Distributed Storage Systems

Juncheng Yang

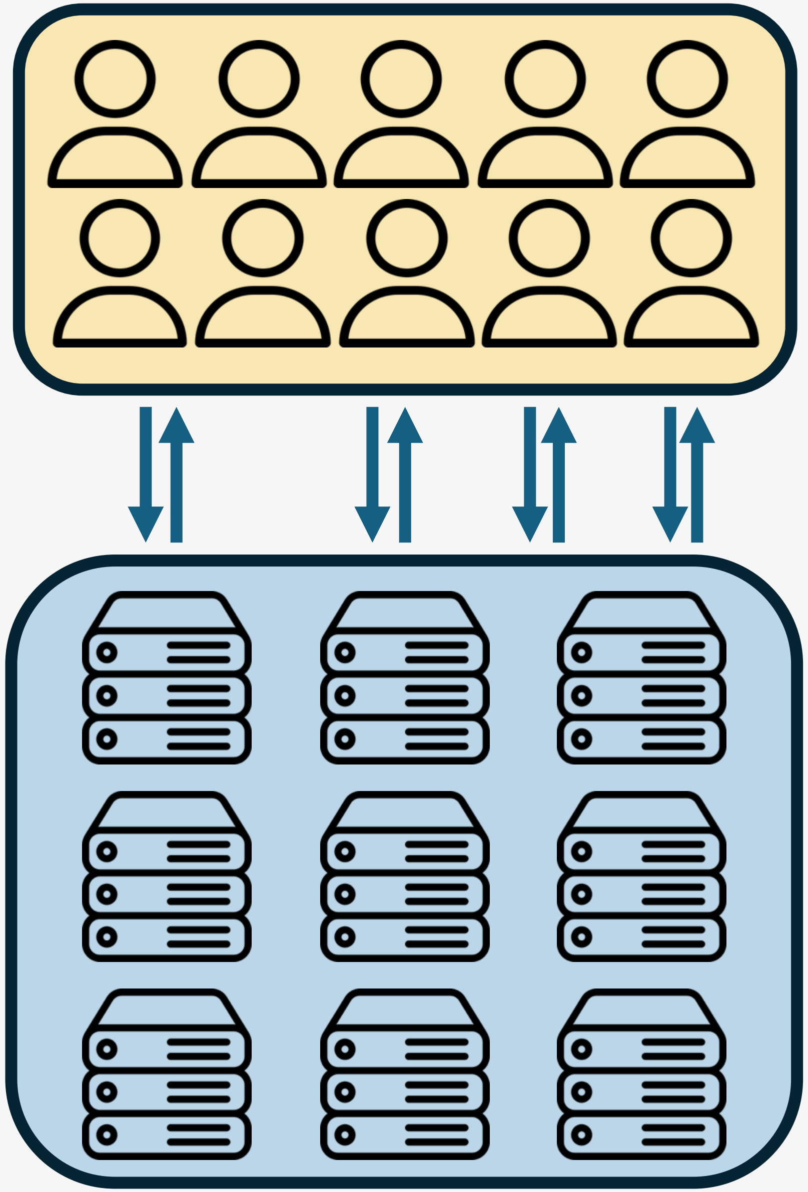
Feb 18

Harvard John A. Paulson
School of Engineering
and Applied Sciences

HARVARD UNIVERSITY
MADSYS
Measurement and Design of Systems Lab

Agenda

- Overview
- Challenges
- Distributed file systems
    - NFS and AFS (1980s)
    - GFS and HDFS (2000s)



Assume some background on distributed systems and will not cover distributed systems content

Three Questions To Ask Yourself After This Class

- Why is designing a distributed storage system challenging?
- What are the differences between NFS and AFS? When should we use NFS or AFS?
- What are the innovations in GFS? How is HDFS different from GFS?

Why Distributed Storage Systems?

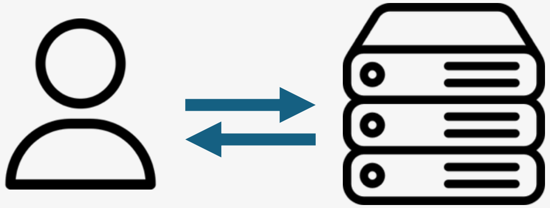
- Capacity
- Performance (bandwidth/IOPS)
- Reliability
- Durability
- Others
  - sharing
  - separation of concerns

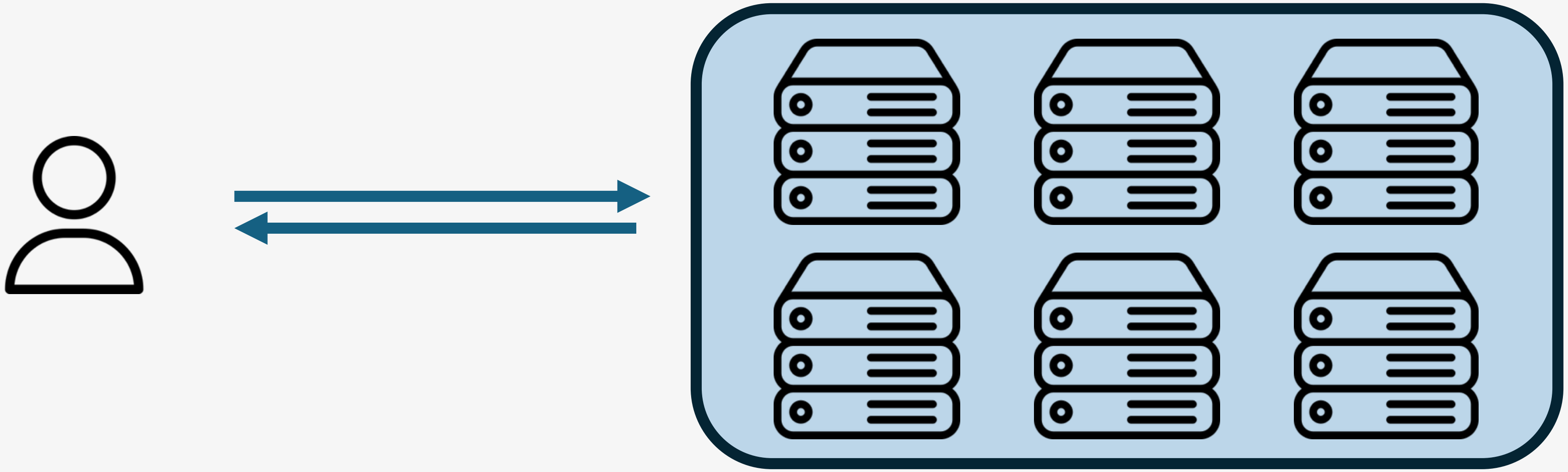
Many Different Terms

- Remote file systems
  - how clients access a file system
- Cluster file systems
  - how multiple servers *coordinate* to provide storage to users
- Parallel file systems
  - how to achieve high aggregated throughput using striping
- Distributed file systems
  - how file systems are built internally, e.g., data, metadata on different nodes
- Distributed *storage* systems
  - more than just file systems, e.g., block, object, key-value

Different Storage System Scenarios

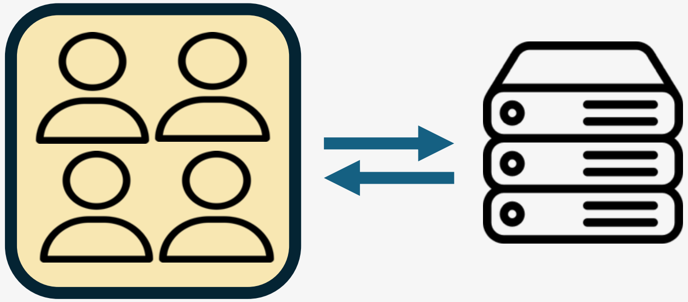
- One user
  - one remote storage server
    - benefit: separation of concern
  - multiple remote storage servers (rare)

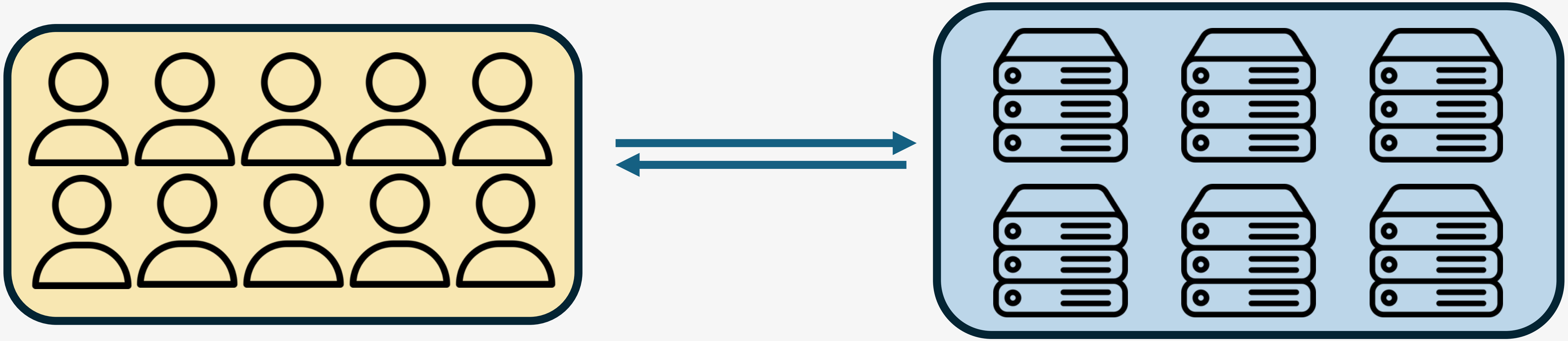




Different Storage System Scenarios

- Many users
  - one remote storage server
    - NFS and AFS
    - NAS (network attached storage)
  - multiple remote storage servers
    - can be within a data center, e.g., GFS
    - can be over the Internet, e.g., Dropbox





Designing a Distributed Storage System Is Challenging

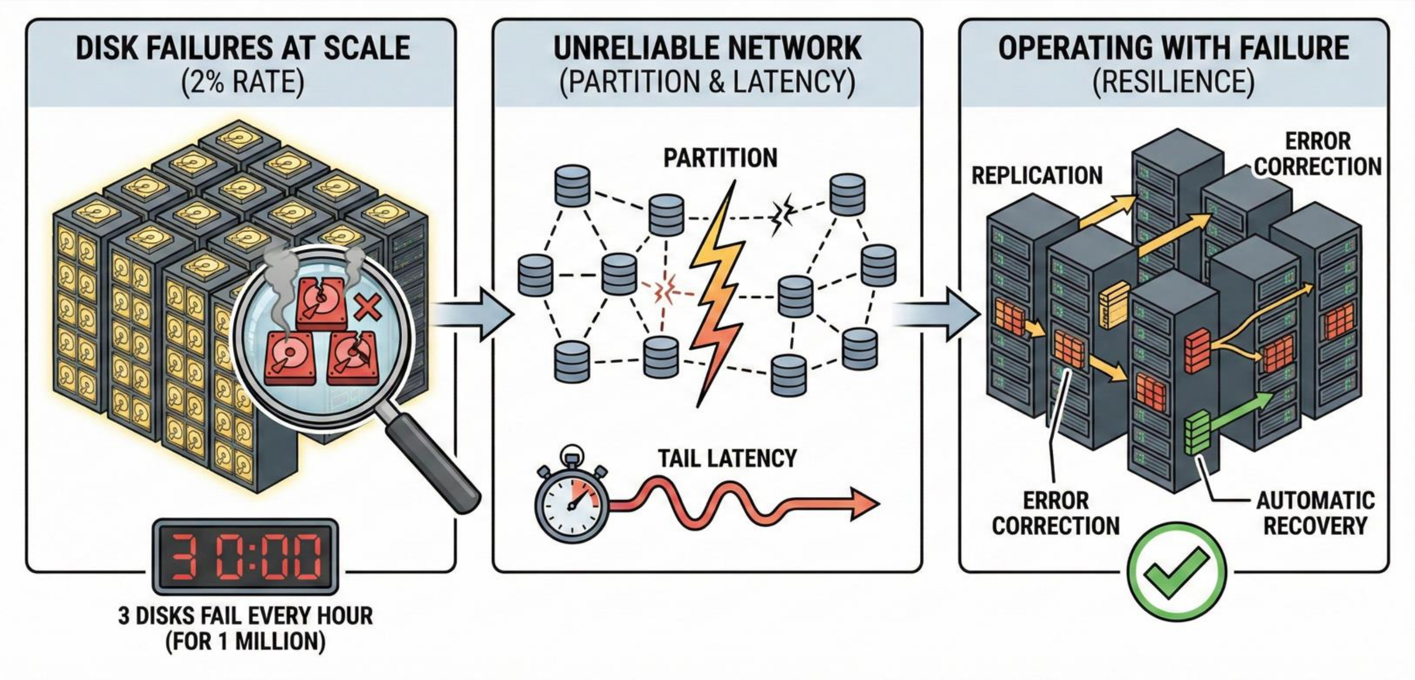
- Performance
  - multiple serialized disk accesses
    - mean per-access latency increases from 10s μs to 100s μs – 10s ms
    - if one access is slow, end-to-end process is slow (tail latency)
    - lock contention
  - load imbalance

more disk accesses for opening files in a deeper directory

operation	inode			data			
	/	cs2640	s1.pdf	/	cs2640	s1.pdf block1	s1.pdf block2
open(/cs2640/s1.pdf)	**read**	**read**	**read**	**read**	**read**		
read()			**r+w**			**read**	
read()			**r+w**				**read**

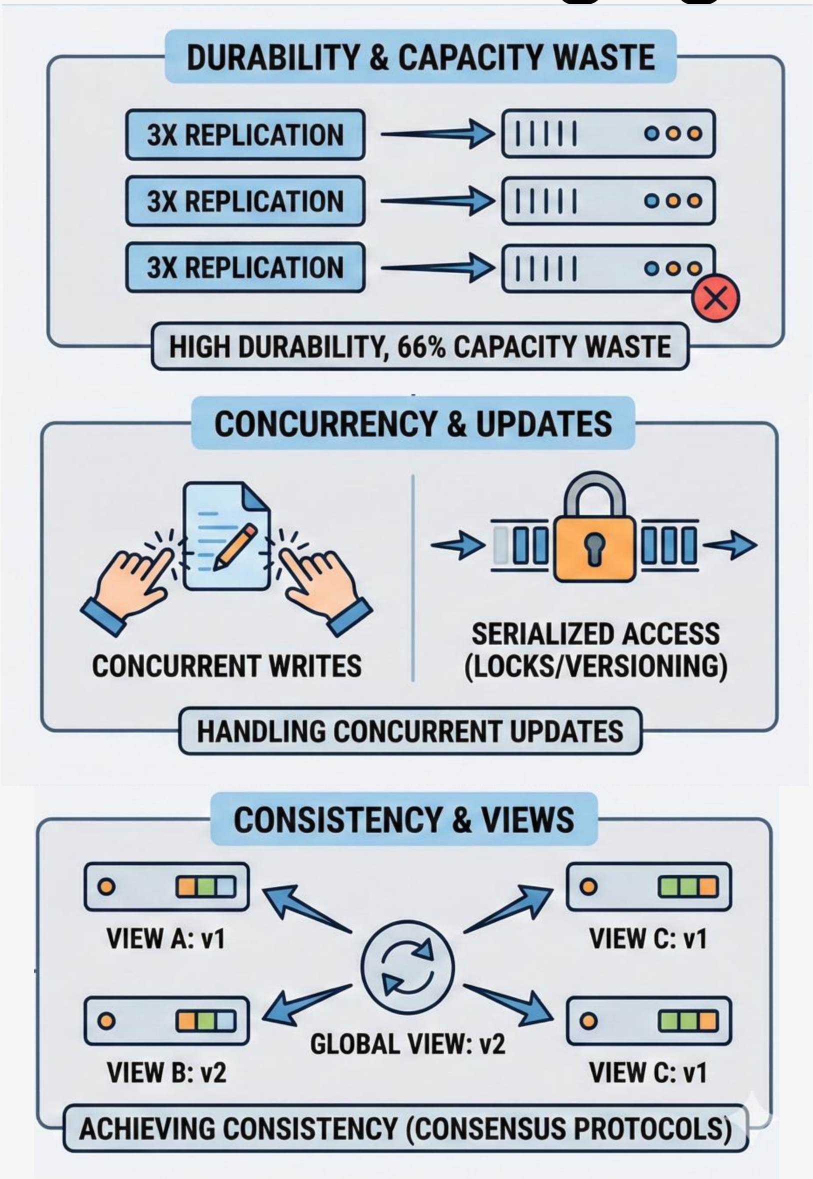
Designing a Distributed Storage System Is Challenging

- Availability and reliability
  - disk failures are common at scale
    - 2% failure rate for 1 millions disk: 3 disks fail every hour
  - network is unreliable
    - partition, and tail latency
  - how to operate with failure

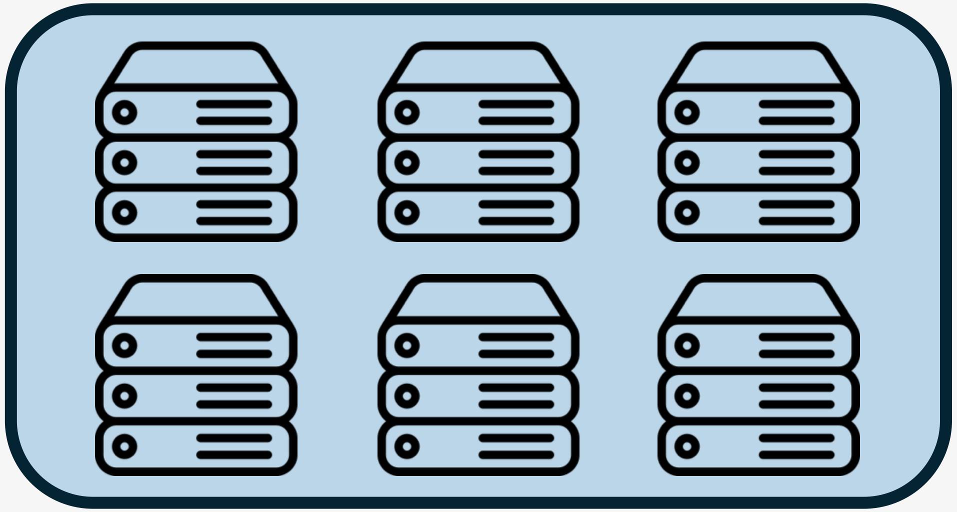


Designing a Distributed Storage System Is Challenging

- Durability
  - how to make sure data is not lost
    - replication improves durability but waste spaces
- Concurrency
  - how to handle concurrent writes
- Consistency
  - how to make different views consistent
- Others
  - authentication, authorization, permission



Designing a Distributed Storage System Is Challenging



- performance
- availability
- durability
- consistency

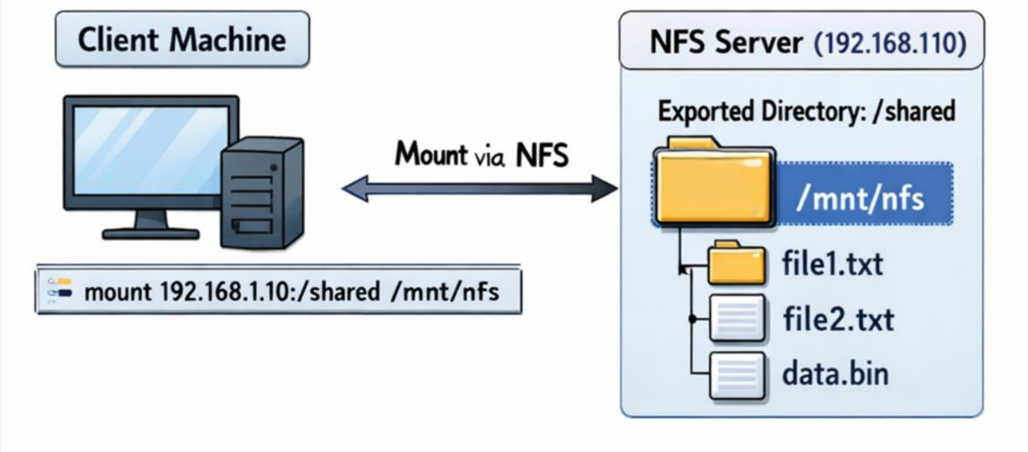
The design of different distributed storage systems are centered around solving these challenges

NFS and AFS

- Network File System (v3)
- Andrew File System

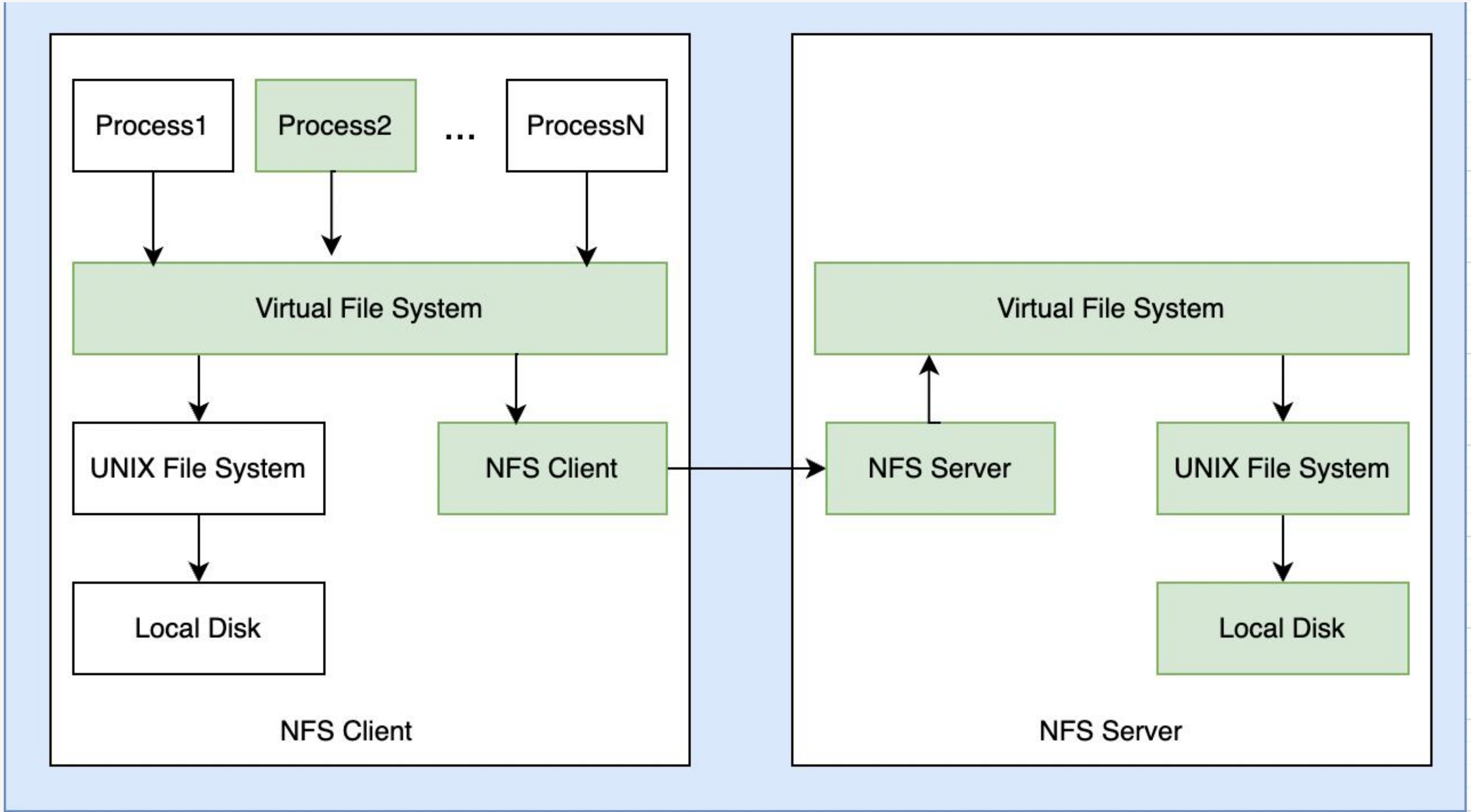
Key question: How do we make remote storage look like a local file system?

NFS Designs



- Developed by Sun Microsystems in 1984
- Philosolphy: transparency (as close to local file system as possible)
  - server exports directories
    - `/nfs/ 10.0.0.0/24(rw,sync)` in /etc/exports
  - clients mount them and access them like local paths
    - `mount -t nfs server1:/nfs /mnt/`
    - requests sent to remote servers when accessing /mnt/
  - use RPC to read and write
  - most operations use file handle (similar to file descriptor)
    - an opaque integer (volume ID, Inode number, generation)
  - VFS was introduced with NFS to achieve transparency

NFS Architecture



NFS Designs

- Stateless (server)
  - server does not keep state: no idea what clients are doing
  - a chatty protocol (client frequently revalidates with server)
    - example: path traversal
  - enable fast crash recovery
    - client still works after NFS server reboots
    - idempotent operations: can be safely retried
      - open, read, write…

NFS Designs

- Caching and write buffer
  - client cache metadata and data blocks with timeout
  - => may see brief inconsistency
- "Close-to-open" consistency model
  - flush dirty blocks upon `close()` note: dirty blocks are also flushed periodically
  - check file modification time upon `open()` using `getattr()`
  - consistency: revalidate during open
  - concurrent updates: last writer wins

AFS

- Designed by CMU and IBM in mid-1980s
- Design philosophy: ***scale*** to many clients ***securely***
- Designs
  - aggressive caching, e.g., whole file caching
    - reduce server load to improve scalability
  - callbacks (inform clients file changes)
  - stateful server (keep track of who has cached what)
  - global namespace
  - stronger security using Kerberos (tied to users)

AFS Global Namespace

- All files live under the global namespace /afs/…
- Cell: an AFS administrative domain, e.g., /afs/***harvard.edu***
- Absolute uniformity
  - a file is always at the same path no matter what machine you use, how you mount
- Cross-organization federation
  - if you want to collaborate with MIT
  - NFS: ask IT to open a firewall, setup a VPN and export an NFS share
  - AFS: cd /afs/mit.edu/…

AFS Volume Management

- A Volume is a logical container of files
  - essentially a directory tree that can be treated as a single object
  - granularity: typically one user = one volume
- VLDB (Volume Location Database): location independence
  - a replicated database that tracks where every volume currently lives
  - decouple path from physical storage, allow better management
  - e.g., /afs/harvard.edu/user/juncheng => server 2
- Replication (read-only volume)
  - effectively a cache for faster access (load, distance) and better availability

AFS Whole File Caching

- Open
  - check disk cache, open if hit
  - miss: client fetches the *entire file* and stores on local disk
- Read/Write
  - happen locally—zero network traffic while you are editing the file
- Close
  - sends the updated file back to server
- Callbacks
  - server uses callback to avoid staleness
  - if a file is changed, tell all clients who have cached the file
    - but client only fetches when needed

AFS Security

- NFS trust machines
  - once connected, UID 1000 on my laptop = UID 1000 on the server
- AFS uses Kerberos
  - client authenticates with password to get a token
  - client passes token to AFS server for every operation
  - enable clients on any machine
- ACLs (Access Control Lists)
  - permissions are per-directory, not per-file
    - everyone has permission to a directory can read all files under it
  - AFS has 7 specific rights with fine-grained control vs. 3 in Unix (rwx)
    - read, lookup, insert, delete, write, lock, admin

NFS vs. AFS

System design is often making the right trade-offs

- Goal
  - NFS: 10-100 machines on a trusted LAN
  - AFS: 1000s machines on campus
- Design philosophy
  - NFS: as close to local file system as possible
  - AFS: scale to many clients (via aggressive caching)
- Namespace
  - NFS: per server
  - AFS: global namespace, separate name from location (a location database tells which servers host that volume)

NFS vs. AFS

- Fault tolerance
  - NFS: rely on underlying storage
  - AFS: volume has read/write or read-only instances
- Security
  - NFS: host-based UID/GID
  - AFS: more complex built around tokens
- Consistency model
  - NFS: close-to-open
  - AFS: callback-based (cache whole file and use callback to invalidate)

NFS vs. AFS

- Q: How do AFS and NFS compare on performance?
- First access: NFS is faster
  - AFS needs to fetch full file
- Sequential read: AFS is better
- WAN (high latency): AFS is better
  - NFS need to contact server for each lookup
  - AFS cache the data
- Scalability: AFS wins
  - AFS server does nothing after sending files to clients
  - NFS server responds to every read/write/open request

NFS and AFS Performance

Workload	Winner	Why?
Home Directories		
Compiling Code (make)		
Databases (Random I/O)		
Video Streaming		
Shared Binaries (/usr/bin)		

Challenges with NFS and AFS

- NFS:
  - availability: single point failure
  - chatty: slow and excessive traffic
  - durability
- AFS:
  - does not work with large data
  - state management is complex
- Both
  - concurrent writes
  - consistency
  - each file is stored on one server: limited bandwidth
  - load imbalance: some files are hotter
  - capacity growth: manual management

Entering Modern Distributed File Systems

Problem	Modern Solution (e.g., GFS, Ceph, Lustre)
Bandwidth	Striping: break files into chunks and store on many servers, also addressed load imbalance problem
Durability	Replication or erasure coding
Availability	Decoupling: separate metadata from data to allow independent scaling and better fault tolerance
Consistency	Tunable Consistency: balance between performance and concurrency

GFS and HDFS

- Google File System (~2003)
- Hadoop Distributed File System

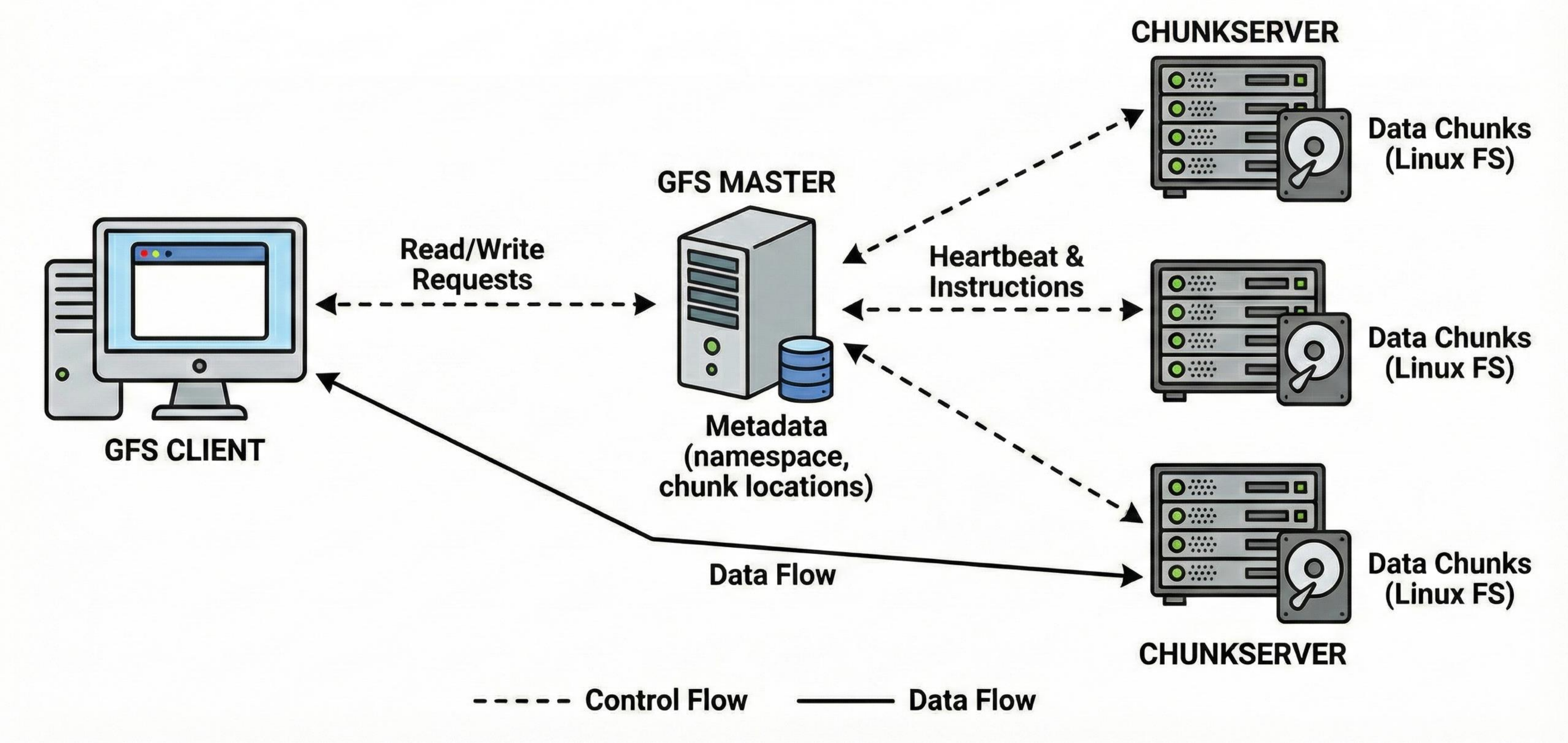
Google File System

- ~2003 designed by Google
- Although called file system
  - does not hook into VFS
  - is not POSIX-compliant
- Why is it called file system then?
  - hierarchical namespace
  - basic file operations, e.g., create/open/read/write

GFS: Design Context

- Hardware
  - move away from expensive mainframe
  - commodity hardware where failure is the norm, not the exception
- Workload (big data mapreduce)
  - throughput was prioritized over latency
- Access pattern
  - huge files (GBs) with sequential appends and streaming reads
  - random writes were rare

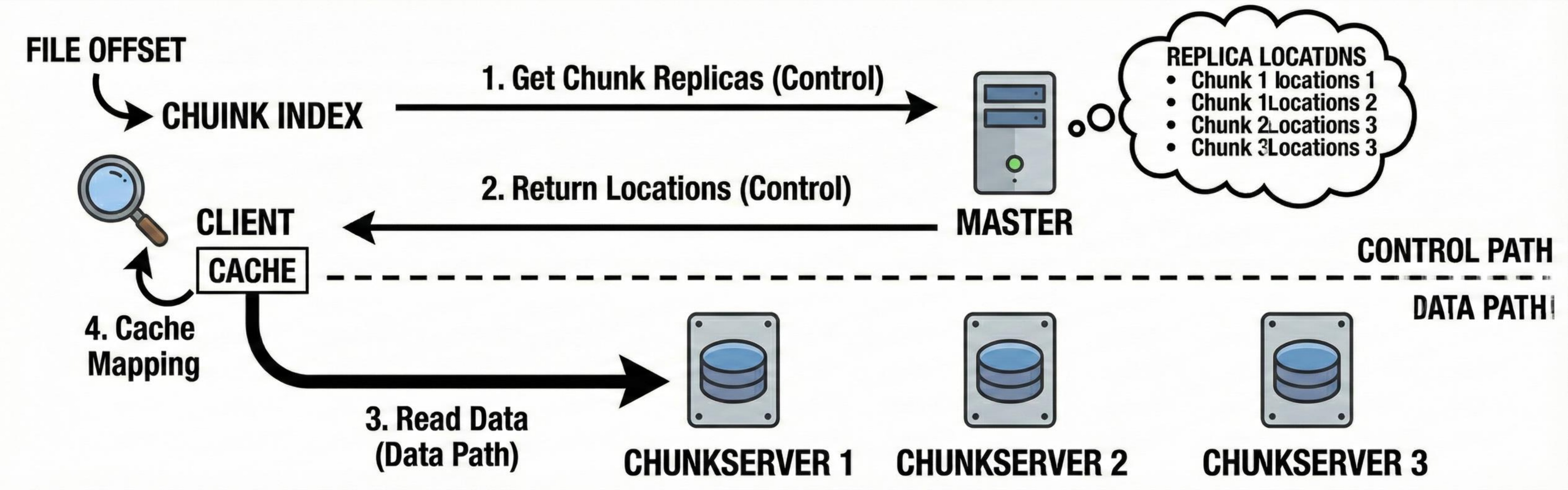
GFS Architecture



GFS Architecture

- Master (metadata)
  - single point of truth: one machine holds all metadata
    - namespace, mapping: file->chunk->location, per-chunk lease state, access control…
  - in-memory: all metadata are in DRAM (fast but limit scalability)
  - operation log: metadata mutation with periodic checkpoint
- Chunkservers (data)
  - dumb storage: store 64MB chunks as standard Linux file
  - no caching: use page cache
- Client (smart driver)
  - metadata caching,
  - direct connection with chunkserver

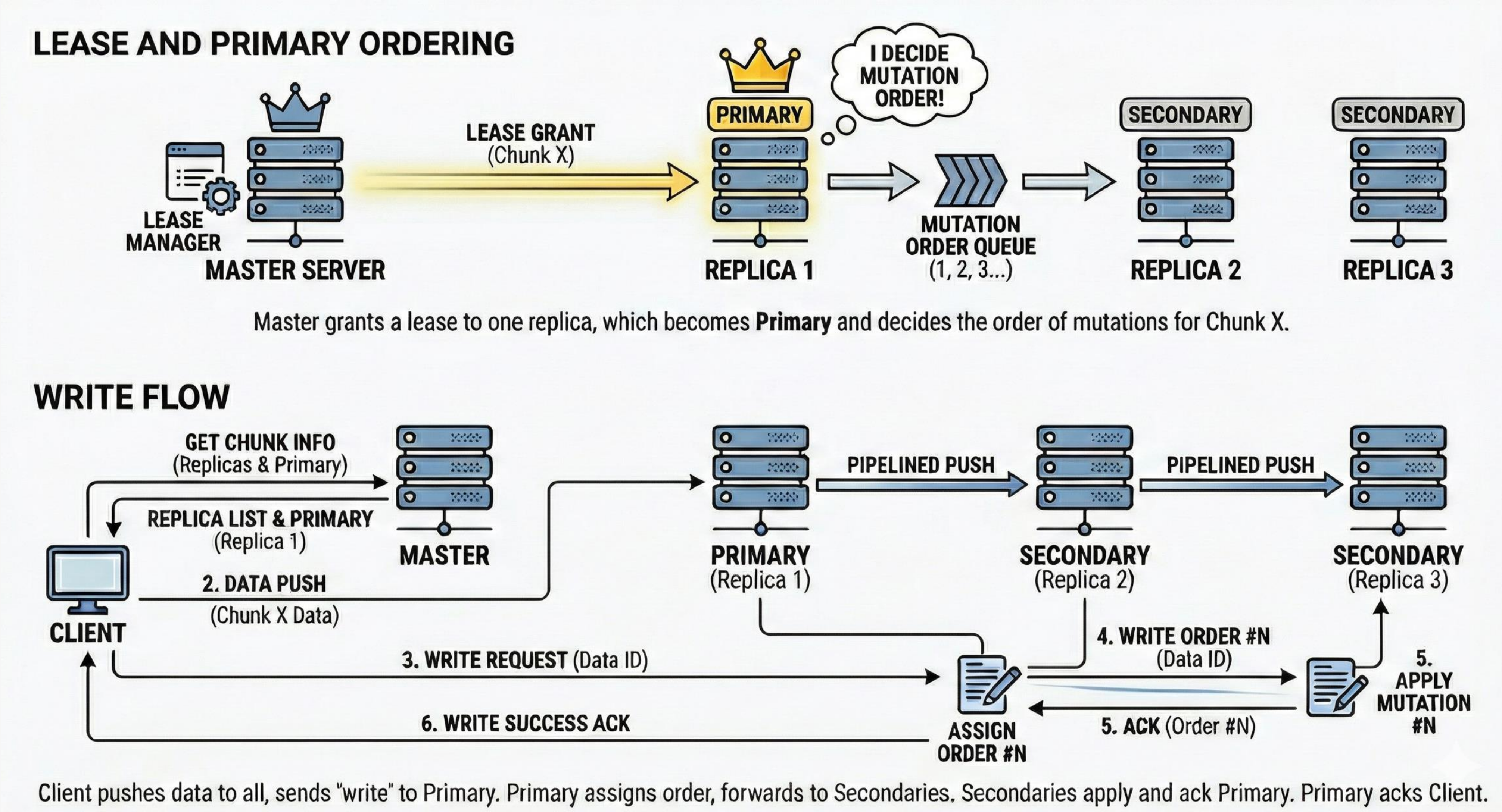
GFS: Read Request Flow



GFS: Read Request Flow

- Client calculates chunk index from file offset
- Client asks master for replicas of the chunk
- Master returns replica locations
- Client reads from a chunkserver and caches the mapping
- **Master is on the control path, not the data path**

GFS: Write Request Flow



GFS: Write Request Flow

- Lease and primary ordering
  - for each chunk, master grants a lease to one replica, which becomes primary that decides the order of mutations
- Write flow
  - client asks master for chunk replicas and which is primary
  - client pushes data to all replicas (pipelined)
  - client sends “write” to the primary
  - primary assigns a mutation order and forwards that order to secondaries
  - secondaries apply the mutation in that order, ack primary
  - primary acks client
- Record append primitive enable concurrent appends

Main Innovations in GFS

- Embrace hardware failure with continuous repair
- Decoupling control and data plane
    - only control goes to master so that a single master node is sufficient
- Pipelined replication
- Atomic record append
    - a new primitive enabling concurrent appends without synchronization
- Single master with metadata in memory
    - fast and consistent

From GFS to Colossus

- Shift from batch processing (MapReduce) to real-time processing, GFS hit limits
  - metadata bottleneck: one master cannot handle the trillions of small files
  - latency: retry on failure model causes high tail latency
  - storage efficiency: 3x replication is expensive, move to Reed-Solomon erasure coding

Hadoop Distributed FS

- Inspired by GFS
- Started in 2004 as part of Apache Nutch
  - creator: Doug Cutting and Mike Cafarella
- Yahoo hired Doug in 2006
  - HDFS ran on 20-node cluster
  - Yahoo provided resources to turn the prototype into a production system
- Architecture similar to GFS

HDFS vs. GFS Similarities

Concept	GFS Term	HDFS Term	Description
Coordinator	Master	**NameNode**	Manages namespace, file block mapping, and access control, entire namespace in RAM.
Worker	Chunkserver	**DataNode**	Stores actual data blocks. Sends heartbeats and block reports to the NameNode.
Unit of Storage	Chunk (64MB)	**Block** (128MB)	Large size minimizes metadata overhead.
Journal	Operation Log	**EditLog**	Persists metadata changes (WAL).
Snapshot	Checkpoint	**FsImage**	Snapshot of the file system metadata.

HDFS vs. GFS Differences

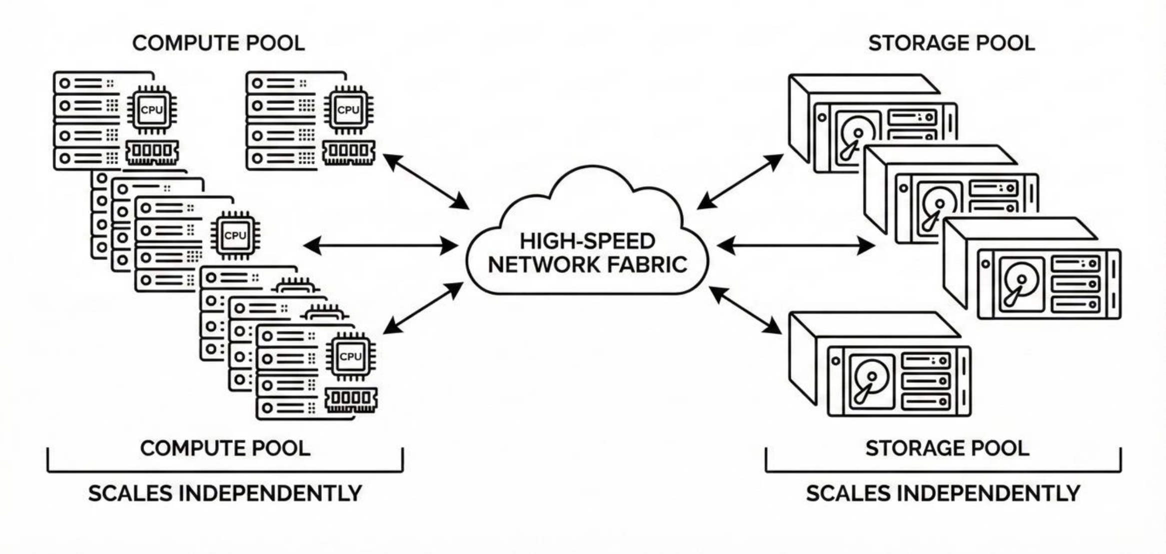
- Data locality
  - move computation is cheaper than moving data
  - coupled with Hadoop (compute runs on the same node)
- Write once, read many
  - append is added later, but does not support concurrent write/append
- Immutability
  - HDFS blocks are immutable after finishing
- High availability
  - active and standby NameNode
  - JournalNode to store metadata change, quorum-based solution

Hadoop Ecosystems

- Hive (Facebook): SQL layer (SQL->MapReduce)
  - make “Big Data” accessible to more users
- Presto (Facebook): SQL speed layer
  - distributed SQL engine query data directly without MapReduce
- Kafka (LinkedIn): Data pipeline
  - bridge between ingestion and HDFS

Issues with HDFS

- Limited scalability (one NameNode)
- Small file huge overhead
- Couple compute with storage
    - the past decade moved towards disaggregated storage
        - better scalability
        - easy management, compute is stateless
        - faster networks



Summary

- Distributed storage systems
- NFS and AFS
- GFS and HDFS

Next time

- Cluster file systems
- Distributed block storage
- (Distributed) object storage
- Distributed data structure store
- Other distributed storage systems