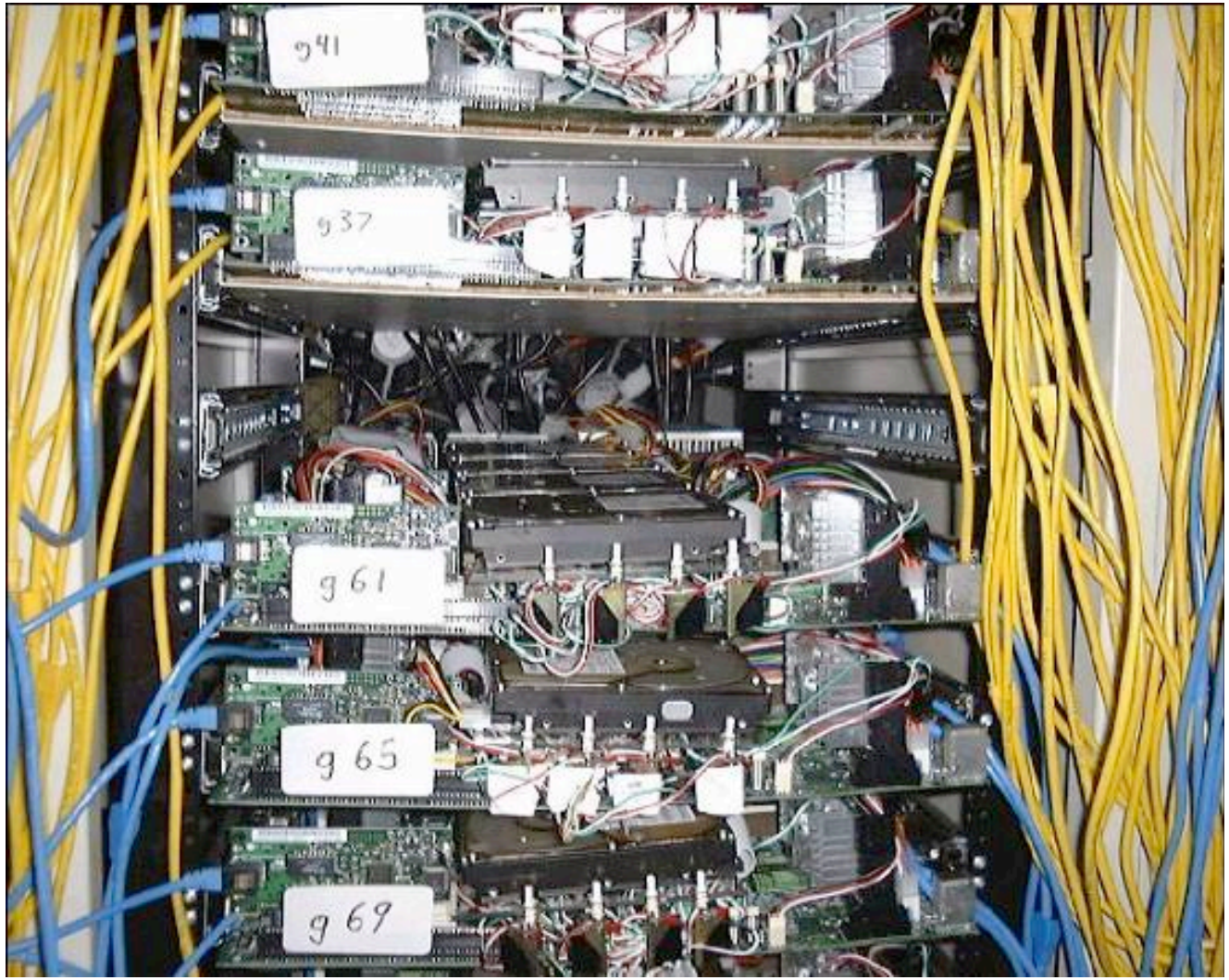
- putting all eggs in one basket!

Is there a way out?

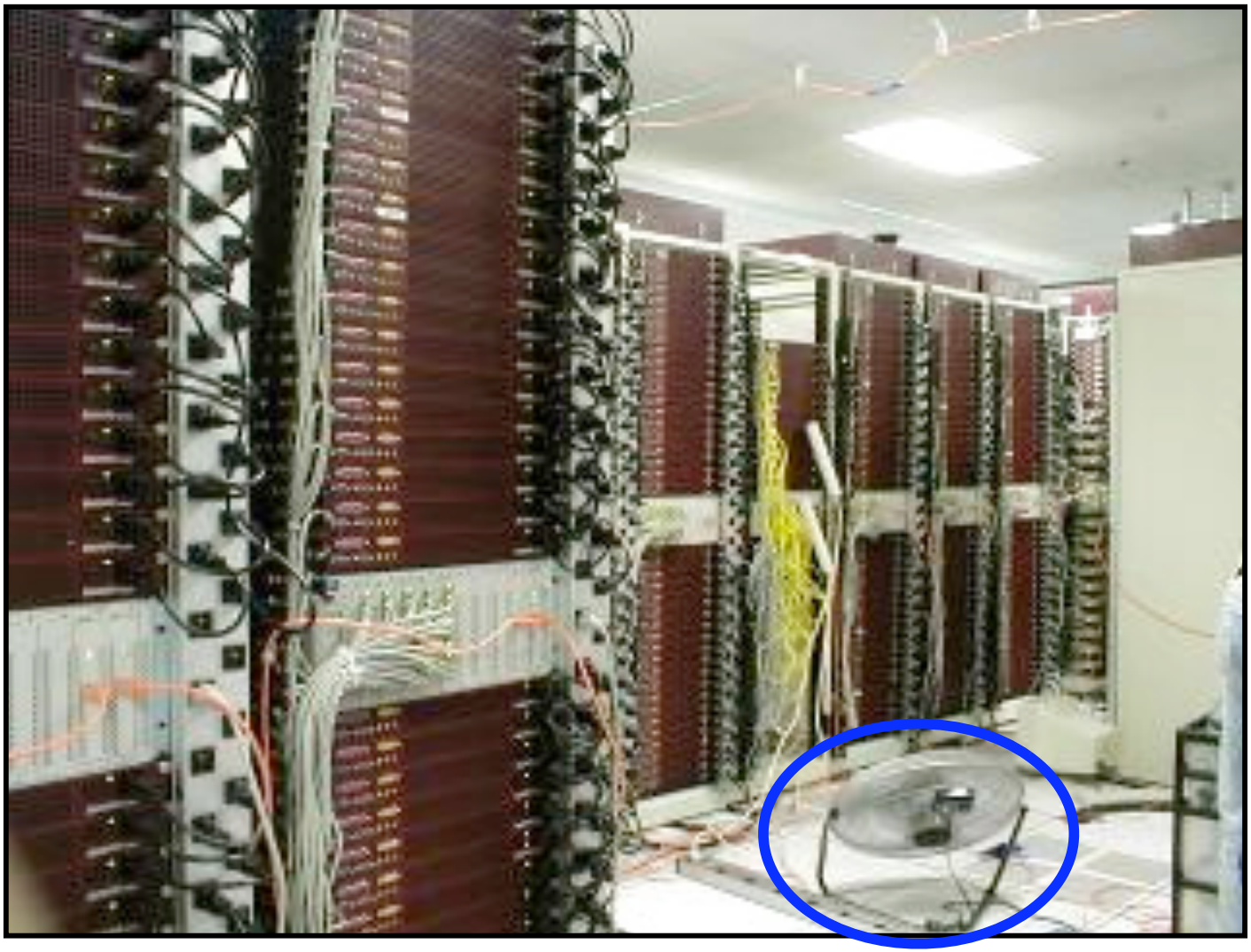
Google's answer: scale "out", not "up"!

“Google” circa 1997 (in Stanford)

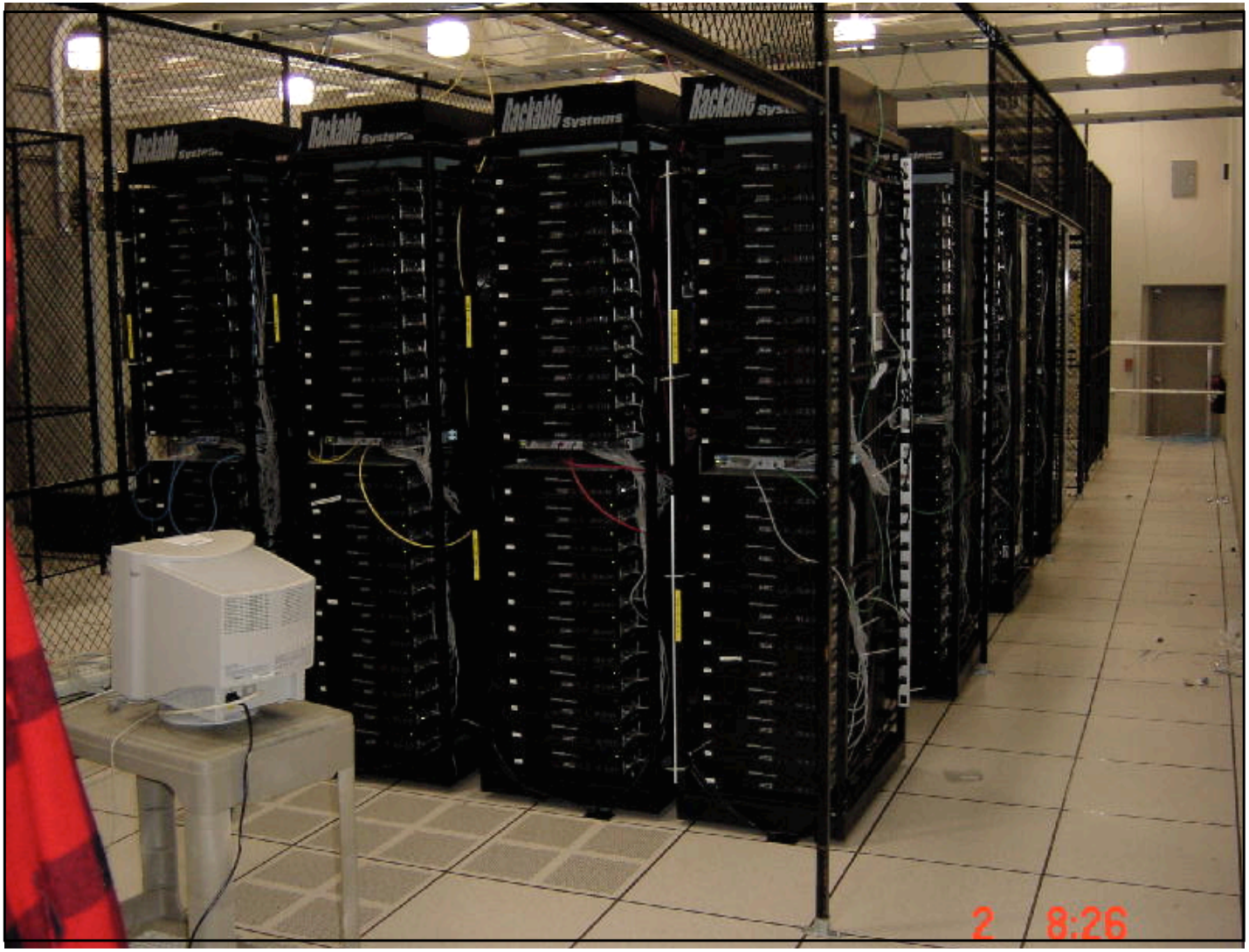
Google (circa 1999)



Google datacenter (circa 2000)



Google datacenter (circa 2001)



Scale “out”, not “up”!

Lots of cheap, commodity PCs, each with disk and CPU

- 100s to 1000s of PCs in cluster in early days

High aggregate storage capacity

- No costly “big disk”

Spread search processing across many machines

- High I/O bandwidth, proportional to the # of machines
- Parallelize data processing

Scale “out”, not “up”!

Suppose we have crawled 100 Tb worth of data and stored in a 1000-node cluster

- State-of-the-art 7,200 rpm SATA drive has 3 Gbps I/O
- 1000 nodes provide 3 Tbps aggregated I/O, 1000x faster than the BIG disk

It takes 100 / 3 = **33 seconds** to scan through the entire data!

A cool idea! But wait…

The joys of real HW in the 1st year

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)

~1 PDU failure (~500-1000 machines powered down, ~6 hours)

~1 rack-move (~500-1000 machines powered down, ~6 hours)

~1 network rewiring (rolling ~5% of machines down over 2-day span)

~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)

~5 racks go wonky (40-80 machines see 50% packet loss)

~8 network maintenance (4 might cause ~30-min connectivity loss)

~1000 individual machine failures, ~thousands of hard drive failures, slow disks, bad memory, misconfigured machines, flaky machines, etc.

The list goes on…

Implications

Stuff breaks

- If you have one server, it may stay up 3 years (1,000 days)
- If you have 10k servers, expect to lose 10 a day

“Ultra-reliable” hardware doesn’t really help

- At large scales, super-fancy reliable hardwares still fails, albeit less frequently
  - software still needs to be fault-tolerant
  - commodity machines w/o fancy h/w give better perf/$

Reliability has to come from the software!

GFS: The Google File System

A highly reliable storage system built atop highly unreliable hardwares

Outline

Target environment

Design decisions

General architecture

File read and write

Fault tolerance

Measurements

Conclusions

Target environment

Thousands of computers

Distributed

- Computers have their own disks, and the file system spans those disks

Failures are the norm

- Disks, networks, processors, power supplies, application software, OS software, human errors

Target environment

Files are huge, but not many

- >100M, usually multi-gigabyte
- a few million files

Write-once, read-many

- Files are mutated by appending
- Once written, files are typically only read
- Large streaming reads and small random reads are typical

Target environment

I/O bandwidth is more important than latency

- Suitable for batch processing and log analytics

It's helpful if the file system provides synchronization for concurrent appends

GFS Design Decisions

File stores as/divided into chunks

Large chunk size: 64 MB

- a file smaller than a single chunk does not occupy a full chunk's worth of storage

Why large chunks?

- minimizes the cost of disk seeks
- pushes up file transfer rate to the disk transfer rate

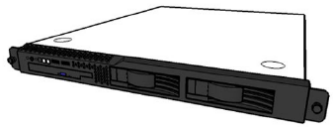
$$T_{File\ transfer} = T_{disk\ transfer} + T_{disk\ seek}$$

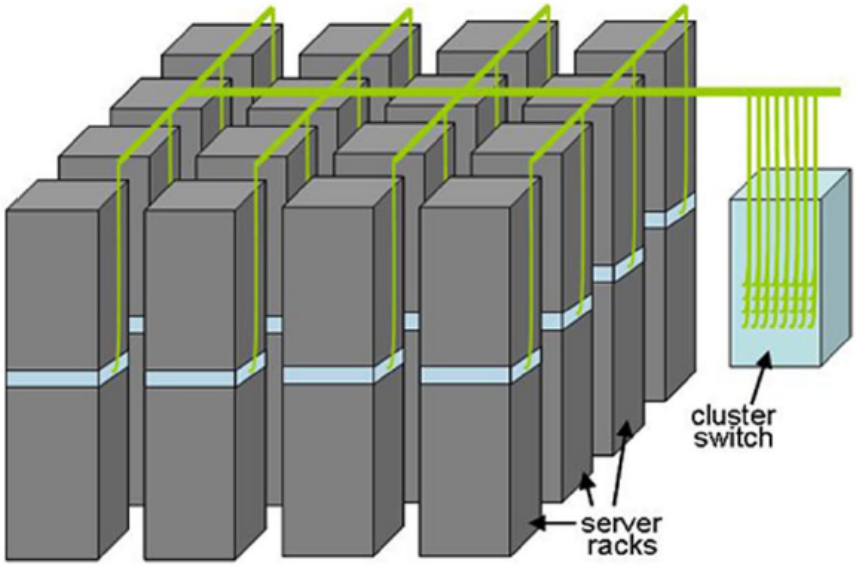
$$\| \quad 0$$

Reliability through replication

3-way replication

- Each chunk replicated across 3+ chunkservers
  - 1 replica in the same rack
  - 2+ in other, different racks





Other design decisions

Single master to coordinate access

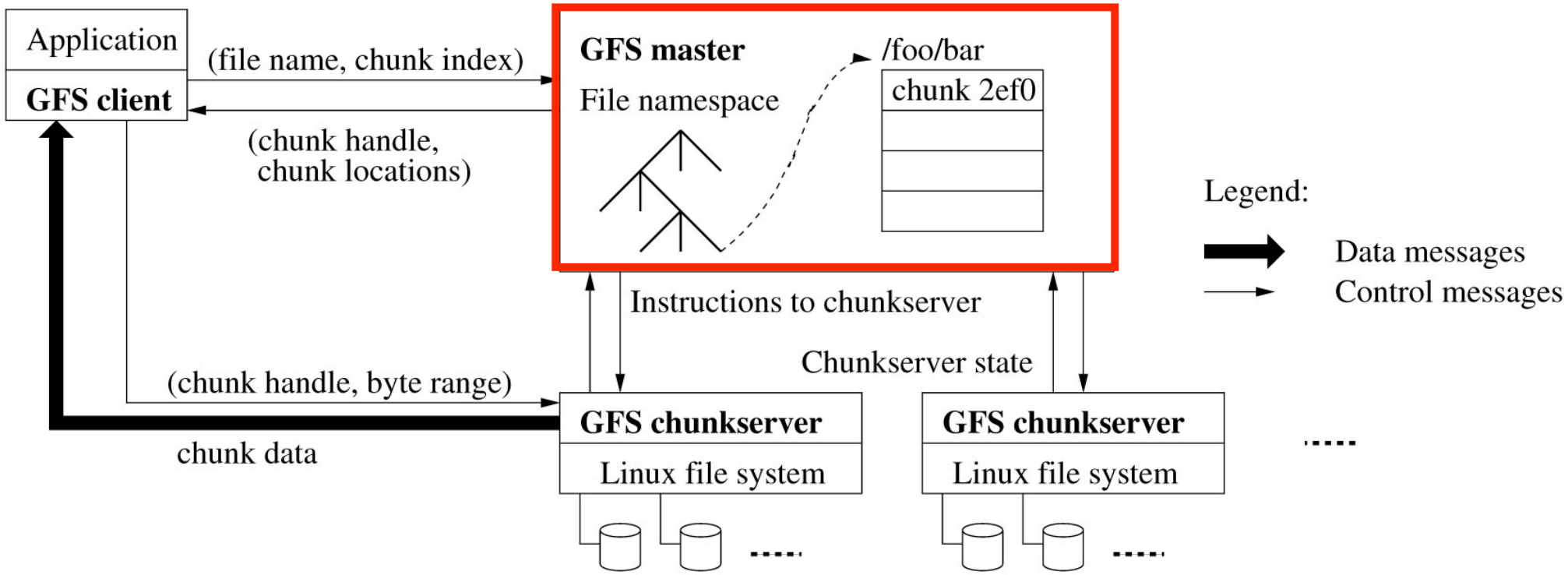
- keeps *metadata*
  - *filename, permissions, chunk index, folder hierarchy, etc.*
- file data stored in chunkservers

Add record append operations

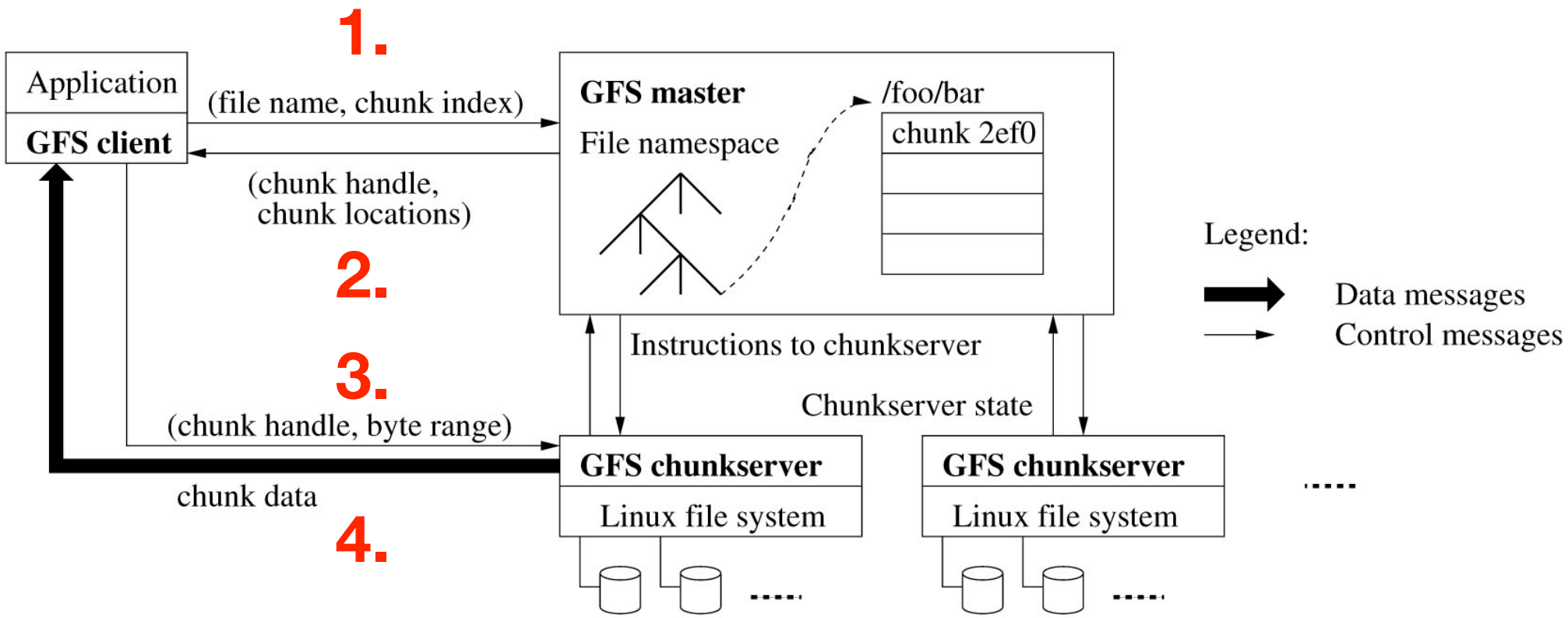
- Support concurrent appends

General Architecture

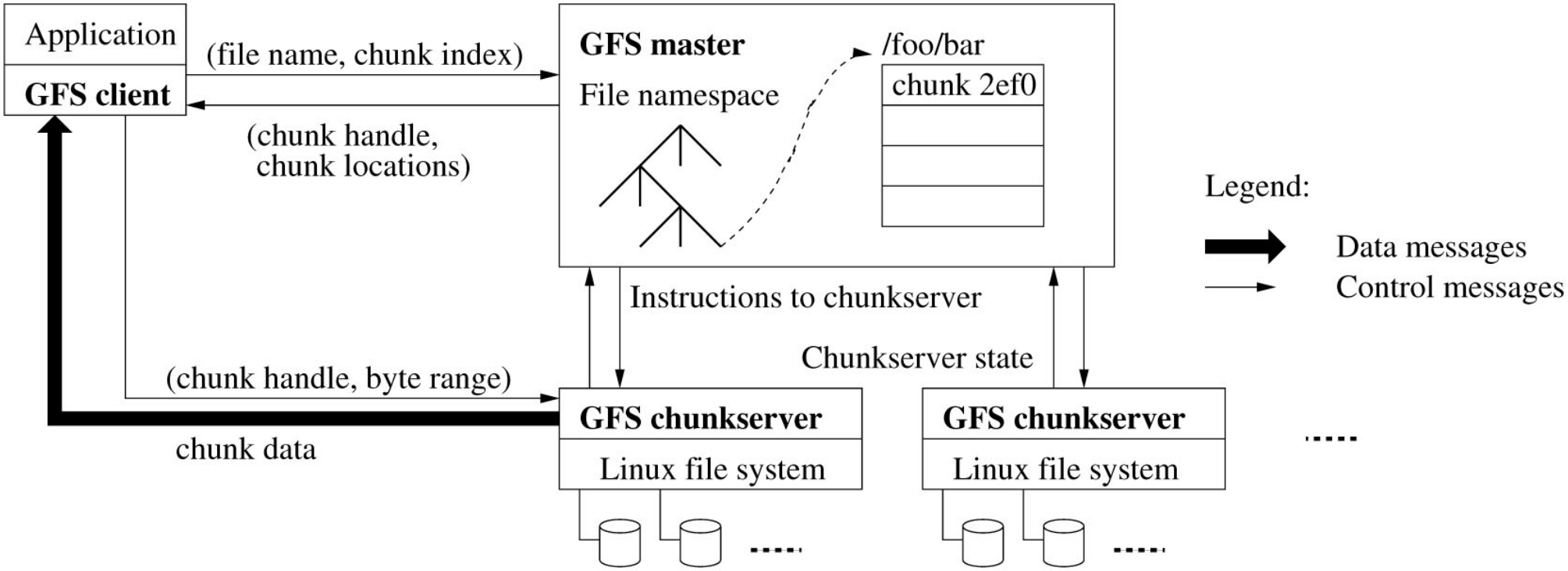
Single master
Multiple chunkservers



Single master
Multiple chunkservers



Anyone can see the potential problem of this design?

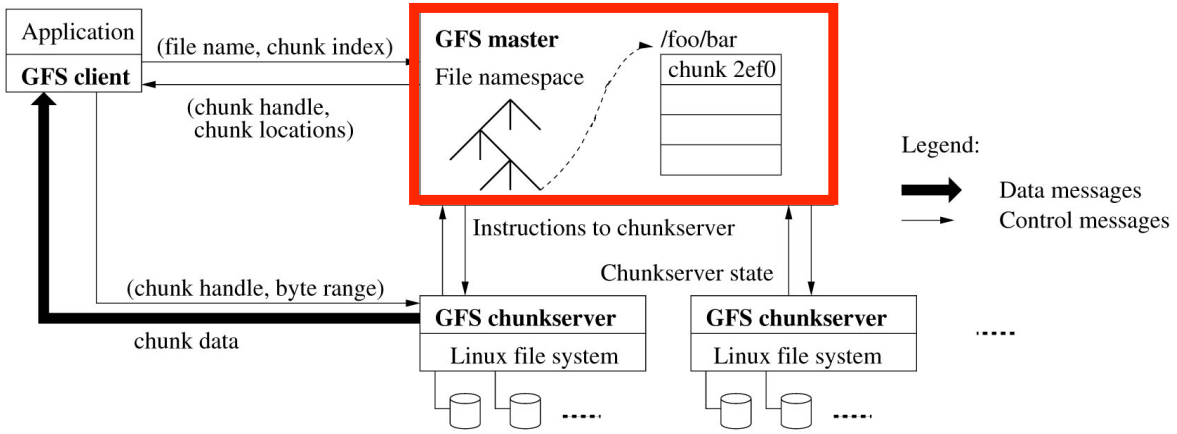


Single master

Problems and concerns

- **Single point of failure:** what if master goes offline?
- **Scalability bottleneck:** what if master is overloaded?

Any solutions?



GFS's answers

Master is the single point of failure

- add a **shadow master**

Master can be overloaded

- Minimize master involvement to address the scalability issue
  - Never move data through it, use only for *metadata*
  - large chunk size: minimizes seeking/indexing time
  - **chunk leases:** master delegates authority to primary replicas in data mutations

Metadata

Three types of metadata, all kept *in memory*

- file and chunk namespaces
- mappings from files to chunks (each chunk has a unique ID)
- locations of each chunk's replicas

Metadata

Three types of metadata, all kept *in memory*

- file and chunk namespaces
- mappings from files to chunks (each chunk has a unique ID)
- locations of each chunk's replicas

First two types are made persistent via an **operation log**

Metadata

Three types of metadata, all kept *in memory*

- file and chunk namespaces
- mappings from files to chunks (each chunk has a unique ID)
- locations of each chunk's replicas

Chunk replica locations learned by polling chunkservers at startup

Chunkserver is final arbiter of what chunks it holds

Master's responsibilities

Metadata storage

Namespace management/locking

Periodic communication with chunkservers

- give instructions, collect state, track cluster health

Chunk creation, re-replication, rebalancing

- spread replicas across racks to reduce correlated failures
- re-replicate data if redundancy falls below a threshold

Master's responsibilities

Garbage collection

- simpler, more reliable than traditional file delete
- master logs the deletion, renames the file to a hidden name
- lazily garbage collects hidden files

Stale replica deletion

- detect "stale" replicas using chunk version numbers

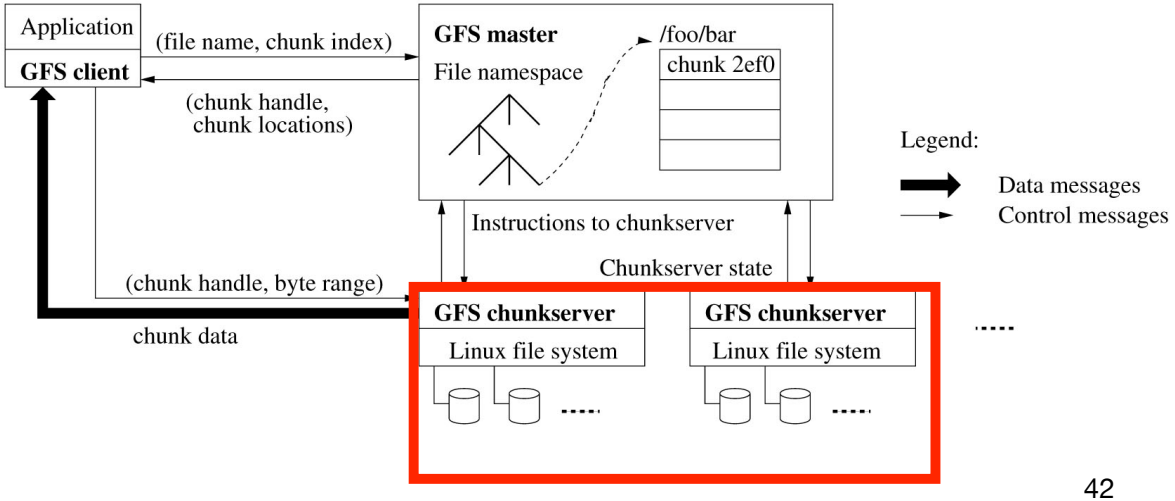
Chunkserver

Stores 64 MB file chunks on local disk, each with **version number** and **checksum**

Read/write requests specify **chunk handle** and **byte range**

Chunks replicated on configurable number of chunkservers (default: 3-way replication)

No file data caching



Client

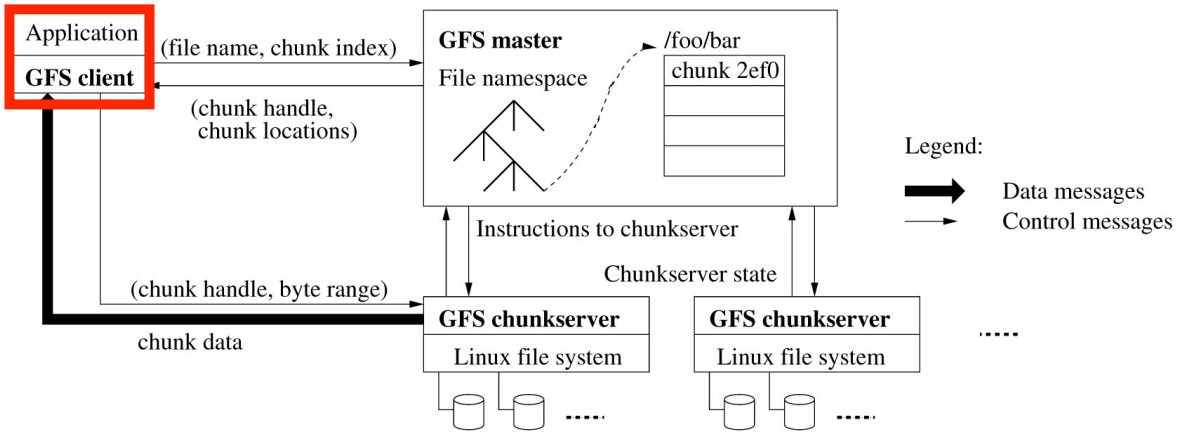
Issues **control** *(metadata)* **requests** to master

- e.g., `ls`

Issues **data requests** directly to chunkservers, e.g., **`cat`**

- minimum master involvement

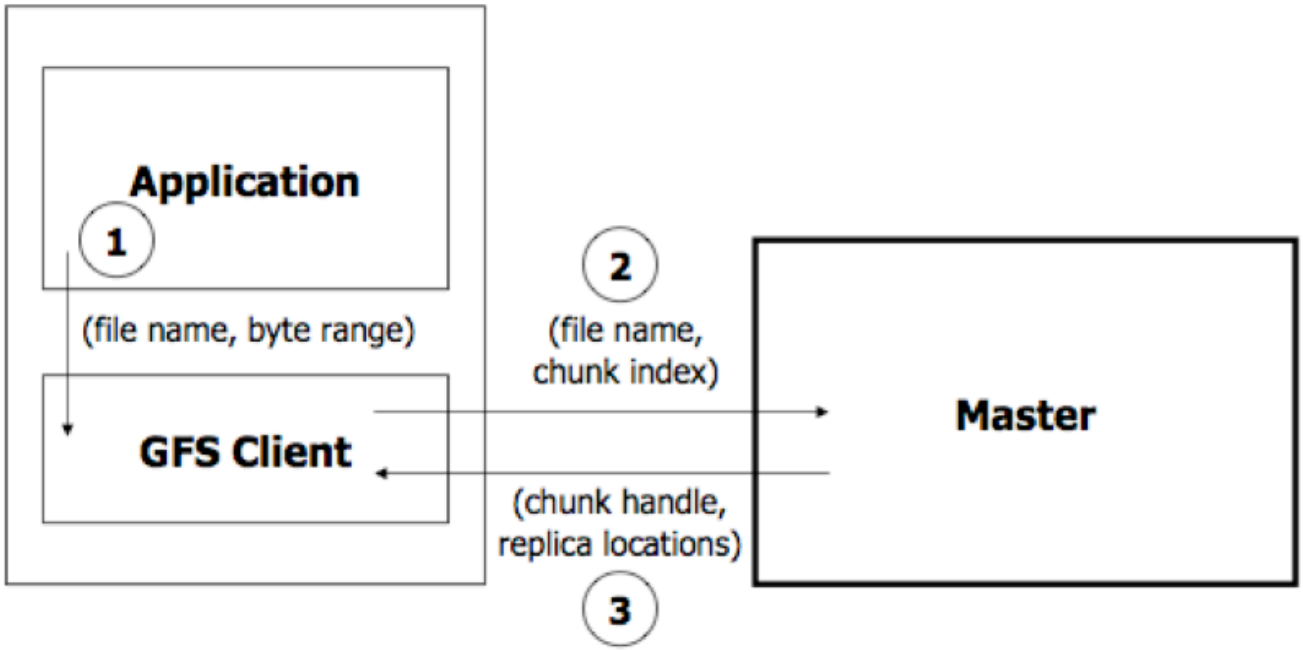
Caches no file data but metadata



File read and write

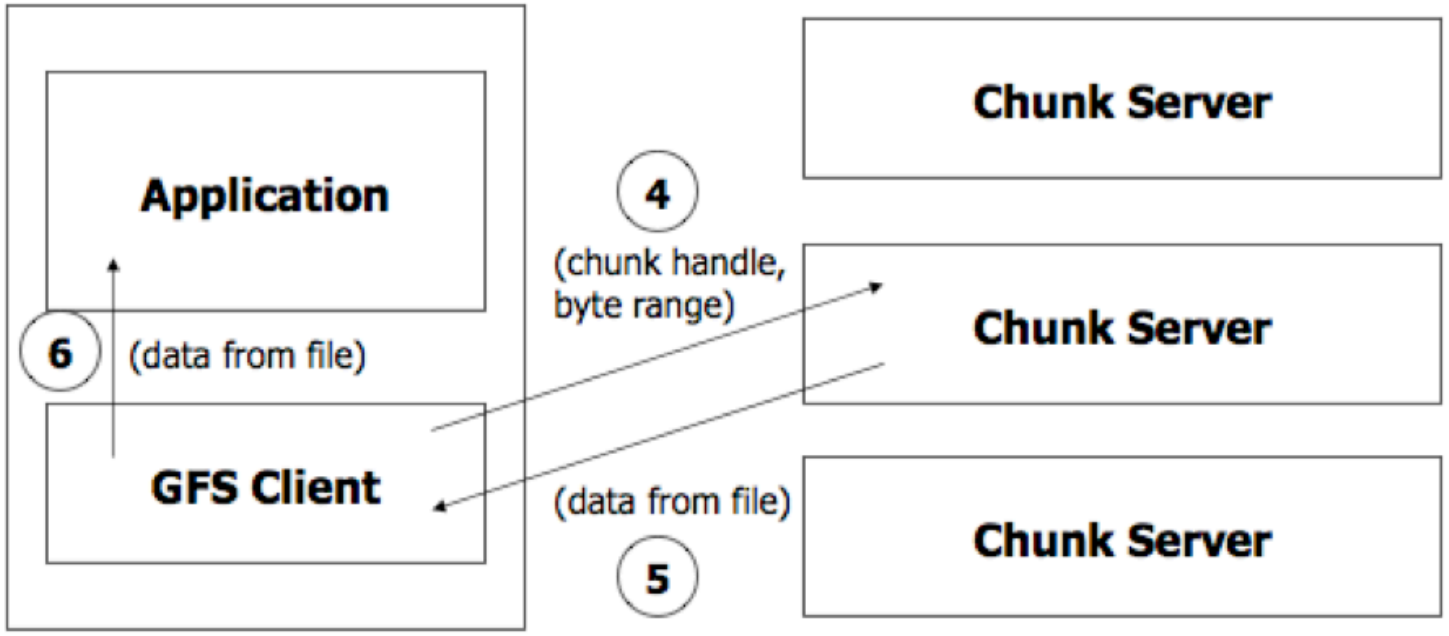
File read

1. Application originates the read request
2. GFS client translates request and sends it to master
3. Master responds with chunk handle and replica locations



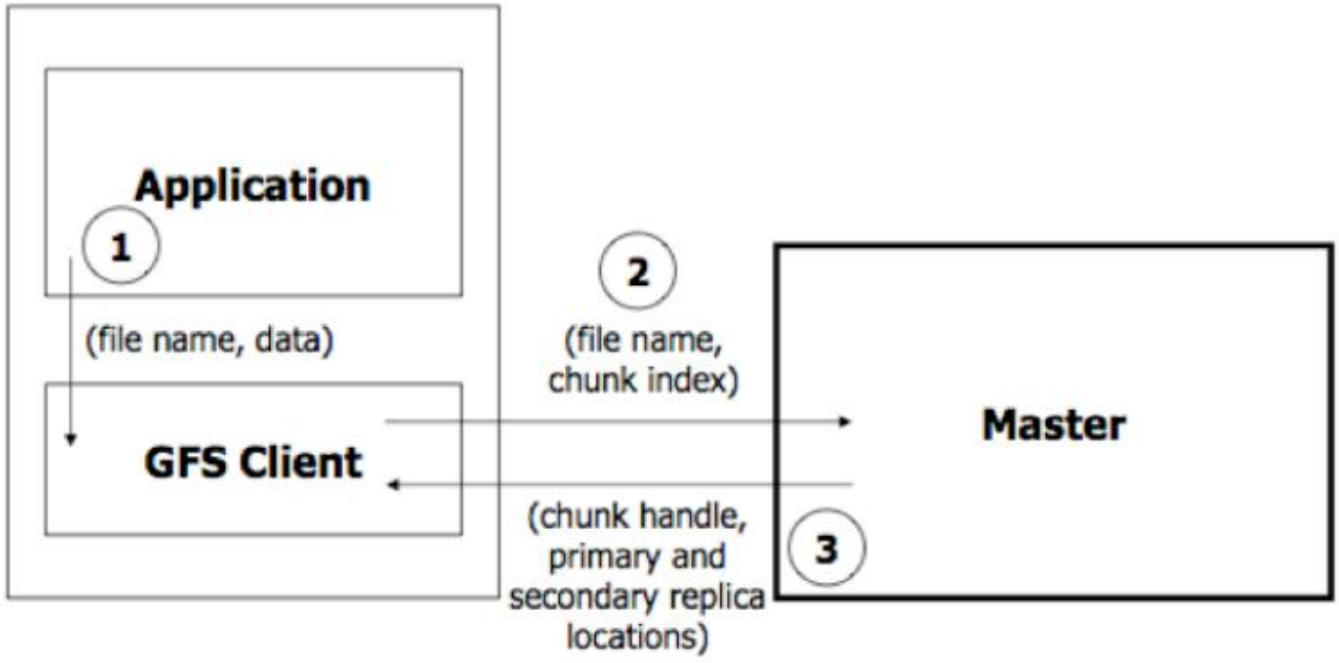
File read

4. Client picks the “closest” location and sends the request
5. Chunkserver sends requested data to the client
6. Client forwards the data to the applicaiton



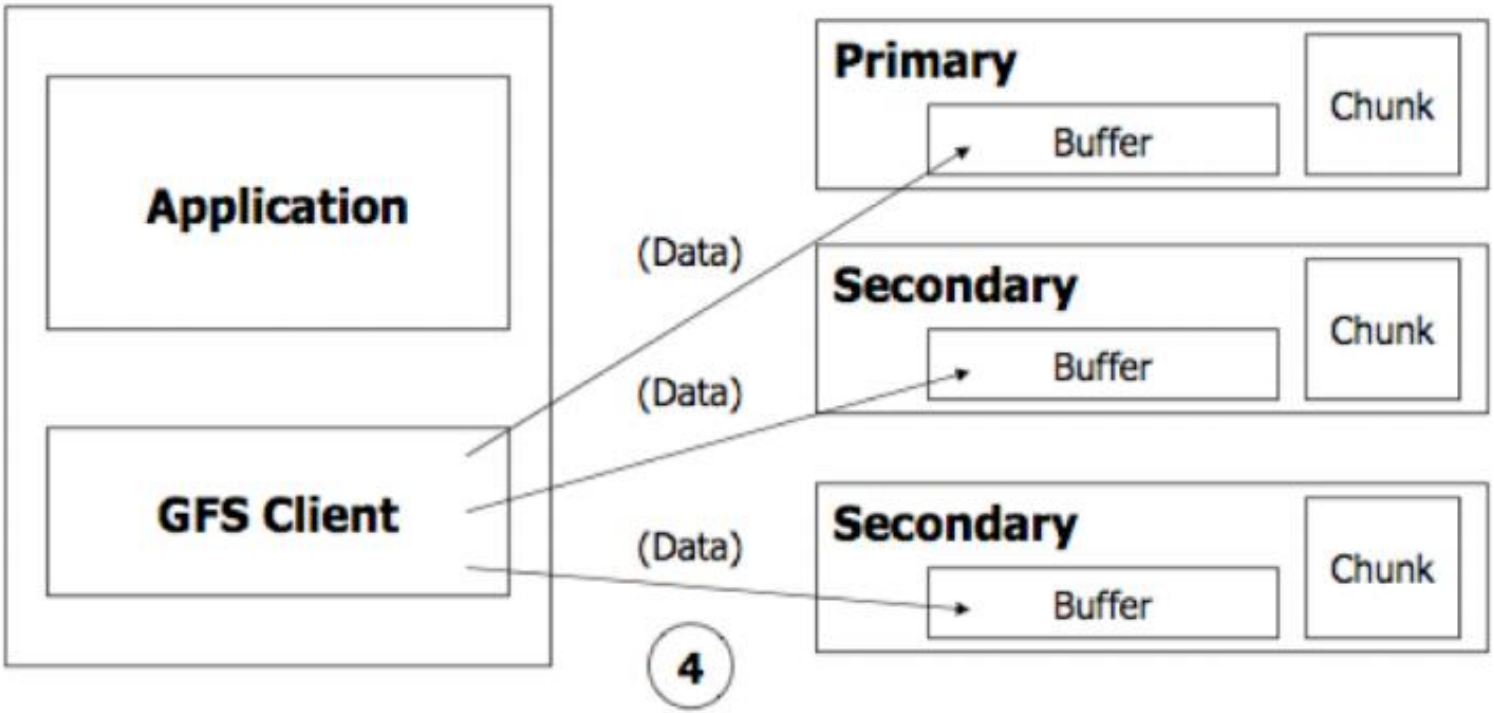
File write

1. Application originates the request
2. GFS client translates request and sends it to master
3. Master responds with chunk handle and replica locations



File write

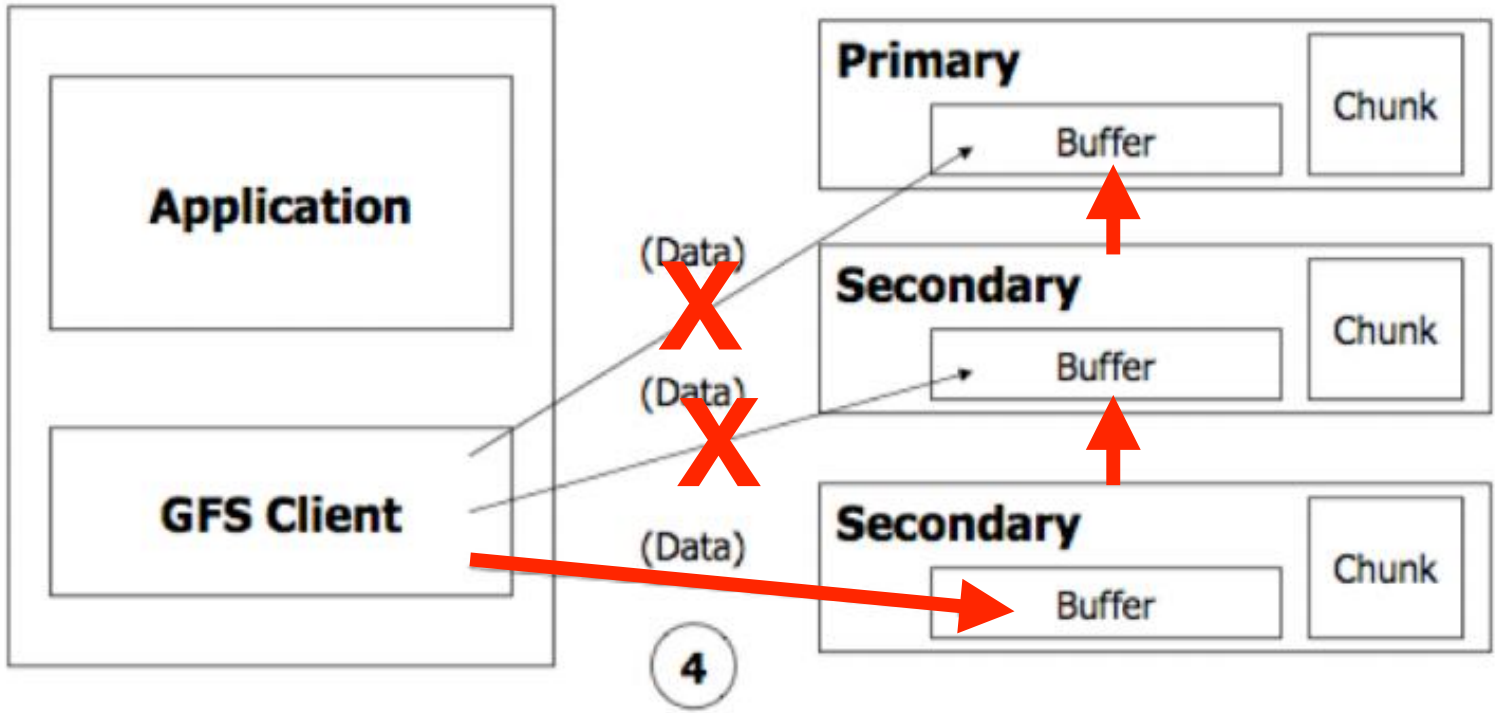
4. Client pushes write data to all locations. Data is stored in chunk server's internal buffers



File write

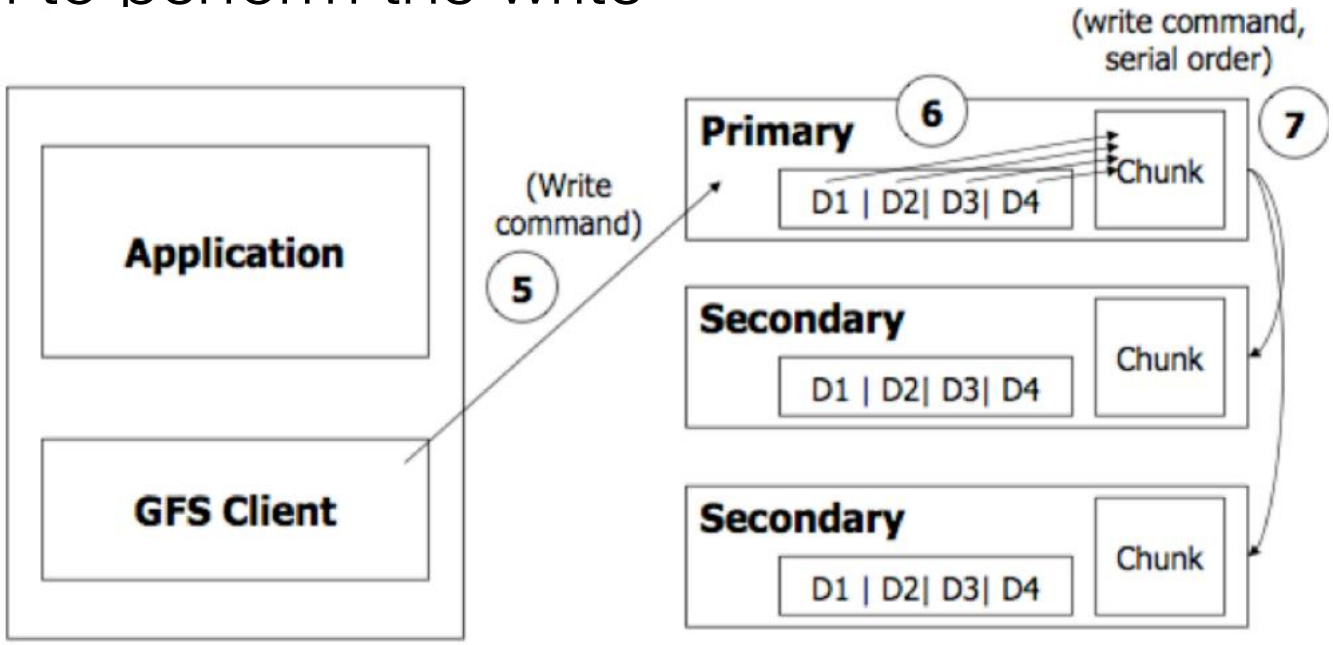
4. Client pushes write data to all locations. Data is stored in chunk server's internal buffers

May form a pipeline



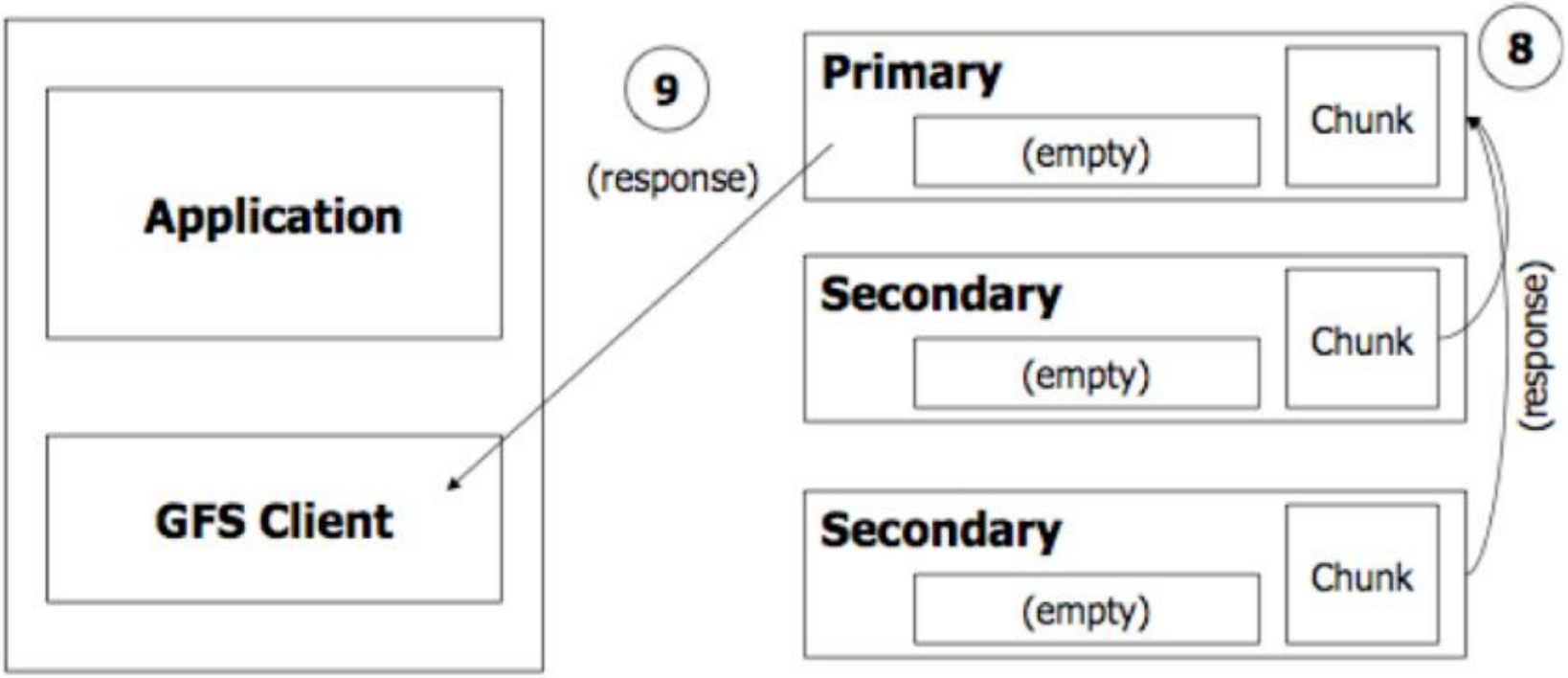
File write

5. Client sends write command to primary
6. **Primary determines serial order for data mutations** in its buffers and writes to the chunk in that order
7. Primary sends the serial order to the secondaries and tells them to perform the write



File write

8. Secondaries respond back to primary
9. Primary responds back to the client



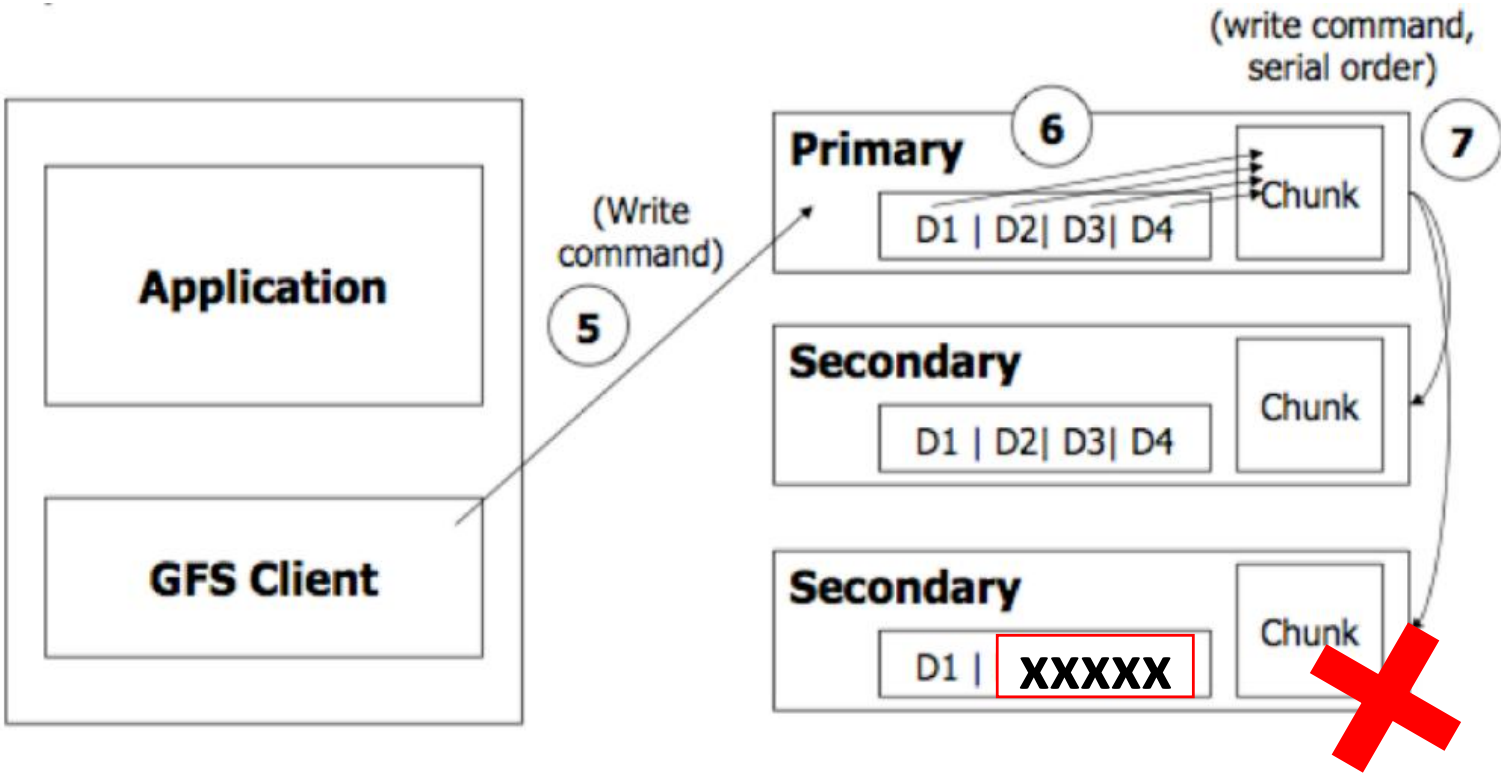
Fault Tolerance

What if write fails?

Rewrite

Chunks are not bitwise identical

- Use checksum to skip inconsistent file regions



What if chunkserver fails?

Master detects a failed “heartbeat” of a chunkserver

Master decrements count of replicas for all chunks on dead chunkserver

Master re-replicates chunks missing replicas elsewhere

Summary of fault tolerance

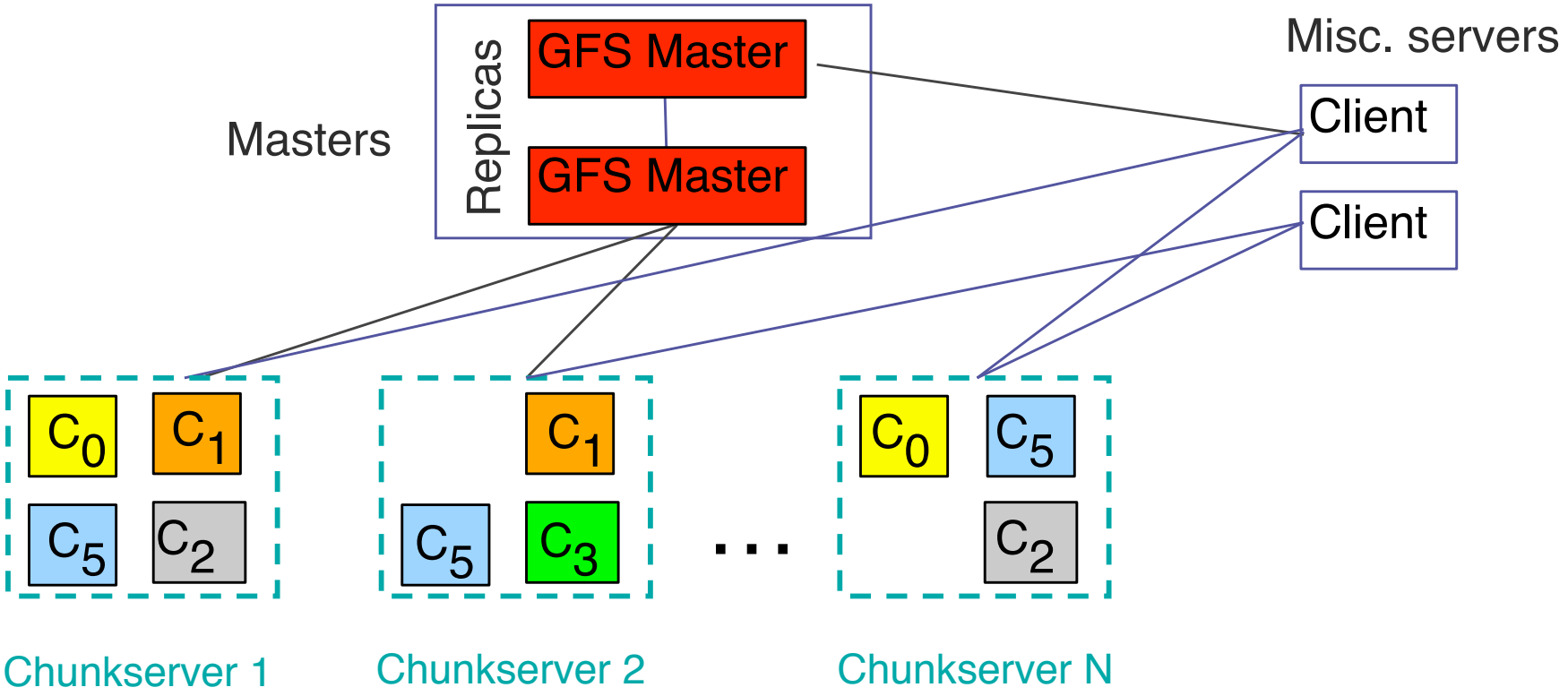
High availability

- Fast recovery
  - master and chunkservers restartable in a few seconds
- Chunk replication: 3 replicas by default
- Shadow master

Data integrity

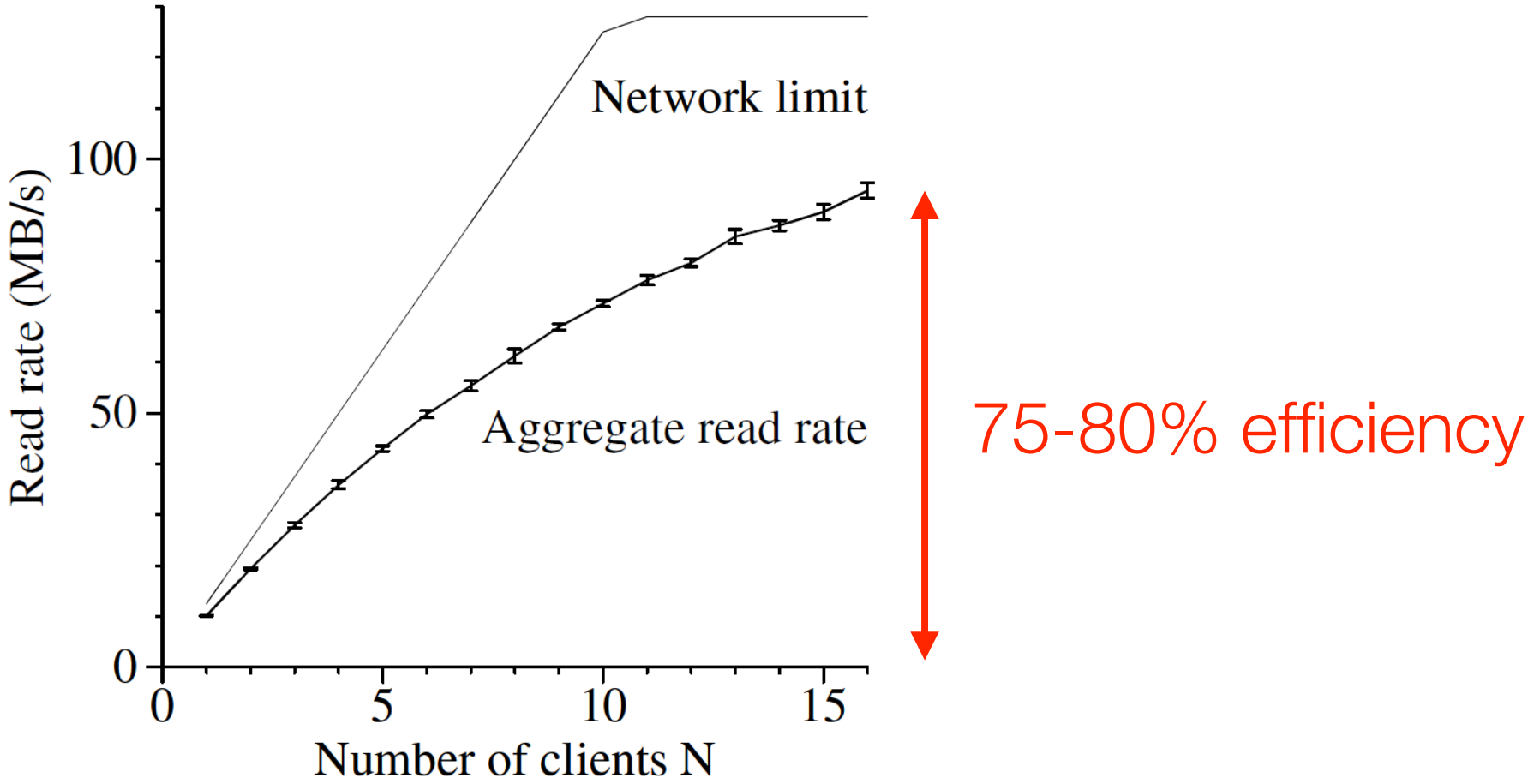
- Checksum every 64KB block in each chunk

Summary of fault tolerance

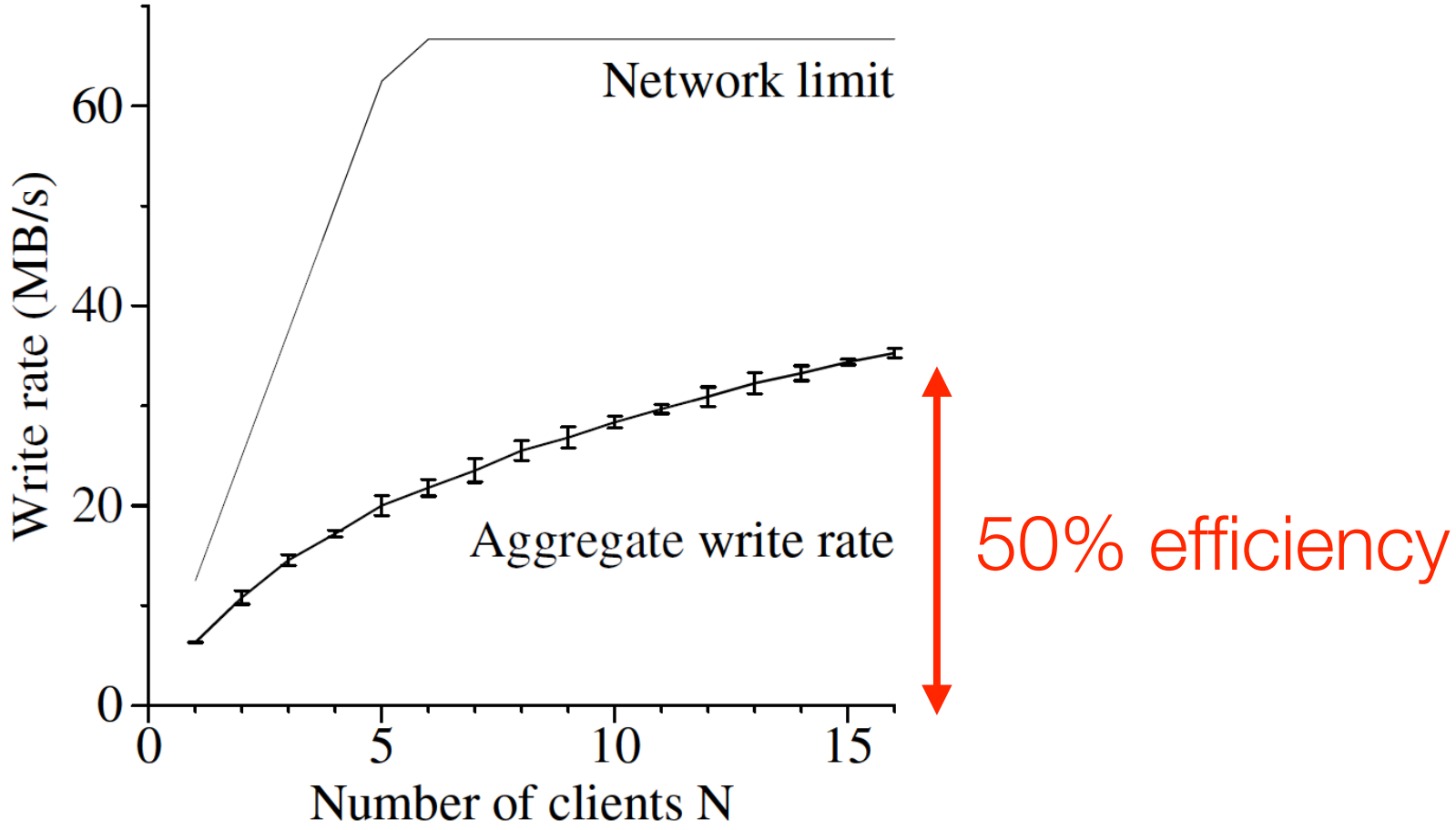


Measurements (2003)

Micro-benchmark: Reads



Micro-benchmark: Writes



Real cluster: Recovery time

Kill 1 chunkserver

- 15,000 chunks (= 600 GB of data) restored in 23.2 mins
- effective replication rate = 440 MB/s

Kill 2 chunkservers

- 16,000 chunks on each (= 660 GB of data)
- reduced 266 chunks to having a single replica
- 266 chunks restored to at least 2x replications within 2 mins

Conclusions

GFS demonstrates how to support large-scale processing workloads on commodity hardware

- tolerates frequent component failures (failures as the norm rather than the exception)
- optimizes for huge files that are mostly appended and then read sequentially
- delivers high aggregate throughput to many concurrent readers and writers

Any limitations?

Limitations

Assumes write-once, read-many workloads

Assumes a modest number of large files

- **Large chunk size:** small files can create hot spots on chunkservers if many clients accessing the same file

Credit

Some slides are adapted from

- Dr. Lazowska and Dr. Chernyak's slides for CSE 490H at UW
- Dr. Alex Moshchuk's slides used in the Google lecture series at UW
- Dr. Jeff Dean's slides used in the Google lecture series at UW
- Dr. Romain Jacotin's slides