Macroarea di Ingegneria
Dipartimento di Ingegneria Civile e Ingegneria Informatica

(Big) Data Storage Systems

Corso di Sistemi e Architetture per Big Data

A.A. 2022/23

Valeria Cardellini

Laurea Magistrale in Ingegneria Informatica

The reference Big Data stack

High-level Frameworks

Data Processing

Data Storage

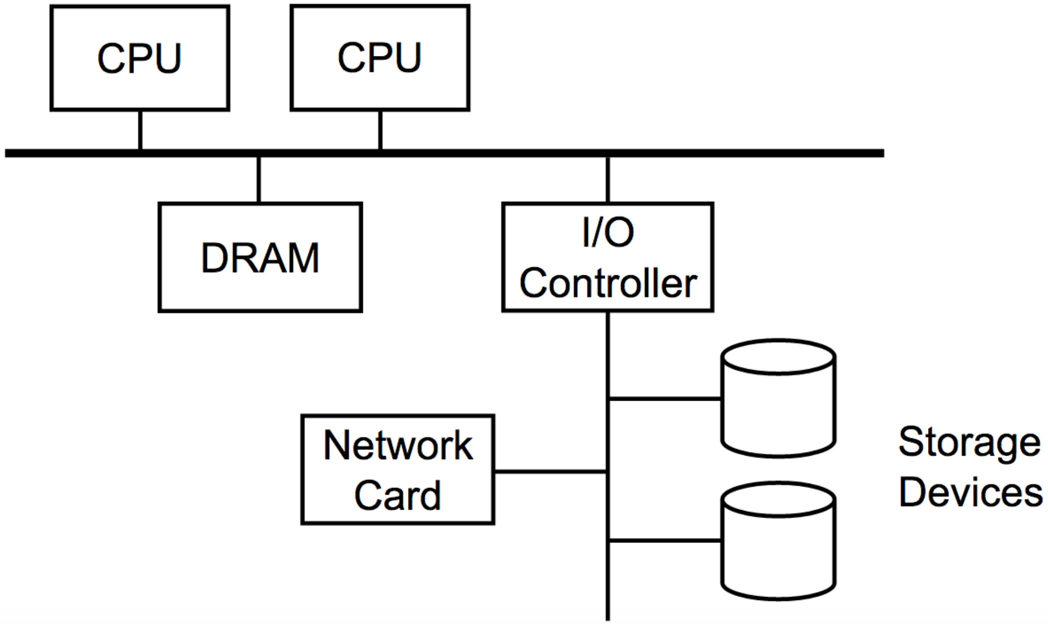
Resource Management

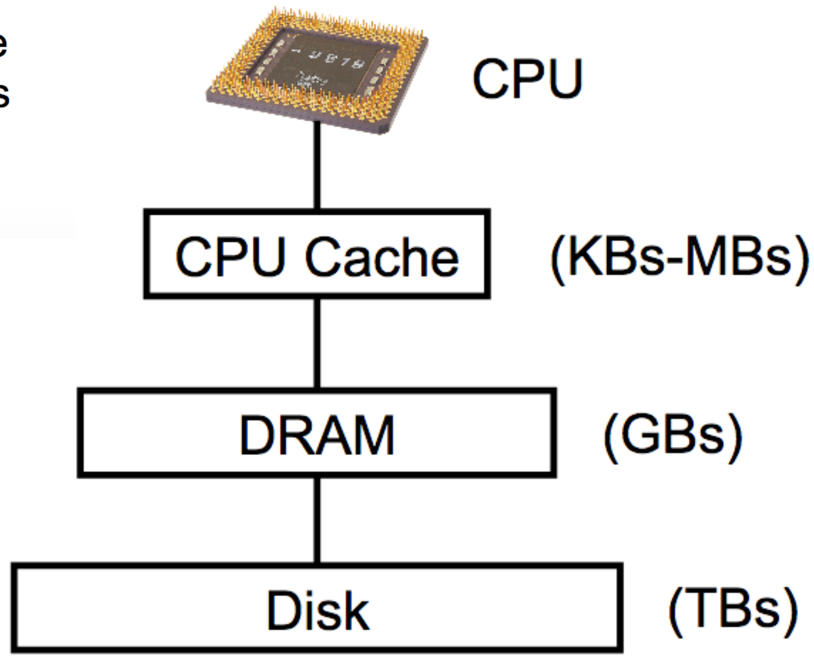
Support / Integration

Where storage sits in the Big Data stack

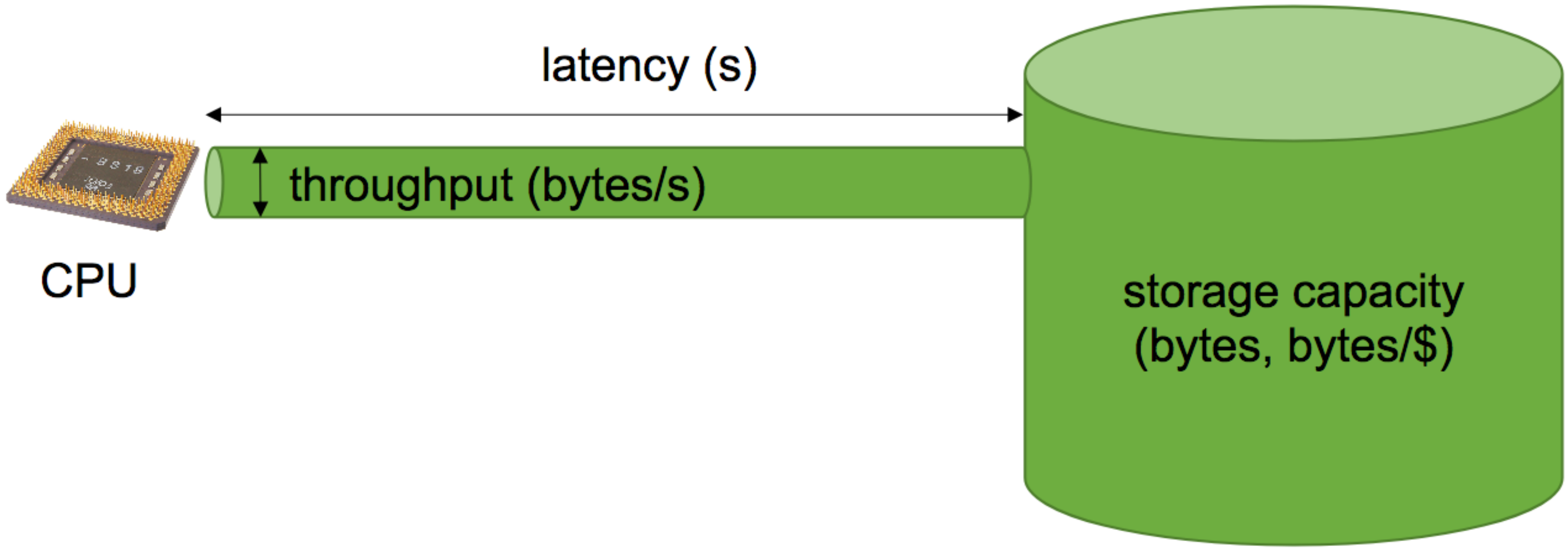
- An example of data lake architecture

Typical server architecture and storage hierarchy



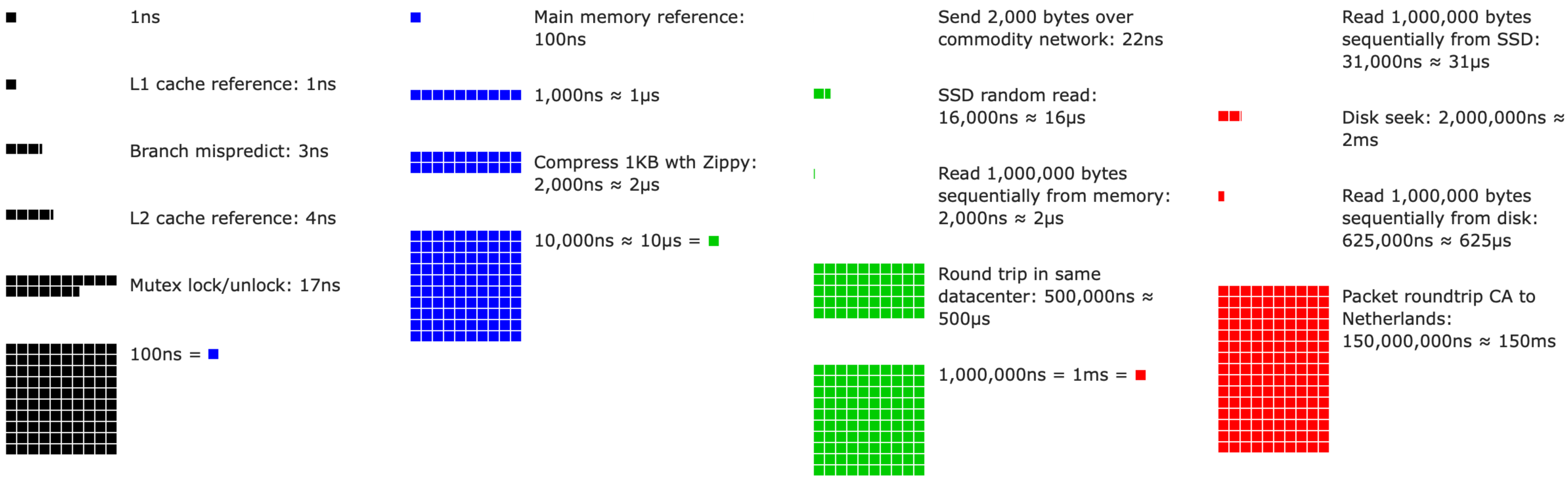


Storage performance metrics



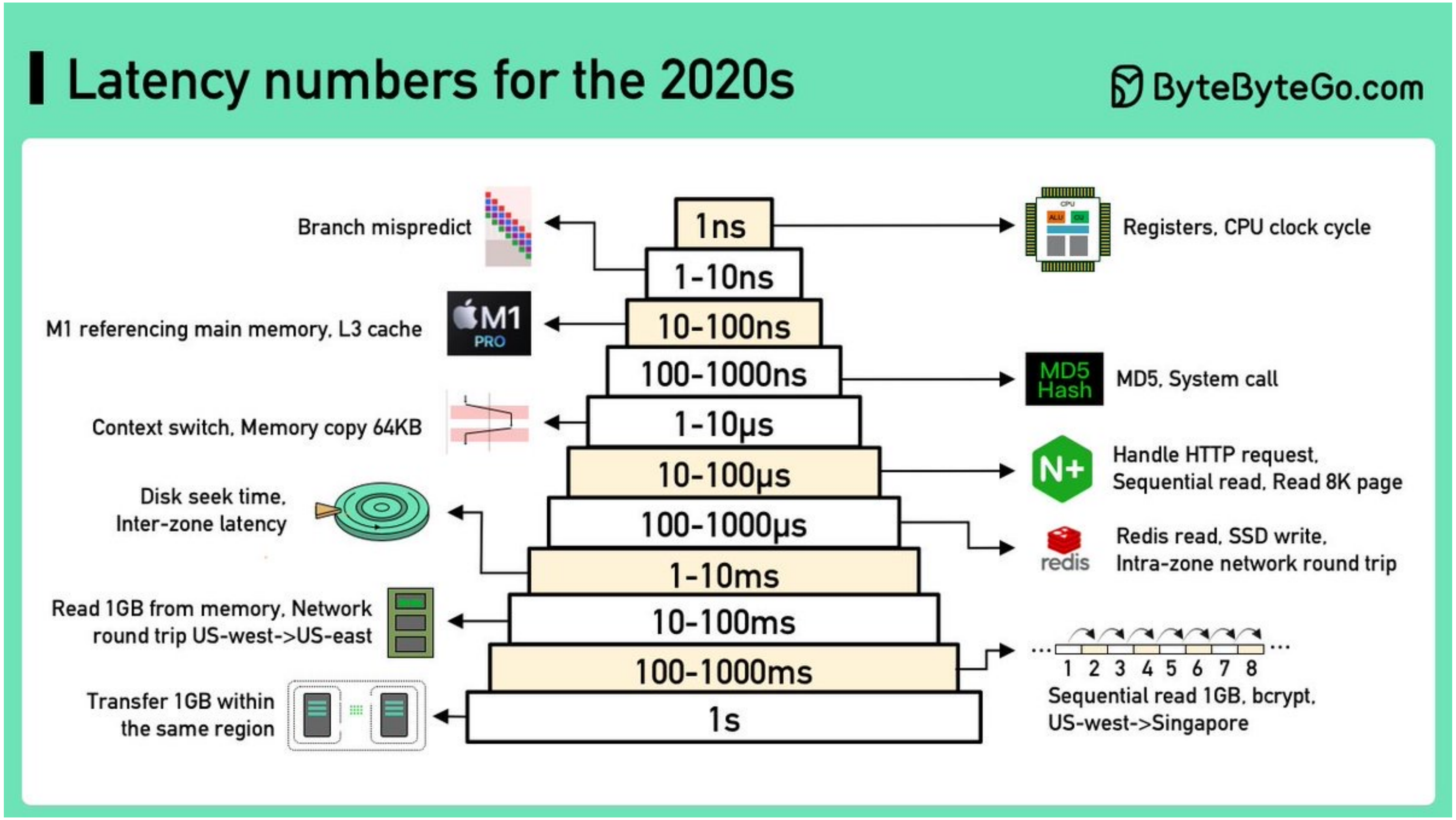
Where to store data?

- See “Latency numbers every programmer should know” (updated in 2020) bit.ly/2pZXIU9



Where to store data?

- "Latency numbers every programmer should know for the 2020s" www.youtube.com/watch?v=FqR5vESuKe0



Maximum attainable throughput

- Varies significantly by device
  - 50 GB/s for RAM
  - 2 GB/s for NVMe (Non-Volatile Memory Express) SSD
  - 130 MB/s for hard disk
- Assumes large reads ($\gg$1 block)

Hardware trends over time

- Capacity/$ grows exponentially at a fast rate (e.g. double every 2 years)
- Throughput grows at a slower rate (~5% per year), but new interconnects help
- Latency does not improve much over time

Data storage: the classic approach

- **File**
  - Group of data, whose structure is defined by the file system
- File system
  - Controls how data are structured, named, organized, stored and retrieved from disk
  - Usually: single (logical) disk (e.g., HDD/SDD, RAID)

- **Relational database (DB)**
  - Organized/structured collection of data (e.g., entities, tables)
- Database management system (DBMS)
  - Provides a way to organize and access data stored in files
  - Enables: data definition, update, retrieval, administration

What about Big Data?

Storage capacity and data transfer rate have increased massively over the years

HDD
Capacity: ~1TB
Throughput: 250MB/s

SSD
Capacity: ~1TB
Throughput: 850MB/s

Let's consider the latency (time needed to transfer data*)

Data Size	HDD	SSD
10 GB	40s	12s
100 GB	6m 49s	2m
1 TB	1h 9m 54s	20m 33s
10 TB	?	?

* we consider no overhead

We need to scale out!

General principles for scalable data storage

- Scalability and high performance
  - Need to face the continuous growth of data to store
  - Use multiple nodes as storage
- Ability to run on commodity hardware
  - But hardware failures are the norm rather than the exception
- Reliability and fault tolerance
  - Transparent data replication
- Availability
  - Data should be available to serve requests when needed
  - CAP theorem: trade-off with consistency

Scalable and resilient data storage solutions

Various forms of storage for Big Data:

- **Distributed file systems**
  - Manage (large) **files** on multiple nodes
  - E.g., Google File System, HDFS, GlusterFS

- **NoSQL data stores**
  - Simple and flexible **non-relational** data models
  - Horizontal scalability and fault tolerance
  - Key-value, column family, document, and graph stores
  - E.g., Redis, BigTable, Cassandra, MongoDB, HBase, DynamoDB
  - Also time series databases built on top of NoSQL (e.g.,: InfluxDB, KairosDB)

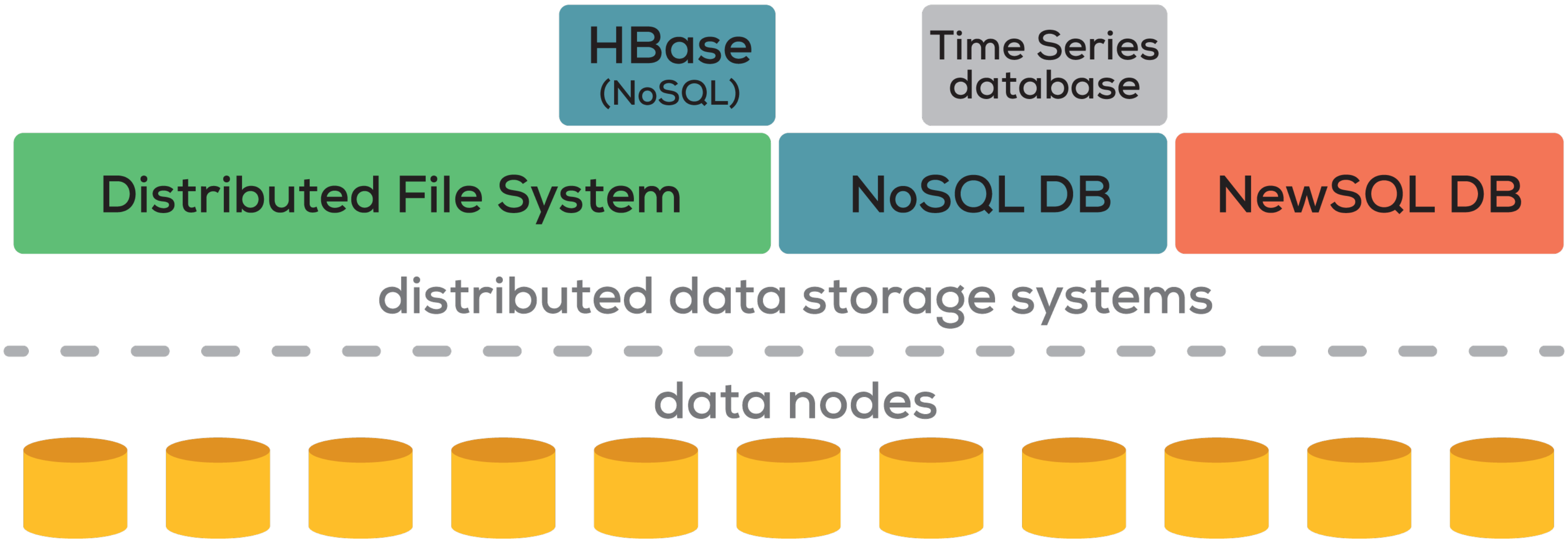
- **NewSQL databases**
  - Add horizontal scalability and fault tolerance to **relational** model
  - E.g., VoltDB, Google Spanner, CockroachDB

Data storage in the Cloud

- Main goals:
  - Massive scaling “on demand” (elasticity)
  - Fault tolerance
  - Durability (versioned copies)
  - Simplified application development and deployment
  - Support for cloud-native apps (serverless)
- Public Cloud services for data storage
  - Object stores: Amazon S3, Google Cloud Storage, Microsoft Azure Storage, …
  - Relational databases (DBaaS): Amazon RDS, Amazon Aurora, Google Cloud SQL, Microsoft Azure SQL Database, …
  - NoSQL data stores: Amazon DynamoDB, Amazon DocumentDB, Google Cloud Bigtable, Google Datastore, Microsoft Azure Cosmos DB, MongoDB Atlas, …
  - NewSQL databases: Google Cloud Spanner, …
  - Serverless databases: Google Firestore, CockroachDB, …

Scalable and resilient data storage solutions

Whole picture of different solutions we will consider



Distributed File Systems (DFS)

- Primary support for data management
- Manage data storage across a network of machines
  - Usually locally distributed, in some case geo-distributed
- Provide an interface whereby to store information in the form of files and later access them for read and write operations
- Several solutions with different design choices
  - **GFS**, **HDFS** (GFS open-source clone): designed for batch applications with large files
  - **Alluxio**: in-memory (high-throughput) storage system
  - **GlusterFS**: scalable network file system
  - Lustre: open-source, large-scale distributed file system
  - Ceph: open-source, distributed object store with Ceph File System on top

Case study: Google File System (GFS)

Assumptions and Motivations

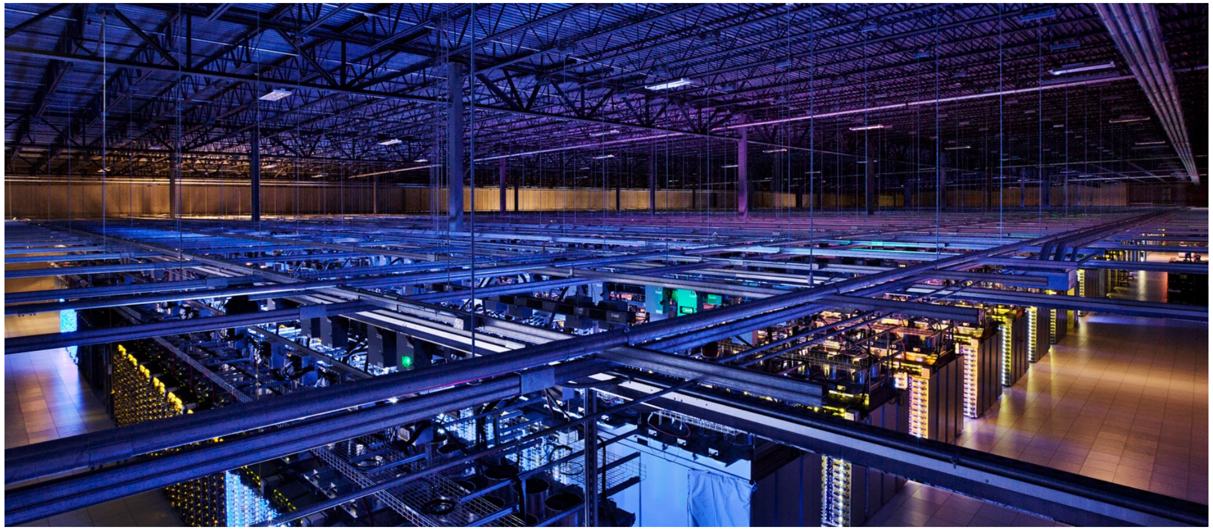
- System built from inexpensive commodity hardware that often fails
  - 60,000 nodes, each with 1 failure per year: 7 failures per hour!
- System stores large files
- Large streaming/contiguous reads, small random reads
- Many large, sequential writes that append data
  - Concurrent clients can append to same file
- High sustained bandwidth is more important than low latency

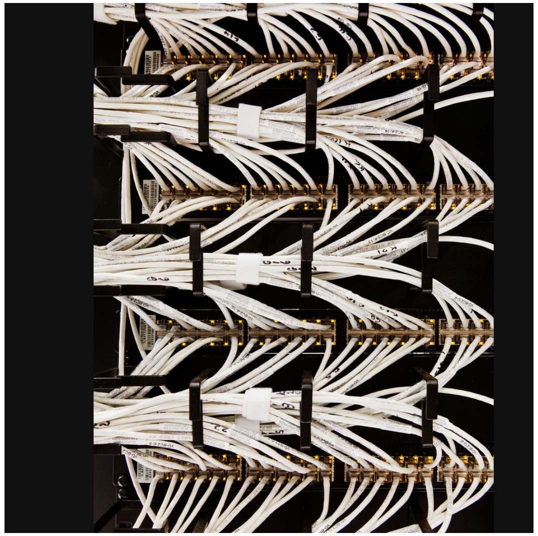
Ghemawat et al., The Google File System, *Proc. ACM SOSP '03*

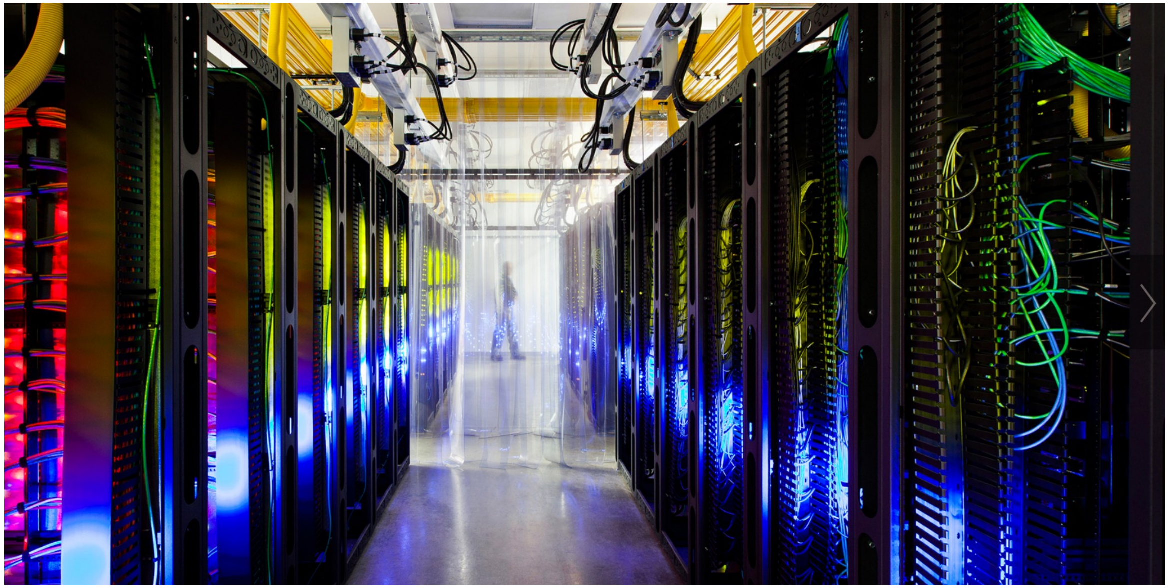
Case study: Google File System

- Distributed file system implemented in user space
- Manages (very) large files: usually multi-GB
- **Divide et impera**: file is split into fixed-size chunks
- ***Chunk***:
  - Fixed size (either 64MB or 128MB)
  - Transparent to users
  - Stored as plain file on chunk servers
- Write-once, read-many-times pattern
  - Efficient *append* operation: appends data at the end of file *atomically at least once* even in the presence of concurrent operations (minimal synchronization overhead)
- Fault tolerance and high availability through chunk replication, no data caching

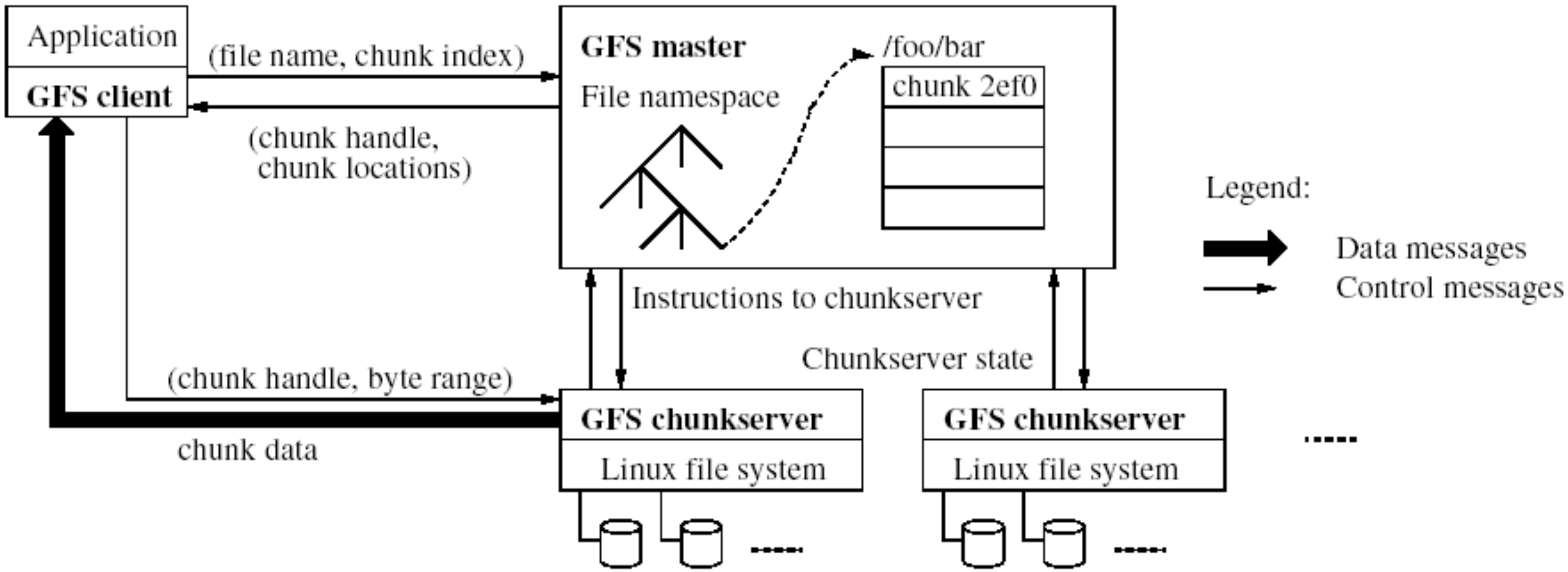
GFS operation environment





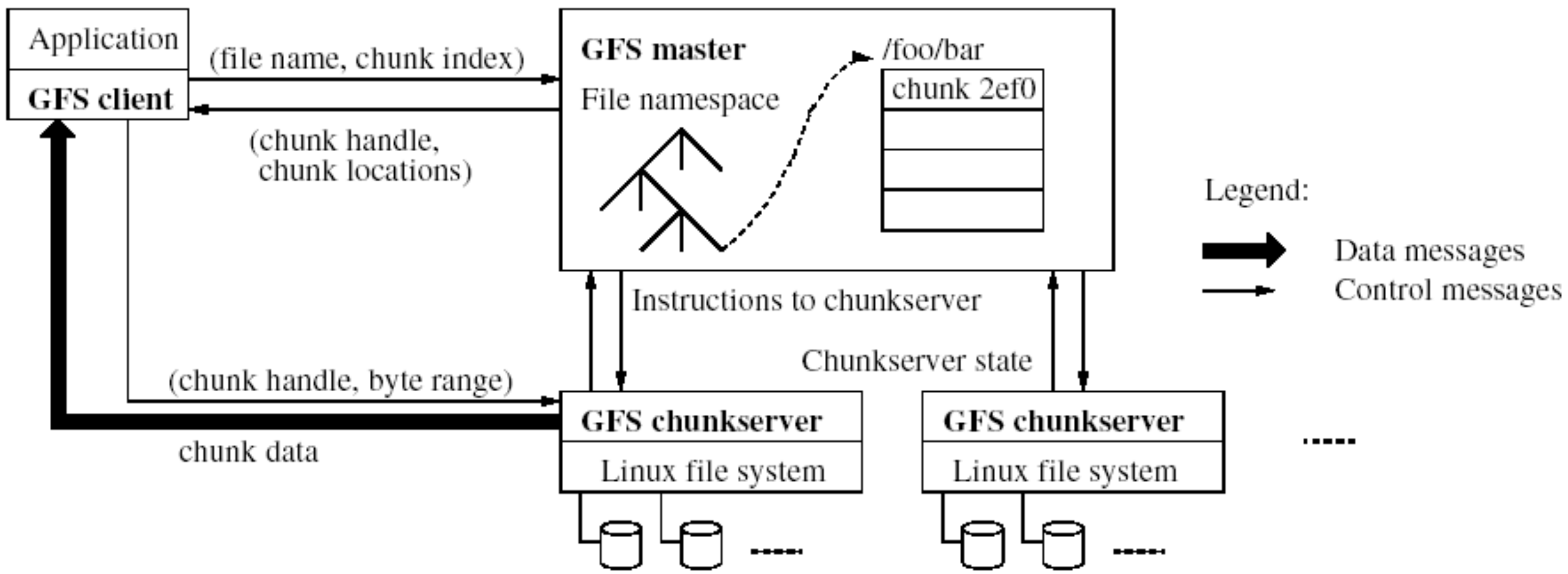


GFS: Architecture



- Master
  - Single, centralized entity (to simplify the design)
  - Manages file metadata (stored in memory)
    - Metadata: access control information, mapping from files to chunks, locations of chunks
  - Does not store data (i.e., chunks)
  - Manages operations on chunks: create, replicate, load balance, delete

GFS: Architecture



- Chunk servers (100s – 1000s)
  - Store chunks as files
  - Spread across cluster racks
- Clients
  - Issue *control* (metadata) requests to GFS master
  - Issue *data* requests to GFS chunkservers
  - Cache metadata, do not cache data (simplifies system design)

GFS: Metadata

- Master stores 3 major types of metadata:
  - File and chunk namespace (directory hierarchy)
  - Mapping from files to chunks
  - Current locations of chunks
- Metadata are stored in memory (64B per chunk)
  - ✓ Fast, easy and efficient to scan the entire state
  - ✗ Number of chunks is limited by amount of master's memory
    "The cost of adding extra memory to the master is a small price to pay for the simplicity, reliability, performance, and flexibility gained"
- Master also keeps an operation log where metadata changes are recorded
  - Log is persisted on master's disk and replicated for fault tolerance
  - Master can recover its state by replaying operation log
  - Checkpoints for fast recovery

GFS: Chunk size

- Chunk size is either 64 MB or 128 MB
  - Much larger than typical block sizes
- Why? Large chunk size reduces:
  - Number of interactions between client and master
  - Size of metadata stored on master
  - Network overhead (persistent TCP connection to the chunk server over an extended period of time)
- Each chunk is stored as a plain Linux file
- Cons
  - ✗ Wasted space due to internal fragmentation
  - ✗ Small files consist of a few chunks, which get lots of traffic from concurrent clients (can be mitigated by increasing replication factor)

GFS: Fault tolerance and replication

- Master controls and maintains the replication of each chunk on several chunk servers
  - At least 3 replicas on different chunk servers
  - Replication based on primary-backup schema
  - Replication degree > 3 for highly requested chunks
- Multi-level placement of replicas
  - Different machines, same rack + availability and reliability
  - Different machines, different racks + aggregated bandwidth
- Data integrity
  - Chunk divided in 64KB blocks; 32B checksum for each block
  - Checksum kept in memory
  - Checksum checked every time app reads data

GFS: Master operations

- Stores metadata
- Manages and locks namespace
  - Namespace represented as a lookup table
  - Read lock on internal nodes and read/write lock on leaves: read lock allows concurrent mutations in the same directory and prevents deletion, renaming or snapshot
- Communicates periodically with each chunk server using RPC
  - Sends instructions and collects chunk server state (*heartbeat* messages)
- Creates, re-replicates and rebalances chunks
  - Balances chunk servers' disk space utilization and load
  - Distributes replicas among racks to increase fault tolerance
  - Re-replicates a chunk as soon as the number of its available replicas falls below the replication degree

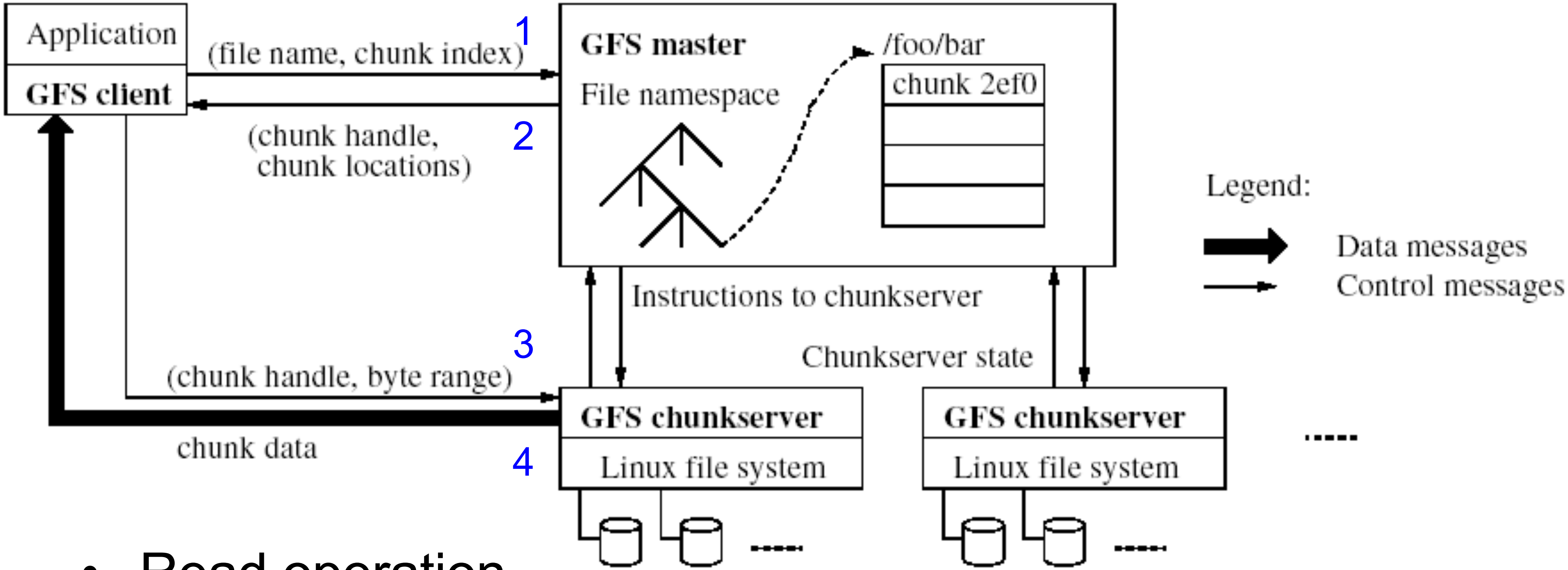
GFS: Master operations

- Garbage collection
  - File deletion logged by master
  - Deleted file is renamed to a hidden name with deletion timestamp, so that real deletion is postponed and file can be easily recovered in a limited timespan
- Stale replica detection
  - Chunk replicas may become stale if a chunk server fails or misses updates to chunk
  - For each chunk, the master keeps a chunk version number
  - Chunk version number updated at each chunk mutation
  - Master removes stale replicas during garbage collection

GFS: Interface

- Files are organized in directories
  - No data structure to represent a directory
- Files are identified by their pathname
  - No alias support
- GFS supports traditional file system operations (but not Posix.compliant)
  - **`create`**, **`delete`**, **`open`**, **`close`**, **`read`**, and **`write`**
- Supports also 2 special operations:
  - **`snapshot`**: makes a copy of a file or a directory tree at low cost (based on copy-on-write techniques)
  - **`record append`**: allows multiple clients to append data to the same file concurrently, without overwriting one another's data

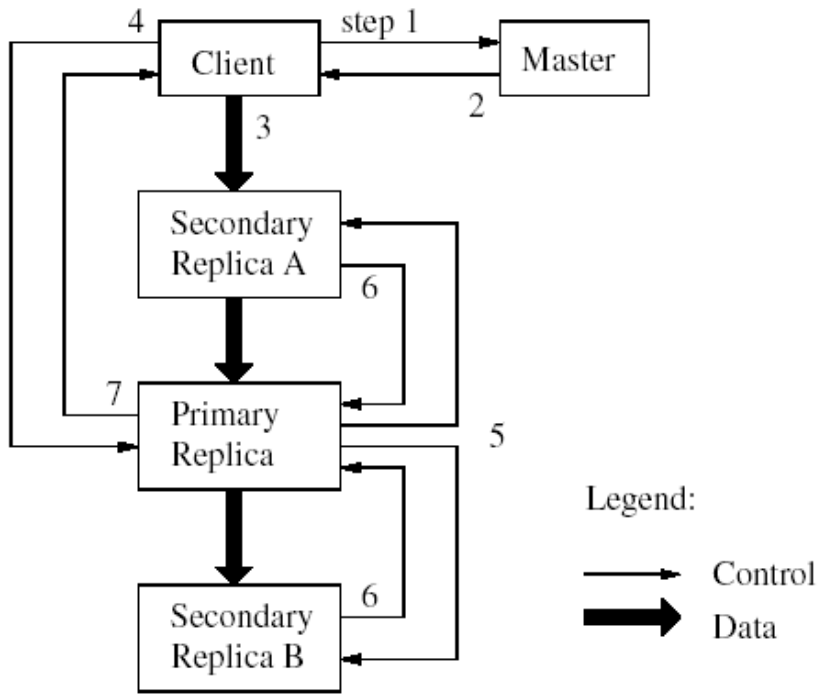
GFS: Read operations



- Read operation
  - Data flow is decoupled from control flow
  1) Client sends master: read(file name, chunk index)
  2) Master replies with chunk ID, chunk version number, replicas location
  3) Client sends read op to “closest” chunk server with replica: read(chunk ID, byte range)
  4) Chunk server replies with chunk data

GFS: Mutations

- Mutations are `write` or `append`
  - Mutations are performed at all chunk's replicas in the same order
- Based on a *lease* mechanism:
  - Goal: minimize management overhead at master
  - Master grants chunk lease to primary replica
  - Primary picks a serial order for all mutations to chunk
  - All replicas follow this order when applying mutations
  - Primary replies to client, see 7)
  - Leases renewed using periodic heartbeat messages between master and chunk servers



- Data flow is decoupled from control flow
- Client sends data to *any* of the chunk servers identified by master, which in turn pushes data to the other chunk servers in a chained fashion so to fully utilize network bandwidth

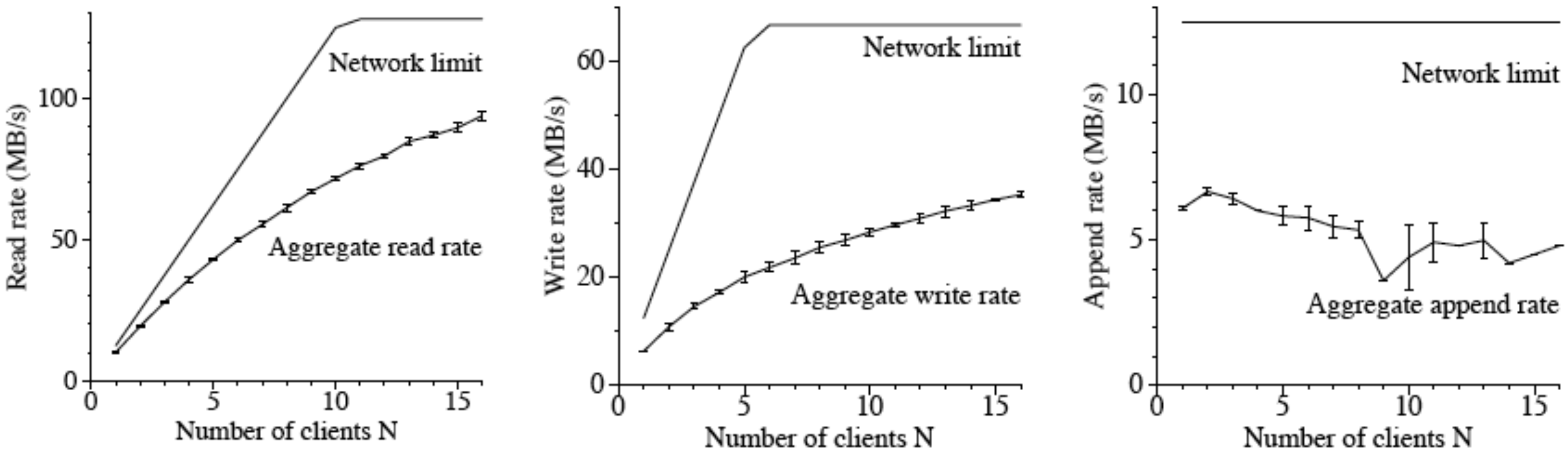
GFS: Atomic appends

- The client specifies only the data (with no offset)
- GFS appends data to the file *at least once* atomically (i.e., as one continuous sequence of bytes)
  - At offset chosen by GFS
  - Works with multiple concurrent writers
  - At least once: applications must cope with possible duplicates
- Append operations heavily used by Google's distributed apps
  - E.g., files often serve as multiple-producers/single-consumer queue or contain results merged from many clients (MapReduce scenario)

GFS: Consistency model

- Changes to namespace (e.g., file creation) are atomic
  - Managed exclusively by master with locking guarantees
- Changes to data are ordered as chosen by primary replica, but chunk server failures can cause inconsistency
- GFS has a “relaxed” model: eventual consistency
  - Simple and efficient to implement

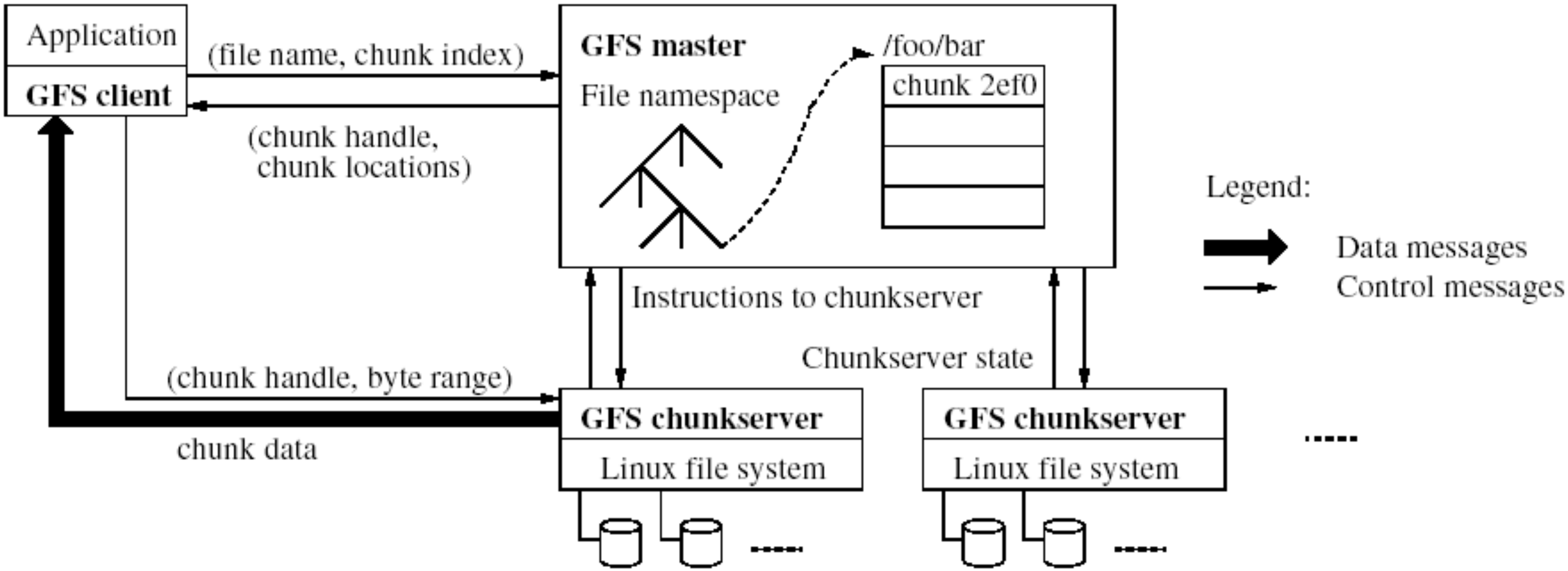
GFS performance



(a) Reads (b) Writes (c) Record appends

- Read performance is satisfactory (80-100 MB/s)
- But reduced write performance (30 MB/s) and relatively slow (5 MB/s) in appending data to existing files

GFS problems



What is the limitation of this architecture?

Single master → Single point of failure (SPOF)
Scalability bottleneck

GFS problems: Single master

- Solutions adopted in GFS to overcome issues related to single master
  - Overcome SPOF: by having multiple “shadow” masters that provide read-only access when the primary master is down
  - Overcome scalability bottleneck: by reducing interaction between master and client
    - Master stores only metadata (not data)
    - Client can cache metadata
    - Large chunk size
    - Chunk lease: delegates the authority of coordinating the mutations to the primary replica
- Overall, simple solutions

GFS summary

- GFS success
  - Used by Google to support search service and other apps
  - Availability and recoverability on commodity hardware
  - High throughput by decoupling control and data
  - Supports massive data sets and concurrent appends
- GFS problems (besides single master)
  - All metadata stored in master memory
    - "Limited" scalability: approximately 50M files, 10PB
  - Semantics not transparent to apps
  - Automatic failover added (but still takes 10 sec.)
  - Delays due to recovering from a failed replica chunk server delay the client
  - Performance not good for all apps
    - Designed for high throughput but not appropriate for latency-sensitive apps like Gmail, because GFS was designed (in 2001) for batch apps with large files

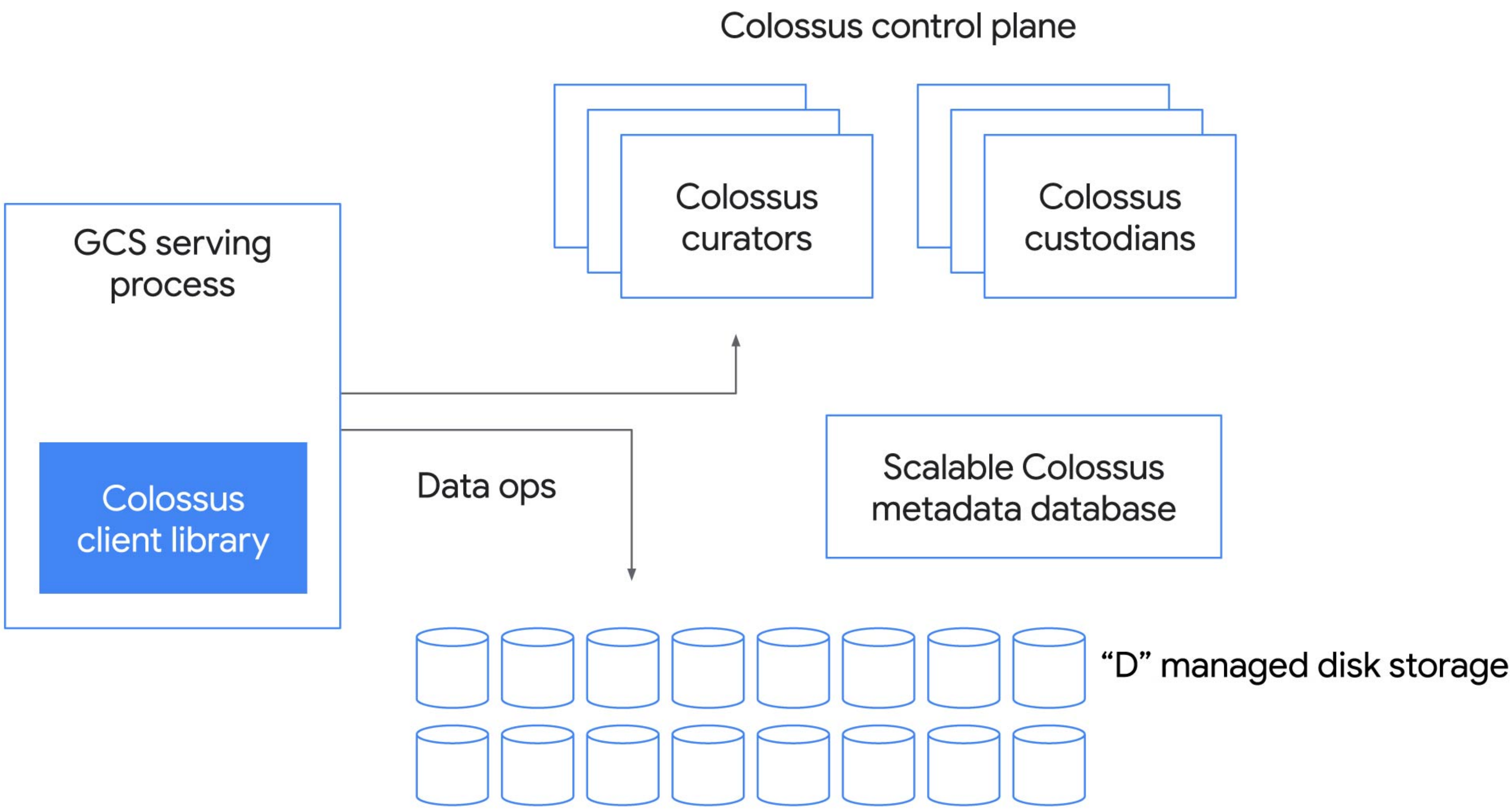
Colossus: successor of GFS

- Next-generation Google DFS (since 2010)
- Designed for a wide variety of Google services (YouTube, Maps, Photos, search ads, …)
- Can handle EB of storage, tens of thousands of servers
- Distributed masters, GFS chunk servers replaced by D
- Scalable metadata layer, built on top of Bigtable
- Error-correcting codes (e.g., Reed-Solomon)
- Can mix high-speed flash memory and disks for storage
- Client-driven encoding and replication
- Google Cloud services built on top of Colossus
  – Cloud Storage (object store) and Cloud Firestore (NoSQL data store)

Colossus under the hood: a peek into Google's scalable storage system, 2021.

https://www.youtube.com/watch?v=q4WC_6SzBz4

Colossus: key components



HDFS

- **Hadoop Distributed File System (HDFS)**
  - Open-source user-level distributed file system
  - Written in Java
  - GFS clone
    - Master/worker architecture
    - Data is replicated across a large cluster of commodity servers
  - De-facto standard for batch-processing frameworks: e.g., Hadoop MapReduce, Spark

Shafer et al., The Hadoop Distributed Filesystem: Balancing Portability and Performance, *Proc. ISPASS 2010*

HDFS: Design principles

- Large data sets: typical file size is GBs or TBs
- Simple coherency model: files follow write-once, read-many-times access pattern
  - E.g., MapReduce apps, web crawlers
- Commodity, low-cost hardware
  - HDFS is designed to work without noticeable interruption to users even when failures occur
- Portability across heterogeneous hardware and software platforms

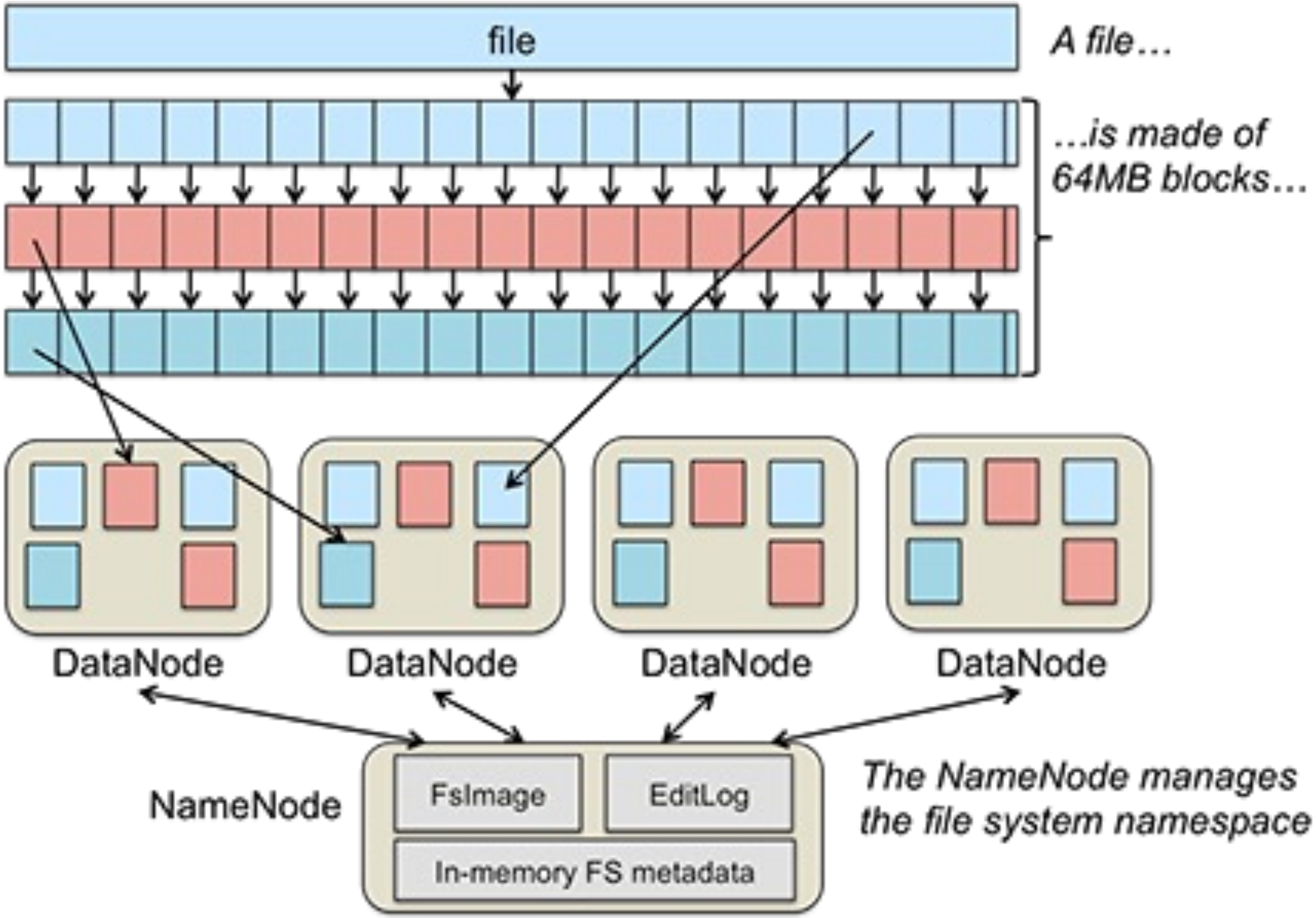
HDFS: Cons

HDFS does not work well with:

- Low-latency data access: optimized for delivering high data throughput, no low latency
- Lots of small files: number of files is limited by amount of master’s memory, which holds file system metadata
- Multiple writers, random writes

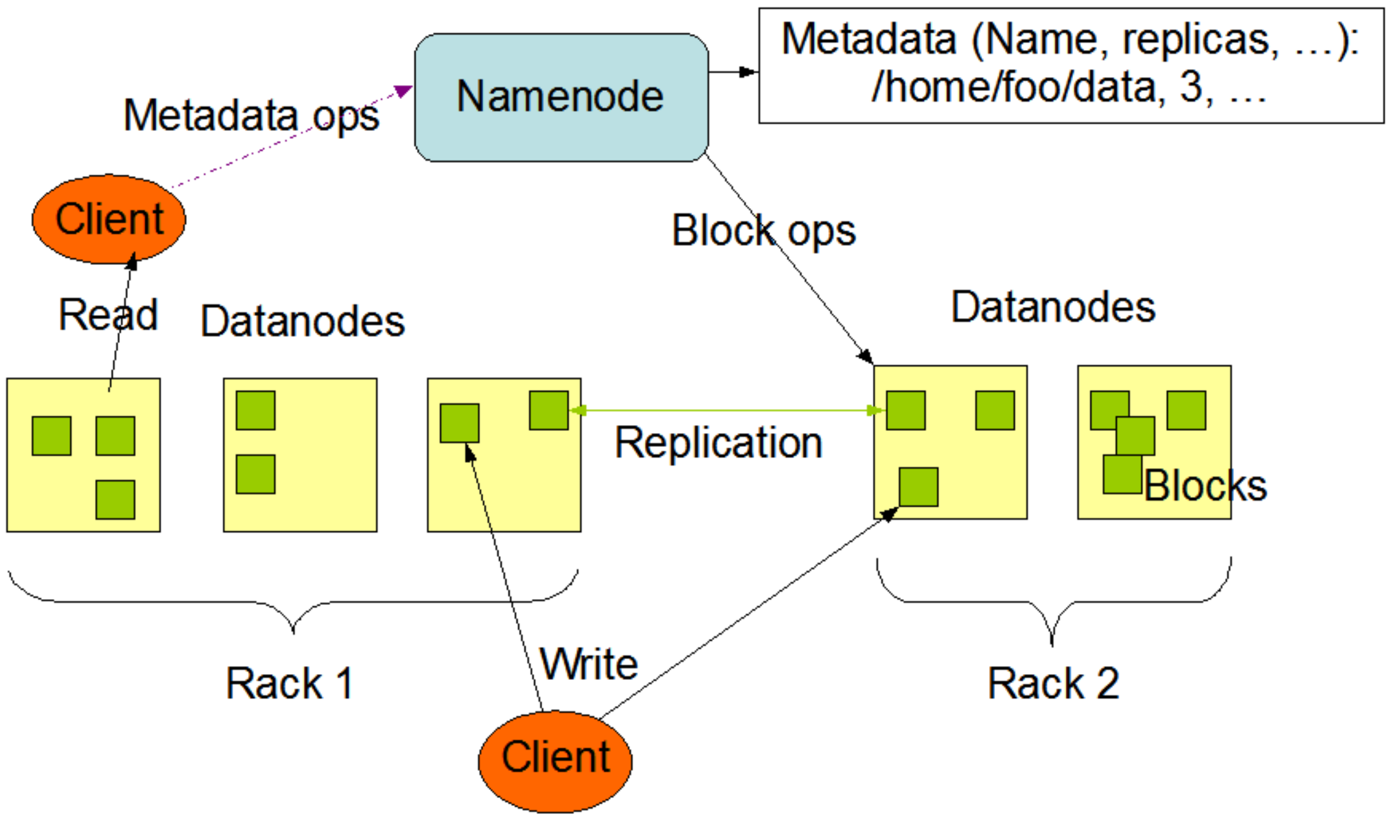
HDFS: File management

- File is split into **blocks** (chunks in GFS) which are stored in a set of storing nodes (named DataNodes)



HDFS: Architecture

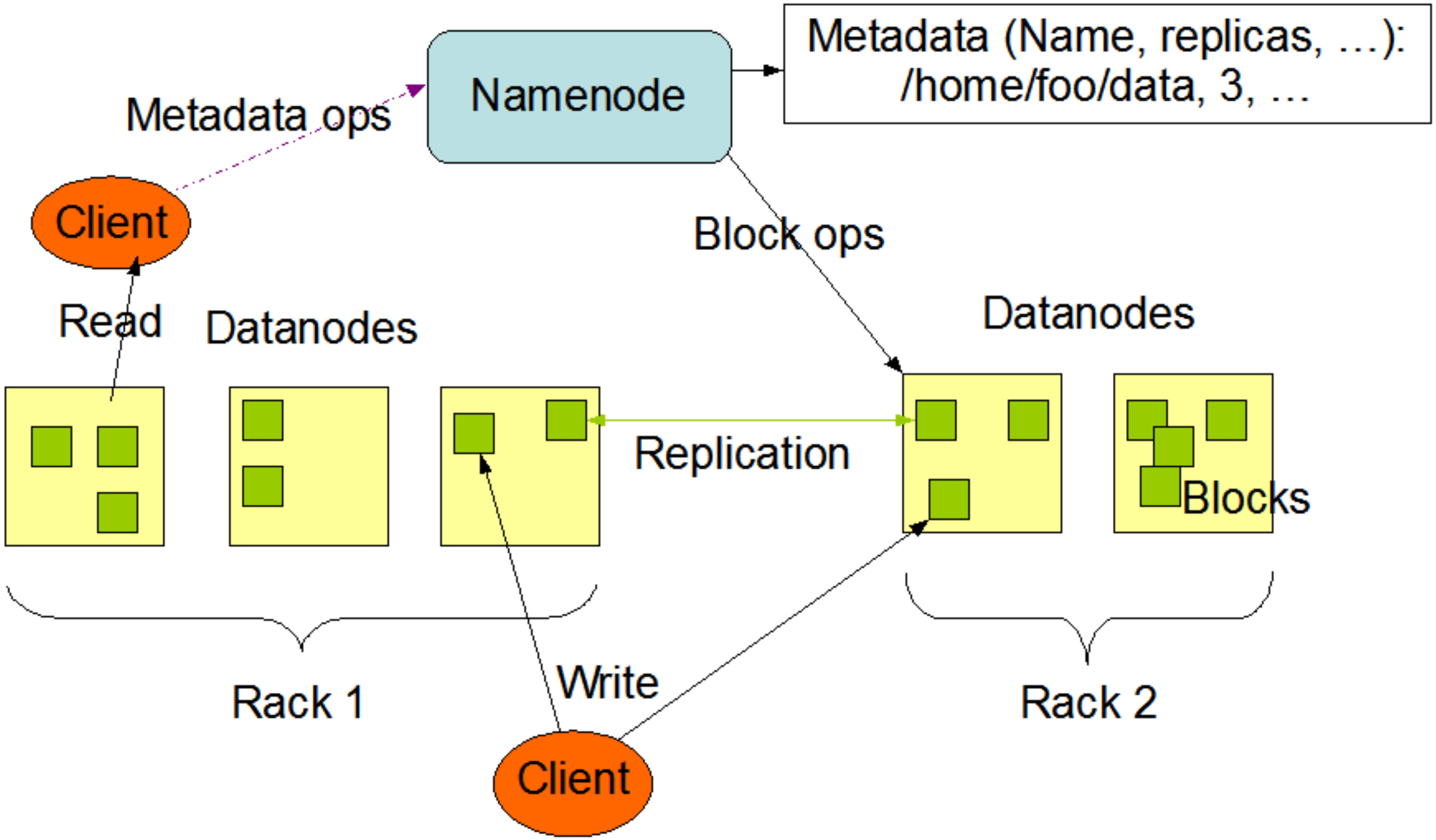
- Two types of nodes in a HDFS cluster:
  - One *NameNode* (master in GFS)
  - Multiple *DataNodes* (chunk servers in GFS)



HDFS: Architecture

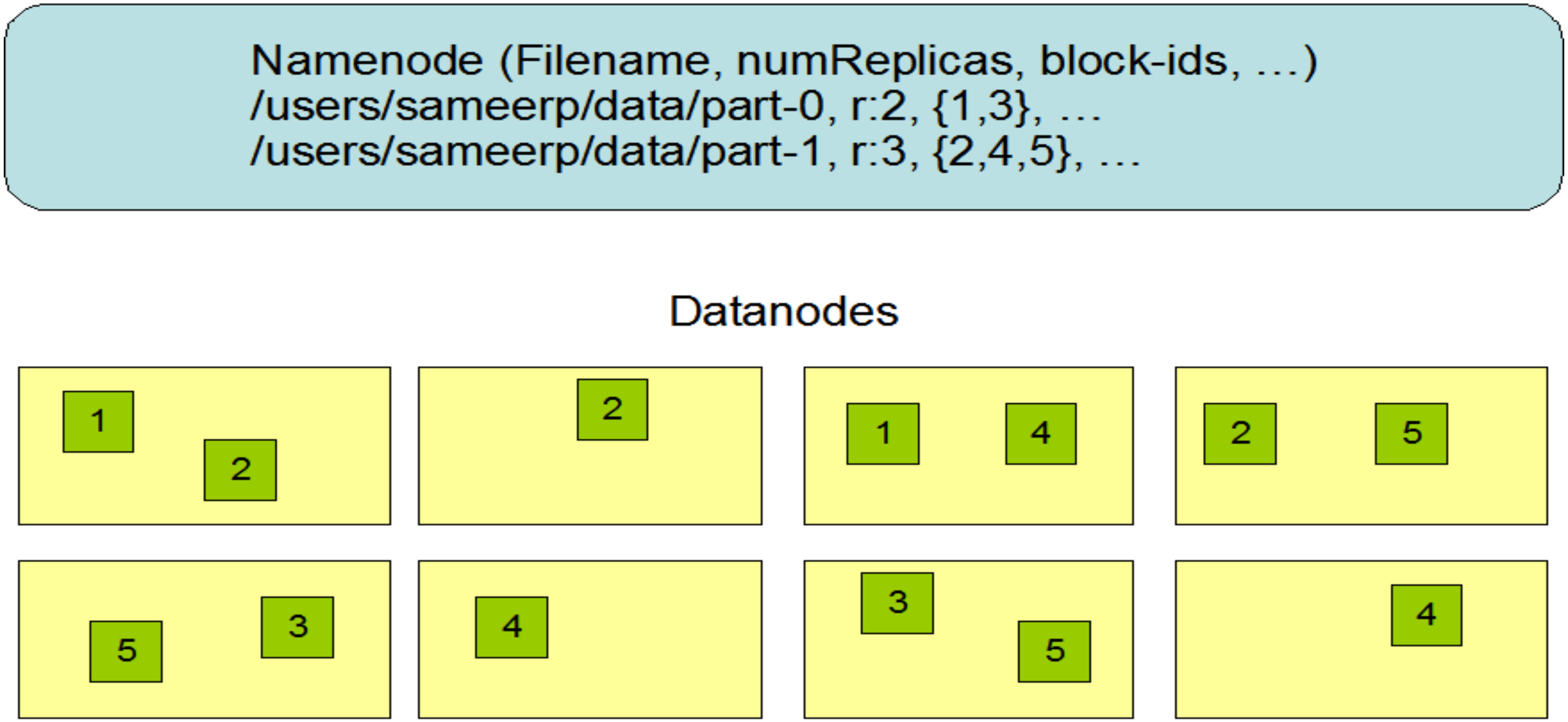
- **NameNode**:
  - Manages file system namespace
  - Manages metadata for files and directories
    - Including mapping of file blocks to DataNodes
- **DataNodes**:
  - Store and retrieve blocks when they are told to (by clients or NameNode)
  - Manage the attached storage
- Without NameNode, HDFS cannot be used
  - NameNode must be resilient to failures
- Large size blocks (default 64 MB): why so large?

HDFS: Architecture

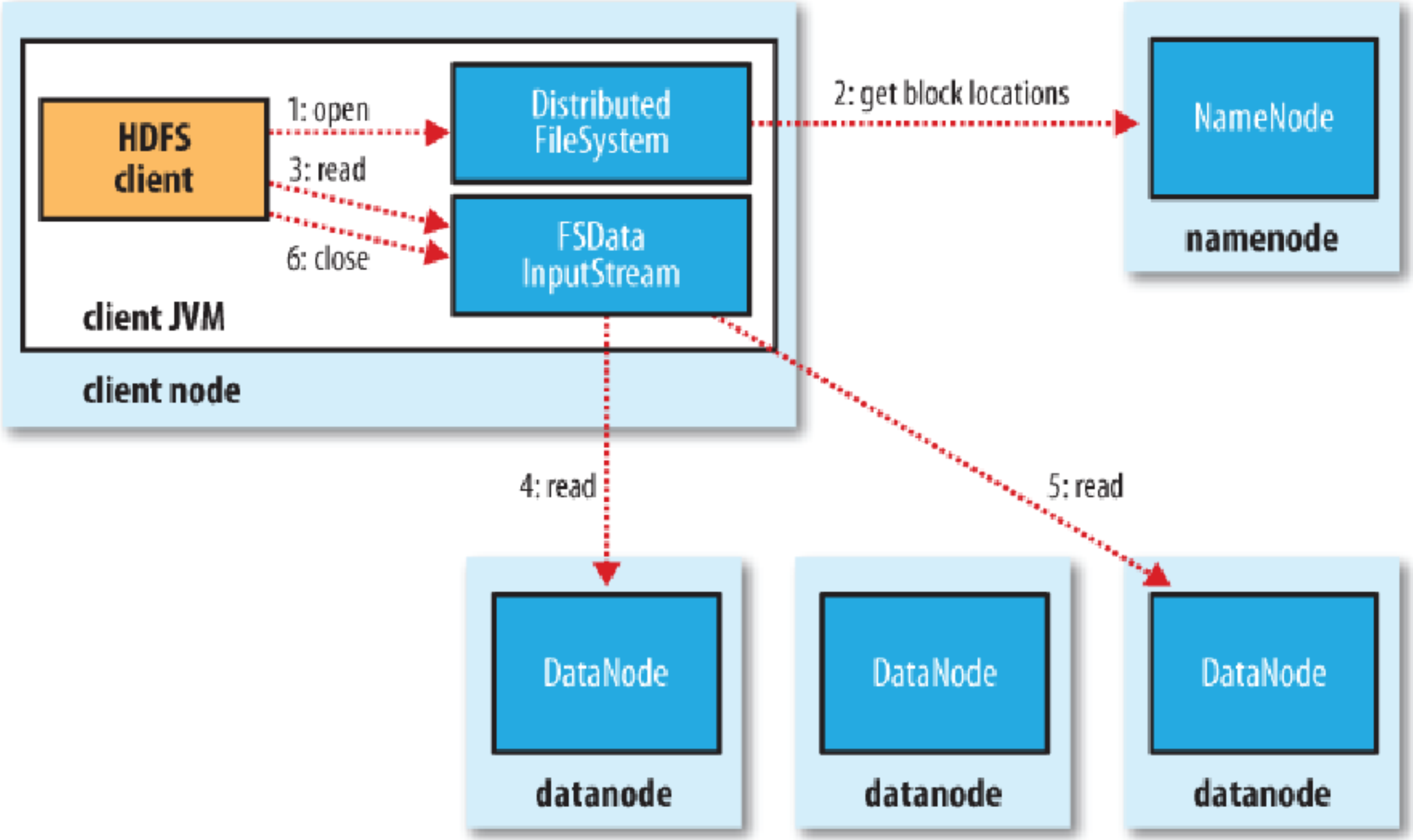


HDFS: Block replication

- NameNode periodically receives a heartbeat and a blockreport from each DataNode
  - Blockreport: list of all blocks on a given DataNode



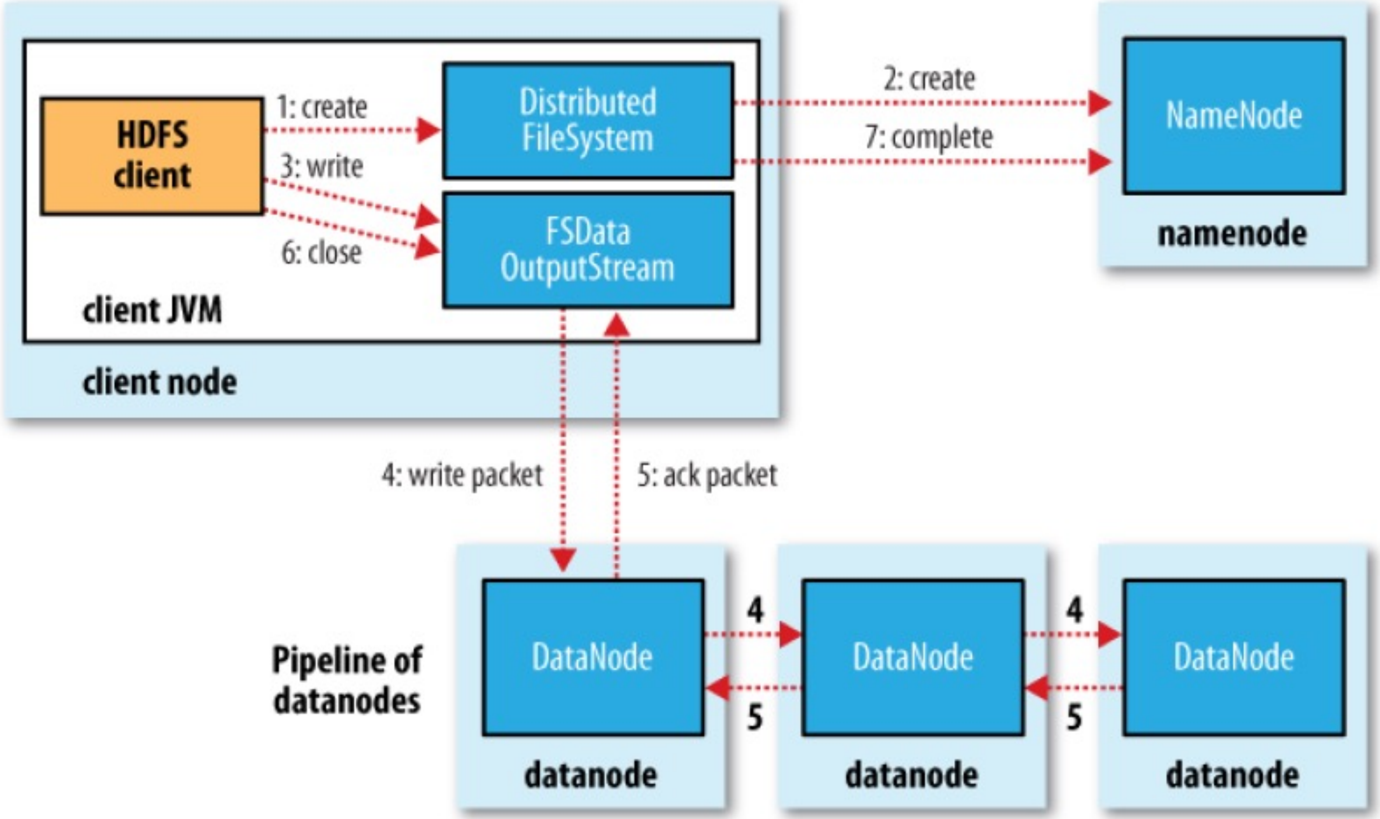
HDFS: File read



Source: "Hadoop: The definitive guide"

- NameNode is only used to get block location

HDFS: File write



Source: “Hadoop: The definitive guide”

- Clients ask NameNode for a list of suitable DataNodes
- This list forms a pipeline: first DataNode stores a copy of a block, then forwards it to the second, and so on

Enhancements in HDFS 3.x

- Erasure coding can be used in place of replication
  - ✓ Same level of fault-tolerance with less storage overhead: from 200% with 3x to 50%
  - ✗ Increase in network and processing overhead
  - – Two codes available: XOR and Reed-Solomon

    See https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/

- Support for more than 2 NameNodes
  - In HDFS 2.x only 1 active NameNode and 1 standby NameNode
  - HDFS high availability

    See https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HDFSHighAvailabilityWithNFS.html

Other Distributed File Systems: GlusterFS

- Linux-based, open source distributed file system https://www.gluster.org/
- Designed to be highly scalable
  - Scaling to several PB (up to 72 brontobytes!)
    - Brontobyte = 10^{27} or 2^{90} bytes

GlusterFS: Features

- Global namespace
  - Idea: metadata is a bottleneck
  - Solution: avoid centralized metadata server
    - No special node(s) with special knowledge of where files are or should be
  - Solution: use consistent hashing (similarly to Chord and Amazon's Dynamo)
    - Benefits of distributed hashing (robustness, load balancing, …)
- Clustered storage
- Highly available storage
- Built-in replication and geo-replication
- Self-healing
- Ability to re-balance data

GlusterFS: Architecture

- Four main concepts:
  - **Bricks**: storage units which consist of a server and directory path (i.e., server:/export)
    - Bricks are the nodes in Chord circle
    - Files are mapped to bricks calculating a hash
  - **Trusted Storage Pool**: trusted network of servers that will host storage resources
  - **Volumes**: collection of bricks with a common redundancy requirement
  - **Translators**: modules that are chained together to move data from point *a* to point *b*
    - Translator converts requests from users into requests for storage

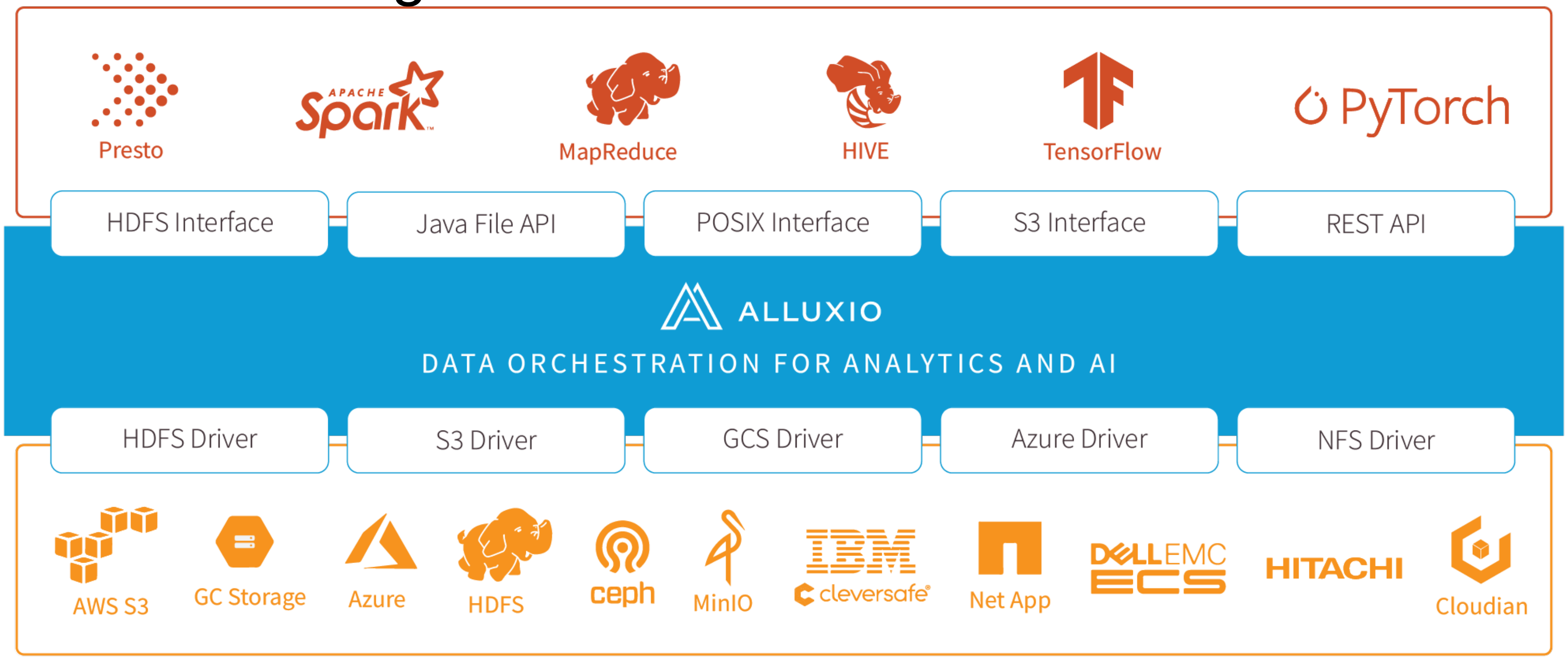
Other Distributed File Systems: Alluxio

- Motivations:
  - Write throughput is limited by disk and network bandwidths
  - Fault-tolerance by replicating data across the network (synchronous replication further slows down write operations)
  - Performance and cost trend: RAM is fast and cheaper
- Alluxio https://www.alluxio.org
  - Open-source, **in-memory** storage system
  - High-throughput reads and writes
  - Re-computation (lineage) based storage using memory aggressively
    - One copy of data in memory (fast)
    - Upon failure, re-compute data using lineage (fault tolerance)

H. Li, Alluxio: A Virtual Distributed File System, 2018

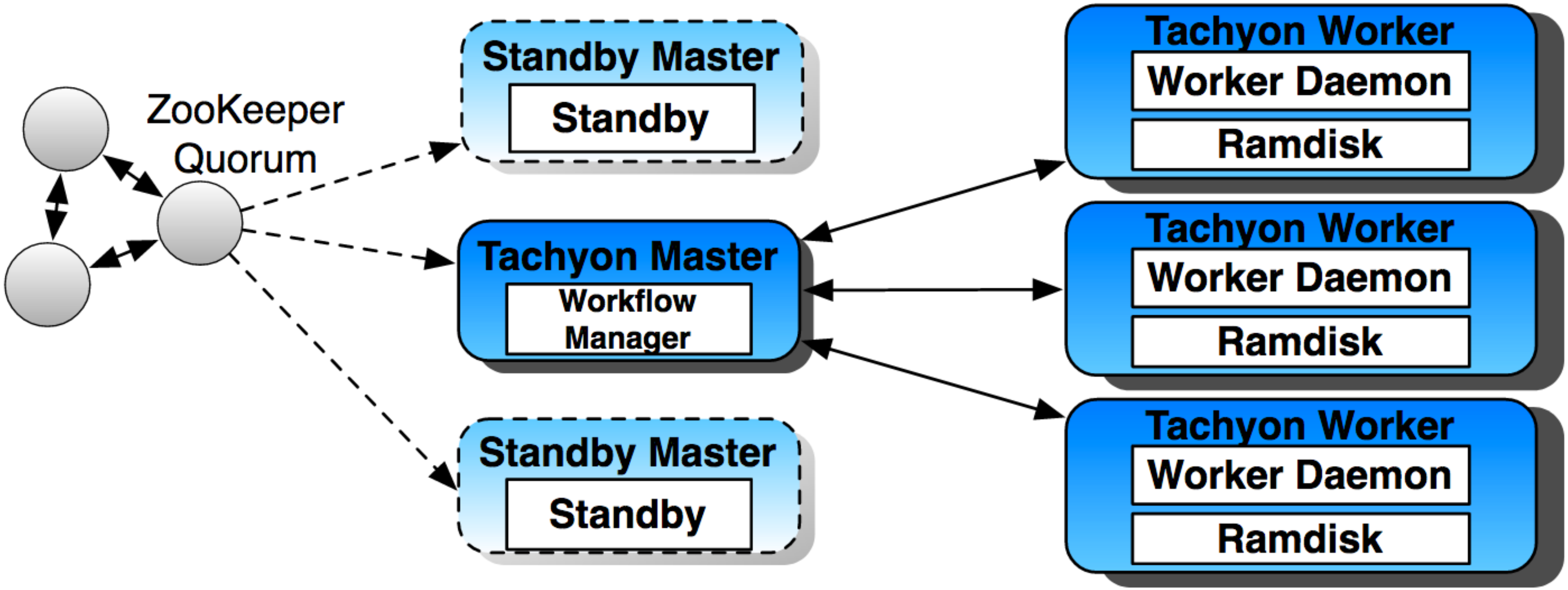
Alluxio

- Adds a layer between the processing layer and the storage layer
  – Big data processing frameworks (e.g., Spark, Flink, MapReduce, TensorFlow, …)
  – Persistence layer (e.g., HDFS, AWS S3, …)
- Goal: storage unification and abstraction



Alluxio: Architecture

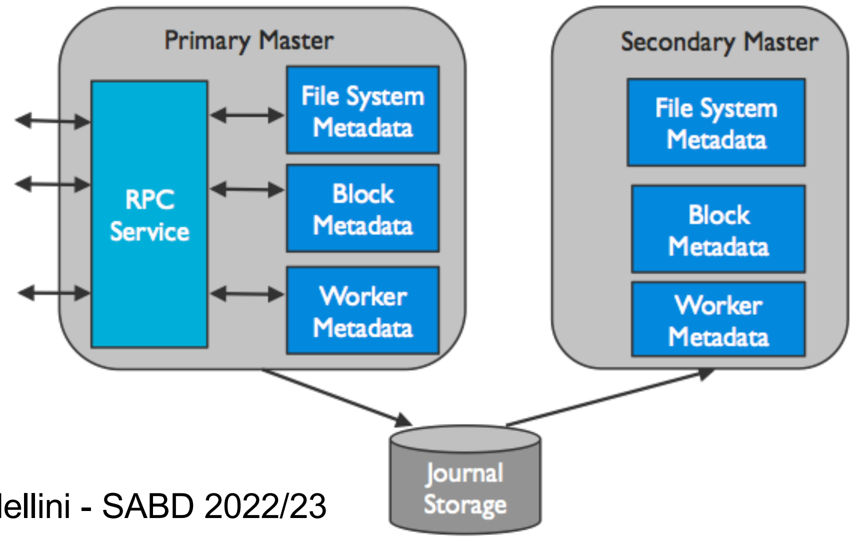
- Alluxio (formerly Tachyon) architecture
  - Master-worker architecture (like GFS, HDFS)
  - 3 components: replicated masters, multiple workers, clients
    - Passive standby approach to ensure master fault-tolerance



Alluxio: Architecture

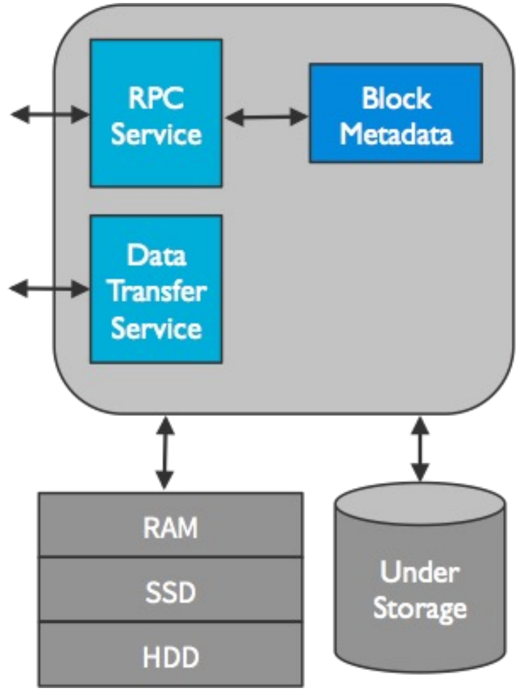
Master

- Stores metadata of storage system
- Only responds to client requests
- Tracks lineage information
  - **Lineage**: lost output is recovered by re-executing ops that created the output
- Computes checkpoint order
- Secondary master(s) for fault tolerance



Workers

- Manage local resources
- Periodically heartbeat to primary master
- RAM disk for storing memory-mapped files

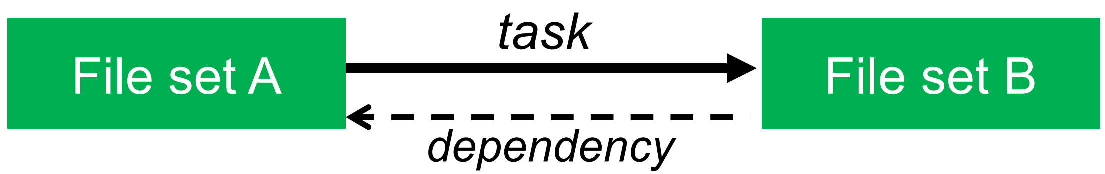


Alluxio: Lineage and persistence

Alluxio consists of two (logical) layers:

- **Lineage layer**: tracts the sequence of jobs that have created a particular data output
  - Data are immutable once written: only support for append operations
  - Frameworks using Alluxio *track* data dependencies and *recompute* them when a failure occurs
  - Java-like API for managing and accessing lineage information

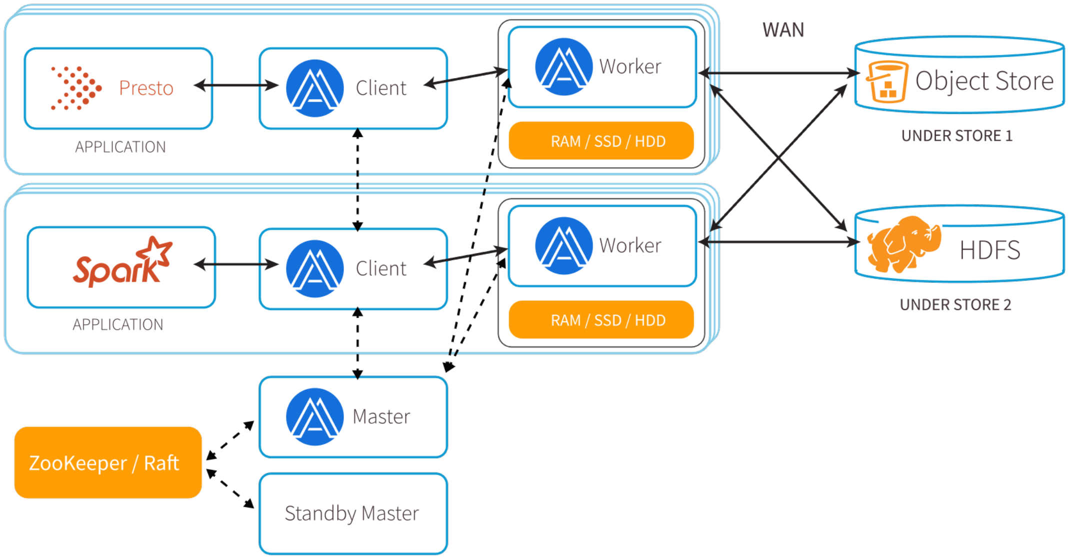
Task reads file set A and writes file set B



- **Persistence layer**: persists data onto storage, used to perform asynchronous checkpoints
  - Efficient checkpointing algorithm
    - Avoids checkpointing temporary files
    - Checkpoints hot files first (i.e., the most read files)
    - Bounds re-computation time

Alluxio: Evolution

- Evolving as data orchestration platform for analytics and AI
  - One of the fastest growing open source projects
- Goals:
  - Bring data closer to compute across clusters, regions, clouds, and countries
  - Make it easily accessible enabling applications to connect to numerous storage systems through a common interface



Data storage so far: Summing up

- Google File System and HDFS
  - Master/worker architecture
  - Decouples metadata from data
  - Single master (bottleneck): limits interactions and file system size
  - Designed for batch applications: 64/128MB chunk, no data caching
- GlusterFS
  - No centralized metadata server
  - Consistent hashing
- Alluxio
  - In-memory storage system, leverages on DFS
  - Master/worker architecture
  - No replication: tracks changes (lineage), recovers data using checkpoints and re-computations

References

- S. Ghemawat, H. Gobioff, and S.-T. Leung, The Google File System, *Proc. ACM SOSP '03*, 2003.
- D. Hildebrand and D. Hildebrand, Colossus under the hood: a peek into Google's scalable storage system, 2021.
- Video on Colossus: A peek behind the VM at the Google Storage infrastructure, 2020
- Shafer et al., The Hadoop Distributed Filesystem: Balancing Portability and Performance, *Proc. ISPASS '10, 2010.*
- H. Li, Alluxio: A Virtual Distributed File System, PhD Thesis, Berkeley Univ., 2018.
- H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica, Tachyon: Reliable, Memory Speed Storage for Cluster Computing Frameworks, *Proc. ACM SoCC '14*, 2014.