Big Data Systems

Djellel Difallah

Spring 2023

Lecture 8 – Distributed File Systems

GFS and HDFS

So far …

- We looked at **Data Management** techniques
  - RDBMSs
    - Relational Schema
  - NoSQL
    - Mostly simple operations such as put/get

- Now we look at very large scale **Data Processing** techniques
  - Massive amounts of data
  - Read mostly workloads (OLAP)
  - Efficiency (data processed per second) over consistency
  - **Separating storage and computation**

Outline

- Introduction
- The Google File System
  - Motivation
  - Design Principles
  - Architecture and Operations
- The Hadoop Distributed File System
  - Architecture
  - Features

Introduction

- Your job is to *process* 1TB of data
  - You: *Sure!*
  - But first, you need to read it.

```
with open(bigFile) as f:
    line = f.readlines()
    …
```

  - You: ☹ It’s taking forever.

- HDD (~ 150 MB/s) ~ 2.5 hours
- SSD (~ 500 MB/s) ~ 1 hour
- NVMe: couple of minutes.
- Multithreading will not help (*especially not for HDD*)

Introduction

- That was an exercise:
  - 10TB, 100TB, 1PB ??
- The data storage medium is clearly a bottleneck
- We need a way to increase the throughput
  - Yes: use multiple disks
  - Not scalable on a single machine (Bus limitations)
- N Machines with K disks
  - Cost effective when using cheap machines.
  - Hard to program

File System

- Metadata = information about the file
  – Includes name, access permissions, timestamps, file size, & locations of data blocks
  – ***inodes*** (**i**ndex **nodes**) are data structures containing meta information about the file
- Data = actual file contents
- A file system stores all the **bits and pieces** of a given file on the same device.
- For example: Linux directories are just files that contain lists of names and inodes

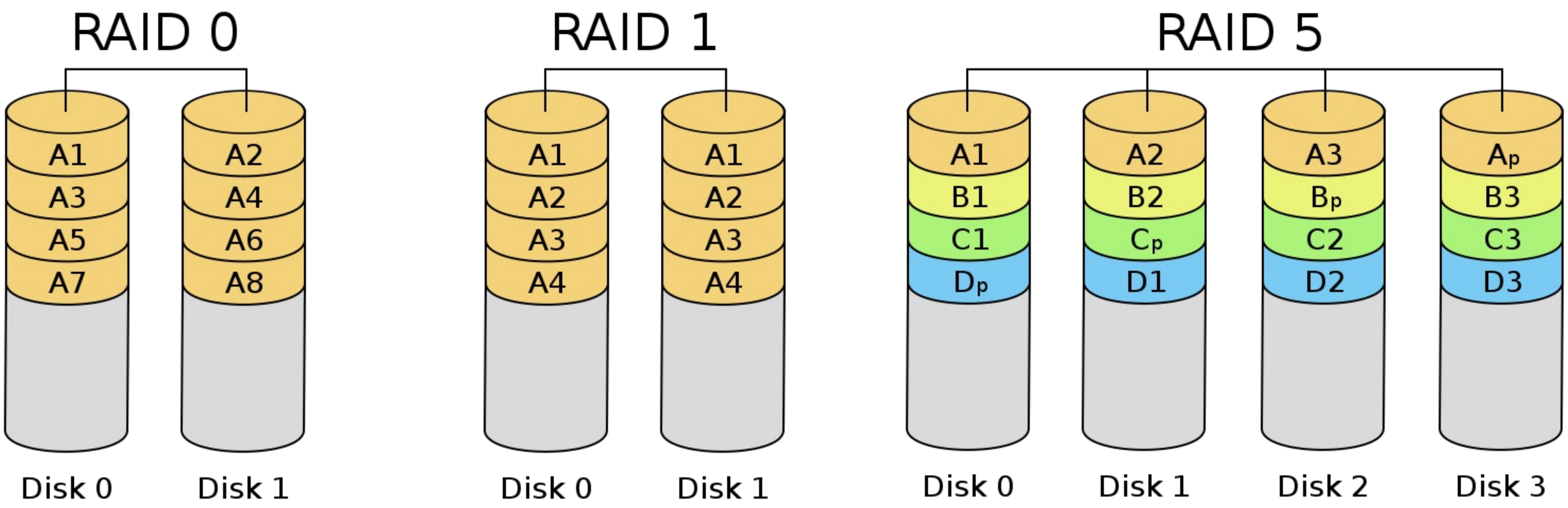
RAID: Redundant Array of Independent Disks

- RAID 0 (Striping):
  - Data is **striped** across multiple disks, which improves read and write performance.
  - RAID 0 does not provide any redundancy or fault tolerance: data loss will occur if a single disk fails.

- RAID 1 (Mirroring):
  - Data is **duplicated** across two or more disks, providing fault tolerance through redundancy.
  - RAID 1 improves read performance but has a higher cost due to the need for additional storage.

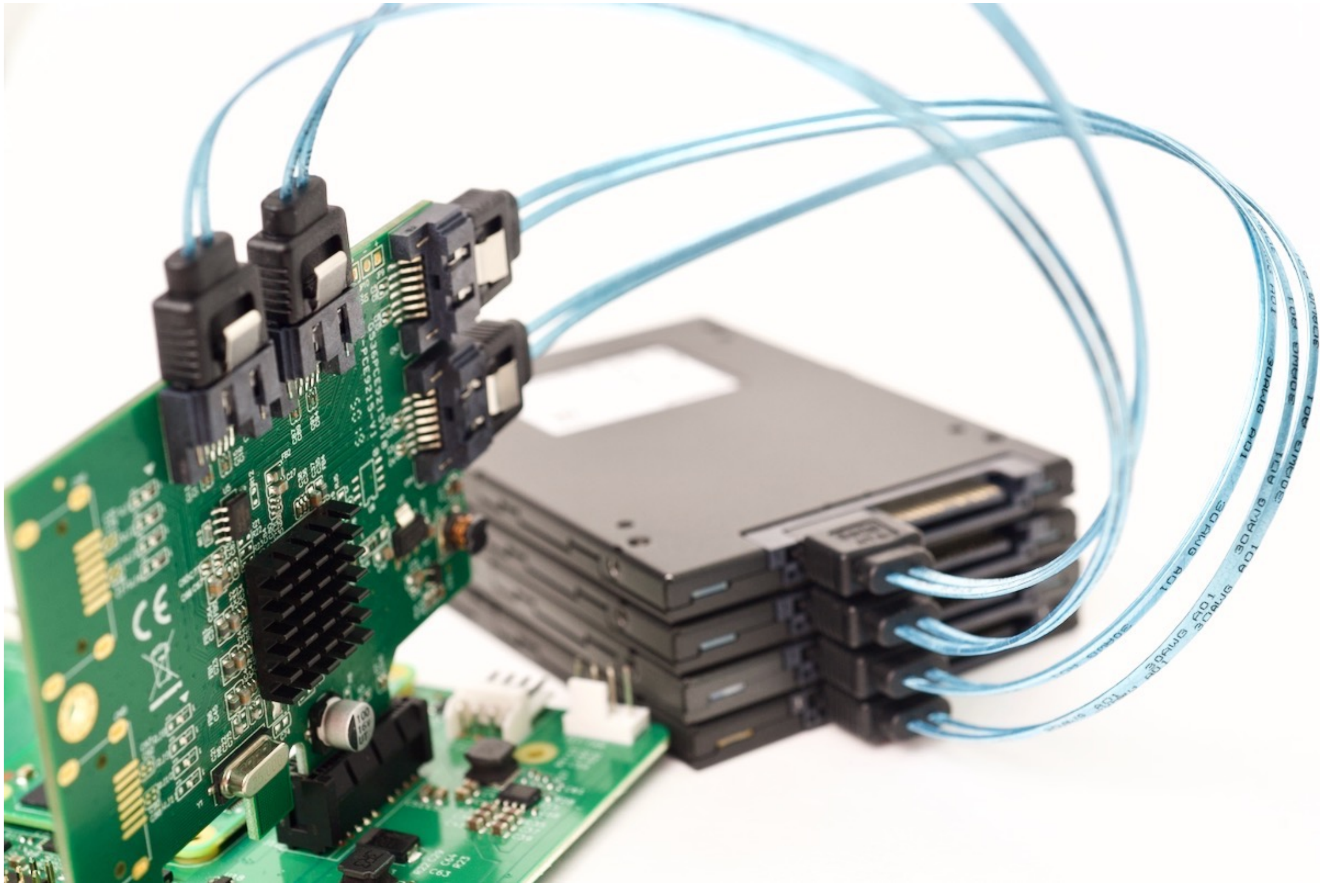
- RAID 5 (Striping with Parity):
  - Data and **parity information** are striped across three or more disks.
    - Parity is an error detection and correction techniques.
  - RAID 5 provides fault tolerance and improves read performance but requires extra storage for parity data.
  - A minimum of 3 disks, and can handle 1 disk failure.

RAID Examples



Images from Wikicommons

RAID Controller



Client Server File Systems

- Network Attached Storage (NAS)
  - Shared Disk Architecture: Single point of failure.
  - Point of congestion
  - Industry solution:
    - Replication, recovery and client caching

Parallel File Systems

- Distributed Architecture
  - **Multiple servers** work together to store, manage, and access data
  - Enhance performance

- Data Chunking (Striping)
  - Data is divided into smaller **chunks** (stripes) and distributed across multiple storage devices
  - **File data can be on multiple servers**
  - Enables concurrent reading and writing for improved performance

- Metadata Management
  - **Metadata can be stored and managed separately from data (stored on a different node)**
  - Allows for efficient searching, indexing, and data organization

Parallel File Systems (cont.)

- Scalability
  - Designed to handle large-scale storage and computing environments
  - Can easily expand storage capacity and performance as needed
- Fault Tolerance & Data Redundancy
  - Built-in mechanisms to protect against data loss and hardware failures
  - Examples: replication, erasure coding, and RAID techniques
- Load Balancing
  - Ensures optimal distribution of data and workload across available resources
  - Minimizes bottlenecks and maximizes performance

Consideration

- Component (Disk) failure is the norm
  - File system = thousands of storage machines
  - Some % not working at any given time
- Most files are large (not small).
  - Block size and I/O operations
- The programming models changes
  - Rethink existing solutions

Programming Model

- Shared Memory
  - Threads
  - Shared data structures with locks etc.
  - Easy to reason about.
    - The memory becomes a limitation

- Message Passing (many techniques here)
  - Master-slave
  - Producer-consumer
  - Very difficult to program correctly and to reason about.
    - Every new task will require reengineering

The Google File System

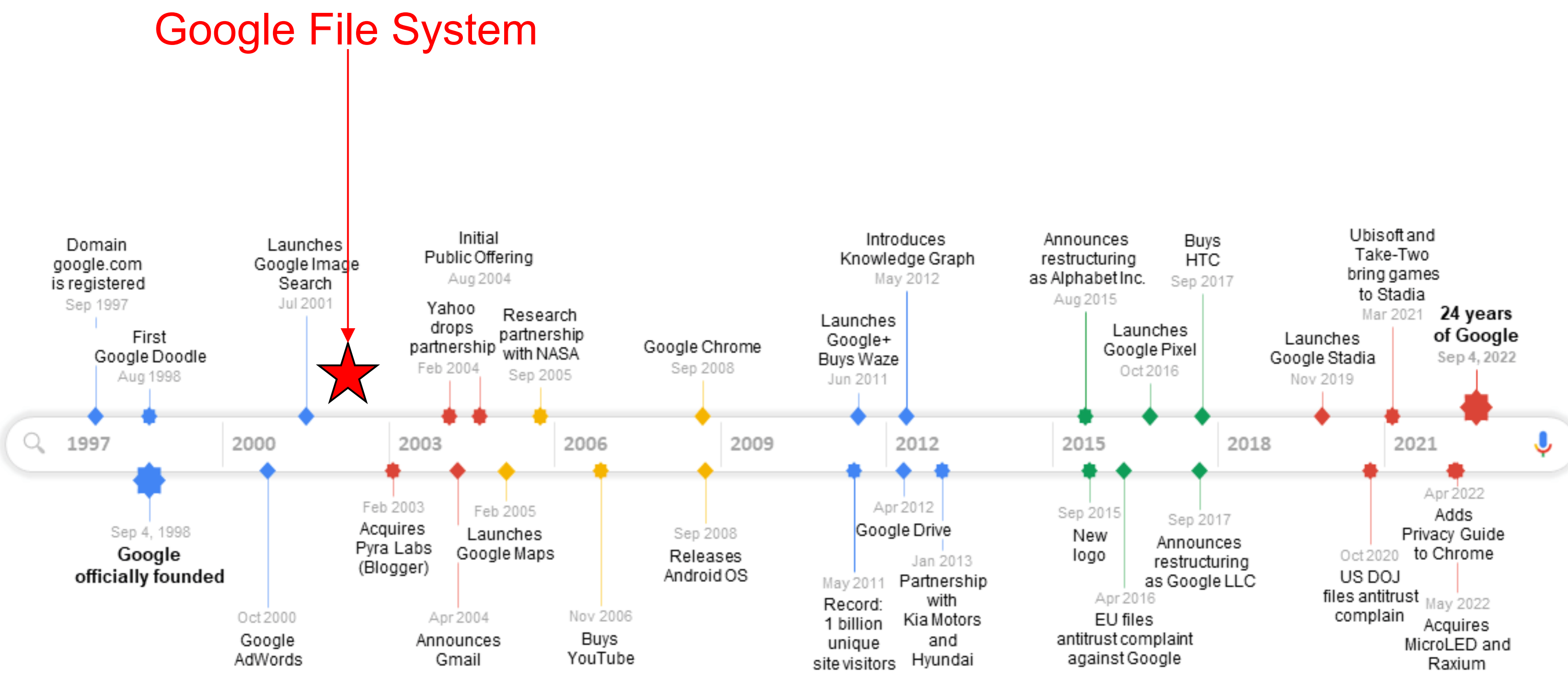
GFS

Introduction

- New parallel file system to meet the needs of Google’s large-scale data processing challges
- *“The Google File System” (2003) by Ghemawat et al.*
  - The paper describes the design and implementation of the Google File System a scalable, distributed file system.

From: https://www.officetimeline.com/blog/google-history-timeline

Google History Context



Why are we learning about Google File System?

- Covers important (and timeless) themes in distributed systems:
  - parallel performance, fault tolerance, replication, consistency
- Practical: a successful real-world implementation and use case.
  - MapReduce, BigTable built on top of GFS
- Large scale
- Demonstrate the use of weak consistency
- Single master architecture
- Simple but thorough system paper to read

Motivation

- Commodity Hardware
  - Error is frequent in large setups
- Large Files
  - Multiple 100Mb+ files
- File operations
  - Read and Append
  - Mostly sequential reads

Design Principles

- Fault tolerance:
  - GFS is designed to handle faults gracefully, including hardware failures and network issues.
  - It achieves this through replication, checksums, and automatic recovery mechanisms.

- Data replication:
  - To ensure durability and availability, the system replicates each chunk across multiple node (chunk servers), default is three replicas.

- High throughput:
  - GFS is optimized for high sustained throughput over low latency, workloads typically involve large data processing tasks.

Design Principles

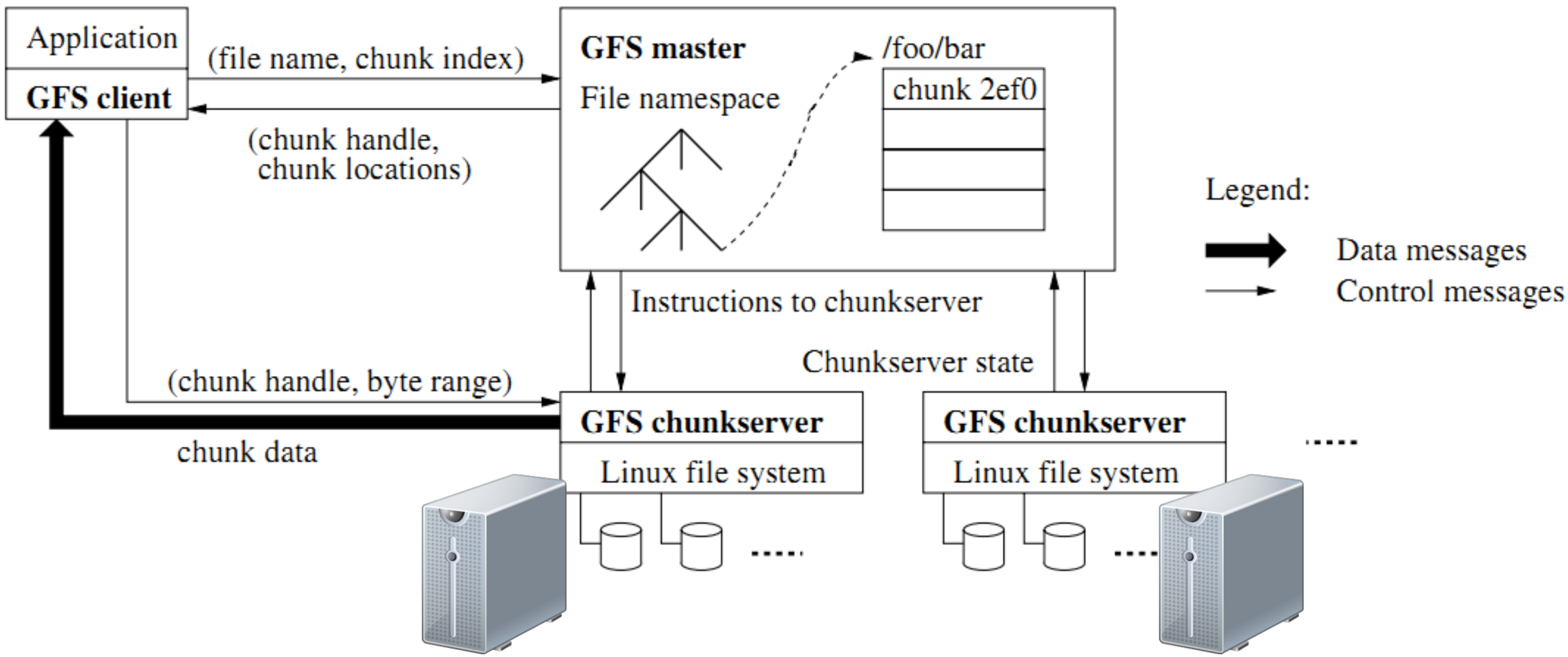
- Consistency and atomicity:
  - GFS provides a consistent view of the file system by using a combination of techniques like write serializability, chunk versioning, and atomic record appends.
- Chunk-based storage:
  - GFS stores files as fixed-size chunks and uses a single master server to manage the metadata associated with these chunks.
  - The actual chunk data is stored on chunk servers distributed across the cluster.

Design Principles

- Scalability:
  - The architecture is designed to support massive scale, allowing it to handle large numbers of files and a vast amount of storage.
  - Runs on commodity hardware

- Master-Slave Architecture
  - A single node (master server) is responsible for managing the file system metadata, such as the namespace, access control information, and chunk location information.
  - The master server also handles garbage collection and chunk migration to balance load and recover from failures
  - Single point of failure

GFS Architecture



Google File System Architecture
(Ghemawat et al., SOSP 2003)

Master

- Stores the full file system metadata
  - Namespace (i.e., file system tree)
  - Access control info
  - Filename to chunks mappings
  - Current locations of chunks
- Manages
  - Chunk leases (locks)
  - Garbage collection (freeing unused chunks)
  - Chunk migration (copying/moving chunks)
- Fault tolerance
  - Operation log replicated on multiple machines
  - New master can be started if the master fails by replaying the operation log.
- Periodically communicates with all chunkservers – Via heartbeat messages to get state and send commands
  - HeartBeat is a short message:
    - Ensure that server is alive
    - Collect statistics
- Advantage
  - Given the setup (read mostly) this architectural choice simplifies the system immensely.

File Chunks and Chunkserver

- Chunk size
  - 64 MB (default)
  - DBMS to Big Data: From page size to chunk size
- Chunk Handle:
  - A global unique identifier assigned by the master
- Chunkservers
  - Stores chunks on local disks as Linux files
  - Stores a 32-bit checksum with each chunk in memory and logged to disk (detect data corruption)

Metadata

- All metadata is stored in the master system memory for fast operations and efficient background scans
  - Background scans are processes used for maintenance, e.g., chunk garbage collection.
- Types of metadata:
  - Namespace: Hierarchical information about files and directories
  - Filename: an array of chunk handlers
  - Chunk handler: list of chunk servers, primary chunk server, chunk version number
- First 2 types of metadata are persisted by logging mutations to operation log (oplog) on the master's local disk
  - Operation log replicated periodically to a remote machine (shadow master)
- Master does not store chunk location information persistently
  - Chunkservers provide chunk location and primary status information on master startup or when joining the cluster

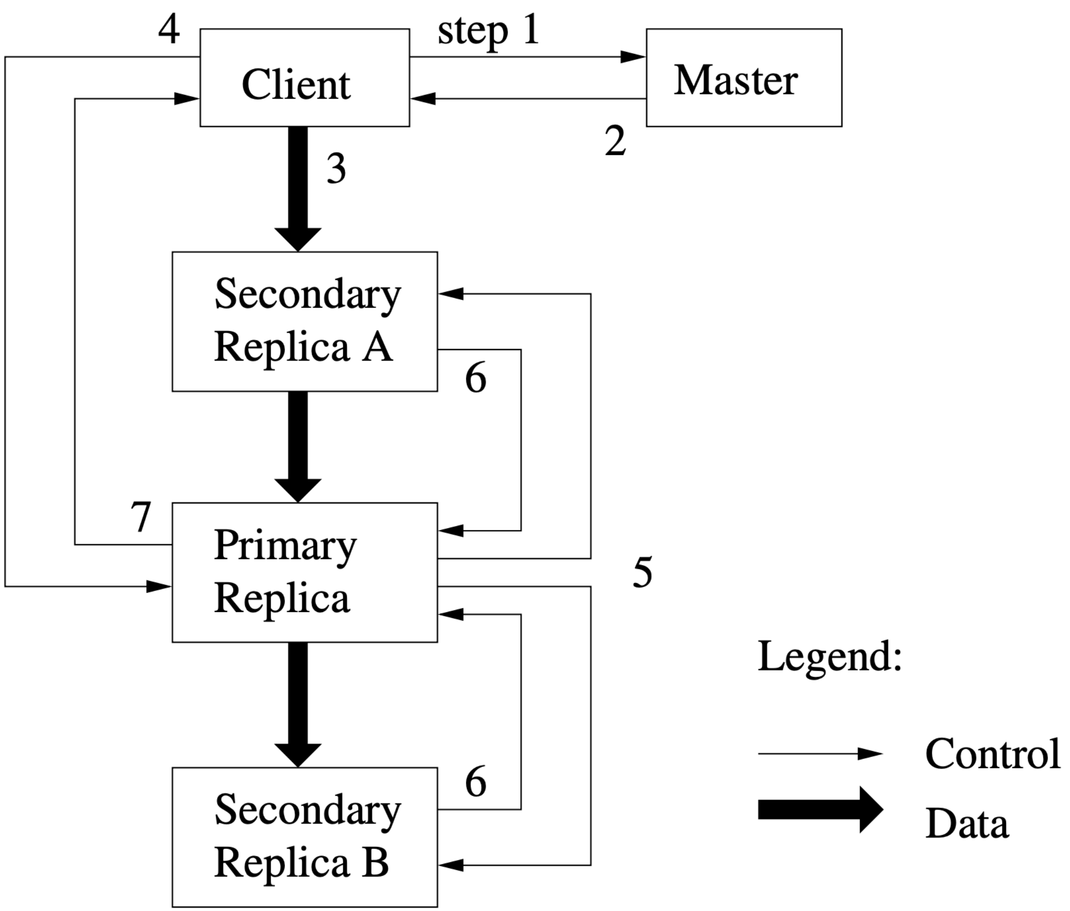
Consistency

- Atomic operations:
  - Namespace mutations are atomic and managed by the master
  - Namespace locking ensures atomicity
  - Master's operation log defines a global order
  - Record append operation is atomic

- Chunk replication:
  - Ensures data consistency after mutations
  - Applies mutations in order by designating a primary replica
  - Uses chunk version numbers to detect stale replicas
    - Managed by the master
  - If stale replicas are not used for mutations or provided to clients; garbage collected when possible

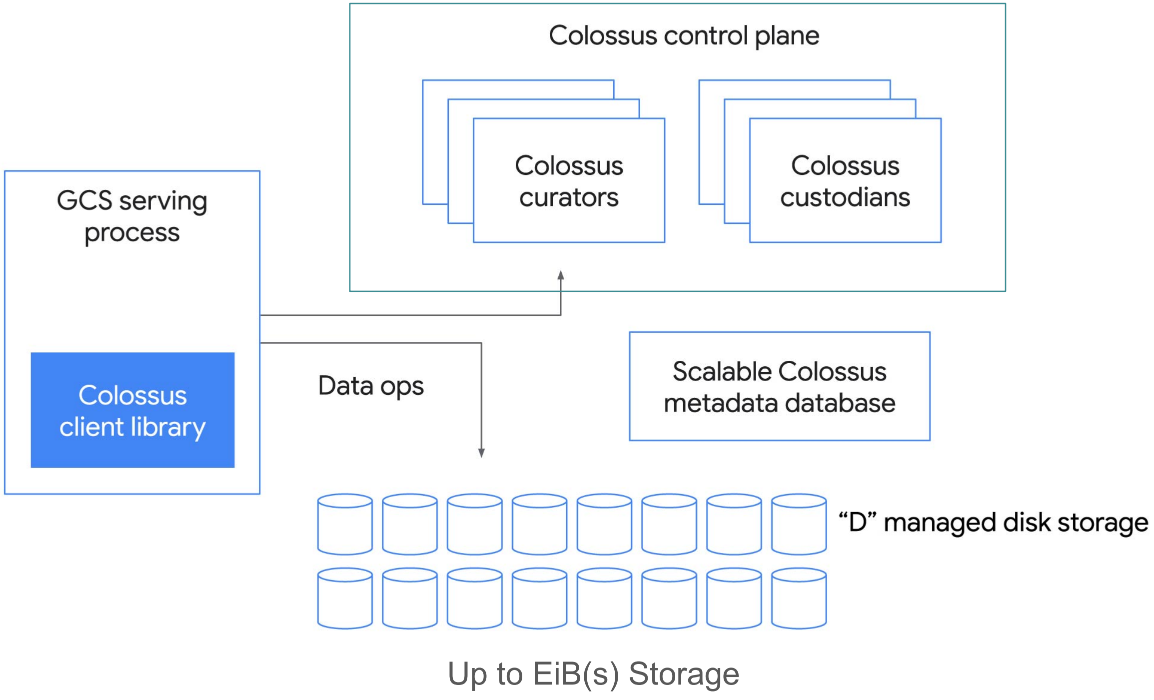
Write Data Flow

1. Lease acquisition
   - Chunk Version Increment
2. Primary selection
3. Data Forwarding
4. Data buffering and Ack
5. Write Command
   - Primary Mutation
   - Secondary Mutation
6. Primary Ack
7. Client Ack



Google next generation GFS: Colossus

- Google Colossus: Better scalability and availability
  - Blog post: ["A peek behind colossus googles file system"]
  - Curator: Distributed metadata (NOSQL)
  - Custodian: storage management (space balancing, raid, etc.)



Apache Hadoop Distributed File System

HDFS

Apache Hadoop

- A Platform (Ecosystem) for Big Data processing
  - Started by Yahoo! and was inspired by the following papers from Google
    - MapReduce

      Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." Communications of the ACM 51.1 (2008): 107-113.
    - GFS

      Ghemawat, Sanjay, Howard Gobioff, and Shun-Tak Leung. "The Google file system." ACM SIGOPS operating systems review. Vol. 37. No. 5. ACM, 2003.
- Written *mostly* in Java
- Main core components
  - Hadoop Distributed File System (HDFS)
  - A Map Reduce implementation
  - Other management services such as *YARN*

HDFS Motivation

- Lots of commodity hardware
  - Cheap PCs
- High failure rates
  - Commodity components failure is much higher than server grade.
- Write-once read many times data
  - Logs, archives etc.
  - Processing is mostly read oriented
- Large streaming reads over random access
  - High sustained throughput over low latency

HDFS Features

- A stand-alone general purpose Distributed File System
  - used as a part of a Hadoop cluster
- Highly scalable
  - Many production clusters with Petabytes of data
- Fault tolerant
- Offers a simplified data access
  - The users see their file (data) as if they were on a single machine
  - default configuration is well suited for many installations
  - Tuning is required for large deployments
- Linux like security, limited to file level permission

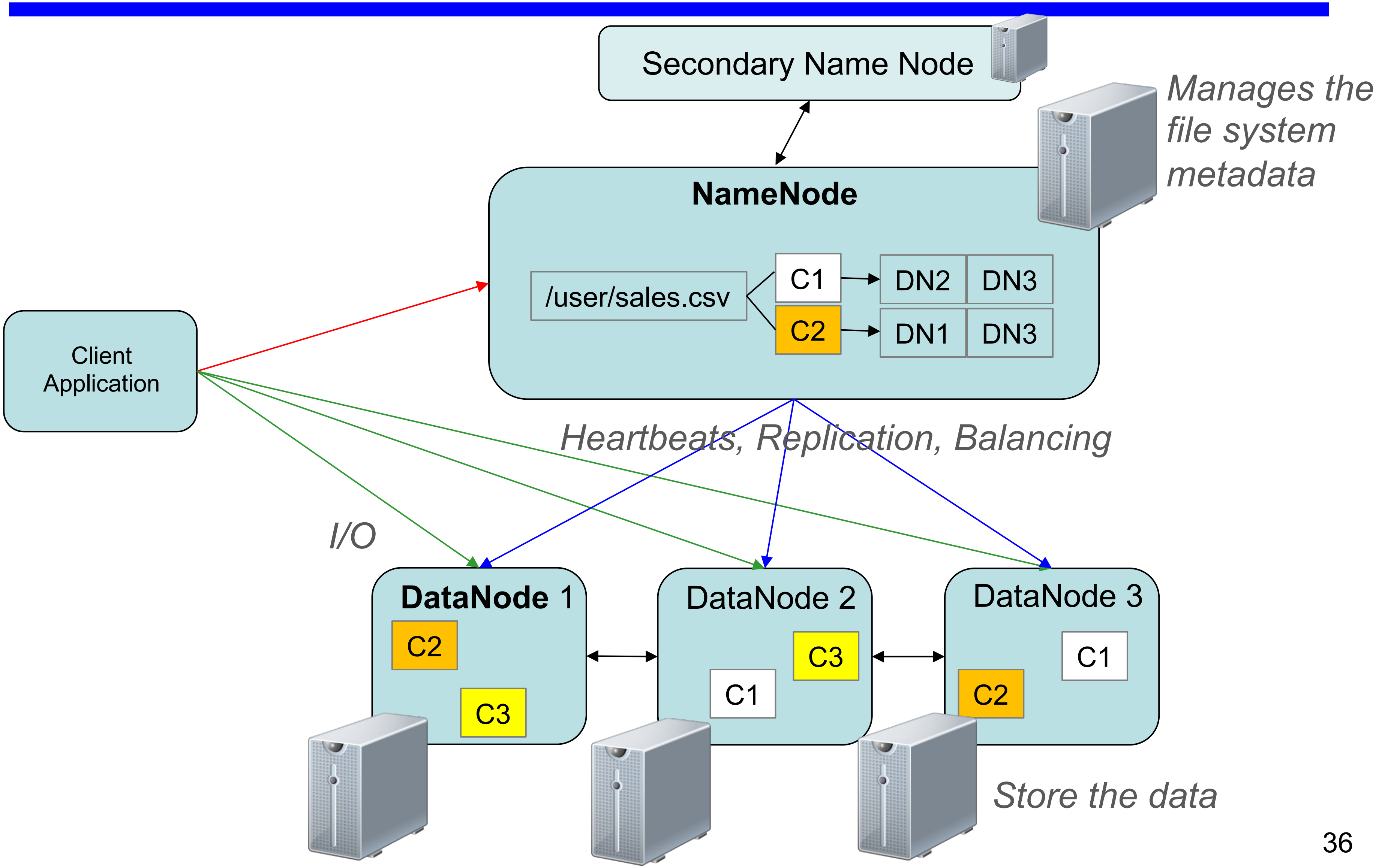
HDFS Basic Operations

- When an input file is added to HDFS
  - File is split into smaller blocks of fixed size
  - Each block is replicated to multiple hosts (machines)
  - Each replicated block is stored on a different host
  - Replication level is configurable
    - Default is 3
  - If a host crashes or is decommissioned
    - All blocks are replicated to a new host
  - In case a new host is added
    - Blocks will be rebalanced (avoid data skew)

Blocks

- Single unit of storage
- Transparent to the end-use
- Block is traditionally 64/128/256 MB
- Block size if fixed whether you store a file of 64MB or less.
- It is large enough to force a tradeoff between a seek and sequential disk read.
  - Time to read a block = seek time + transfer time
  - We want to minimize $\frac{Seek\ Time}{Transfer\ Time}$

HDFS Architecture



Name Node

- Managing the file system namespace:
  - Holds file/directory structure, metadata, file-to-block mapping, access permissions, etc.
- Coordinating file operations:
  - Directs clients to datanodes for reads and writes
  - No data is moved through the namenode
- Maintaining overall health:
  - Periodic communication with the datanodes
  - Block re-replication and rebalancing
  - Garbage collection
- Executes file system namespace operations like opening, closing, and renaming files and directories

Data Node

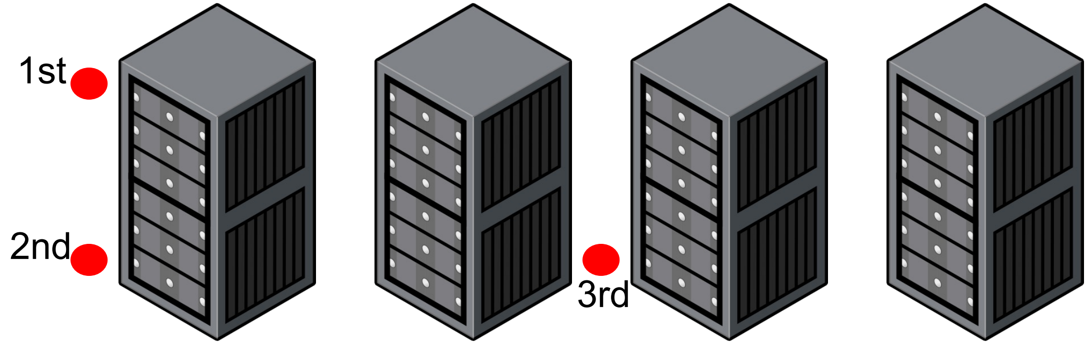
- Actual storage and management of data block on a single host
- Provides clients with access to data
- Perform block creation, deletion, *and* replication upon instruction from the NameNode
- Sends heartbeat (every 3 seconds) and a blockreport (6hours) to NameNode

Secondary Name Node

- The name node keep the current state of the HDFS in an image file (`fsimage`) which it reads during the startup
- All modifications are written to a log (`edits`)
- NN merges fsimage and edits during startup
- The Secondary NameNode is not a failover NN
  - Role: check pointing
  - It merges periodically fsimage and edits
  - Every 1 Hour or 1 Million transactions by default (configurable)

Replication

- Maximize the reliability, availability and bandwidth
  - Opportunistic execution
- Replicas spread across machines and racks
- Consider geo replication for disaster recovery
- The master determines replica placement. In HDFS:
  - 1st replica on local node (w.r.t writing process) or random node
  - One replicas is usually placed on the same rack as the node writing to the file cross-rack network I/O is reduced
  - One replica on a different node in a different rack



Rebalancer

- HDFS
  - May produce non-uniform distribution of blocks across hosts
    - e.g., new machines added or removed
  - Does not consider different hardware generations
  - Does not consider current hardware utilization
  - Operates on block level not on file level
- HDFS architecture is compatible with data rebalancing schemes
- The system administrator can perform periodic cluster wide rebalancing of the blocks.

```
hadoop balancer <threshold>
```

  - The threshold (X%) sets a target for the balance state (default 10)
  - **Each** node's utilization ratio is within X% of the overall cluster utilization
    - Utilization Ratio = $\frac{utilizedSpace}{TotalCapacity}$
  - Smaller threshold leads to more balanced distribution
    - Might take a lot of time, esp. when concurrent utilization

Block Placement Policy

- HDFS allows the implementation of custom block placement policies
- Alternative Placement Policies, e.g.,:
  - Use lower utilization servers first
  - Operate on file level
  - Assign weight to DataNodes
  - Move more blocks to newer generation hardware