# 龙星计划课程: 文件系统和分布式数据管理系统

# Building File Systems and Distributed Data Management Systems for Performance and Reliability

## Lecture 2: Distributed File Systems

# Filesystems Overview

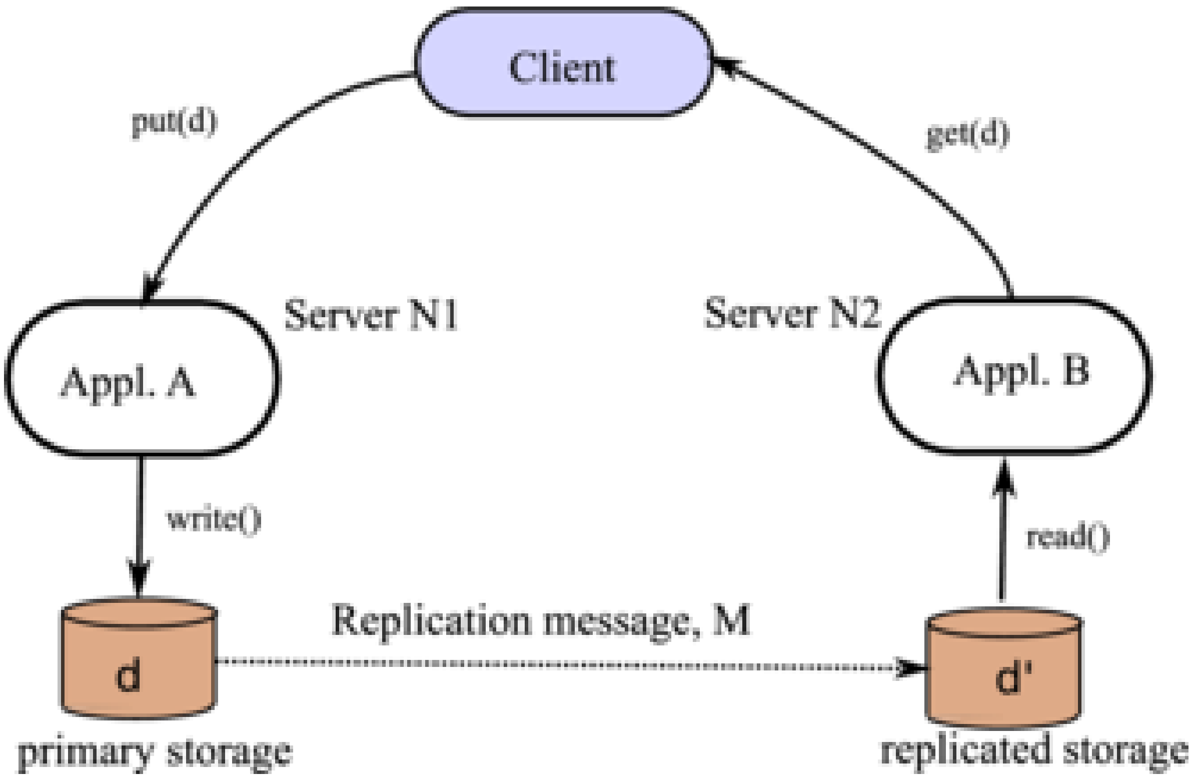
- Permanently stores data
- Usually layered on top of a lower-level physical storage medium
- Divided into logical units called “files”
  - Addressable by a *filename* (“foo.txt”)
  - Usually supports hierarchical nesting (directories)
- A file *path* = relative (or absolute) directory + file name
  - /dir1/dir2/foo.txt

# Distributed Filesystems

- Support access to files on remote servers
- ***Must*** support concurrency
  - Make varying guarantees about locking, who “wins” with concurrent writes, etc...
  - Must gracefully handle dropped connections
- Can offer support for replication and local caching
- Different implementations sit in different places on complexity/feature scale

# Putting Everything Together: the CAP Theorem

No distributed system can simultaneously provide all three of the following properties: **C**onsistency (all nodes see the same data at the same time), **A**vailability (node failures do not prevent survivors from continuing to operate), and **P**artition tolerance (the system continues to operate despite arbitrary message loss).



# Google's GFS: Motivation

- Google needed a good distributed file system
  - Redundant storage of massive amounts of data on **cheap** and **unreliable** computers
- Why not use an existing file system?
  - Google's problems are different from anyone else's
  - Different workload and design priorities
  - **GFS** is designed for **Google apps** and **workloads**
  - **Google apps** are designed for **GFS**

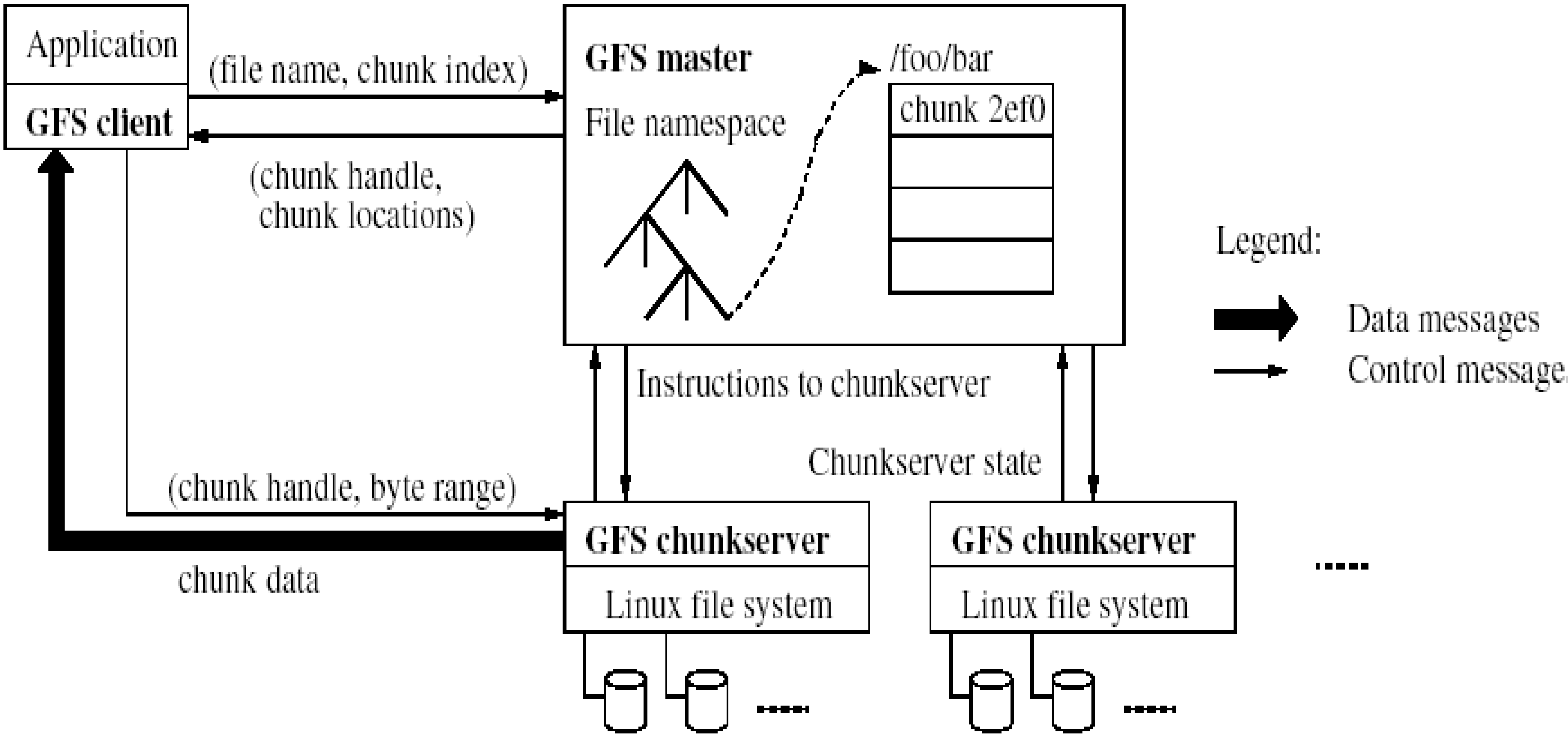
# Assumptions

- **High** component failure rates
  - Inexpensive commodity components fail all the time
- "Modest" number of HUGE files
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- Files are write-once, mostly appended to
  - Perhaps concurrently
- Large streaming reads
- High sustained throughput favored over low latency

# GFS Design Decisions

- Files stored as chunks
  - Fixed size (64MB)
- Reliability through replication
  - Each chunk replicated across 3+ *chunkservers*
- **Single** master to coordinate access, keep metadata
- Simple centralized management
- ***No*** data caching
  - Little benefit due to large data sets, streaming reads
- Familiar interface, but customize the API
  - Simplify the problem; focus on Google apps
  - Add ***snapshot*** and ***record append*** operations

# GFS Architecture




Each chunk is identified by an immutable and globally unique 64 bit *chunk handle* assigned by the master at the time of chunk creation.

*…Can anyone see a potential weakness in this design?*

# Single master

- Problem:
  - **Single** point of failure
  - Scalability bottleneck
- GFS solutions:
  - *Shadow* masters
  - Minimize master involvement
    - ✓ **never** move data through it, use only for metadata and cache metadata at clients
    - ✓ large chunk size
    - ✓ master delegates authority to primary replicas in data mutations (chunk leases)
- Simple, and good enough for Google's concerns

# Metadata

- Global metadata is stored on the master
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
- All in memory (64 bytes / chunk)
  - Fast
  - Easily accessible
- Master has an ***operation log*** for persistent logging of critical metadata updates
  - Persistent on local disk
  - Replicated
  - Checkpoints for faster recovery

# Master's Responsibilities

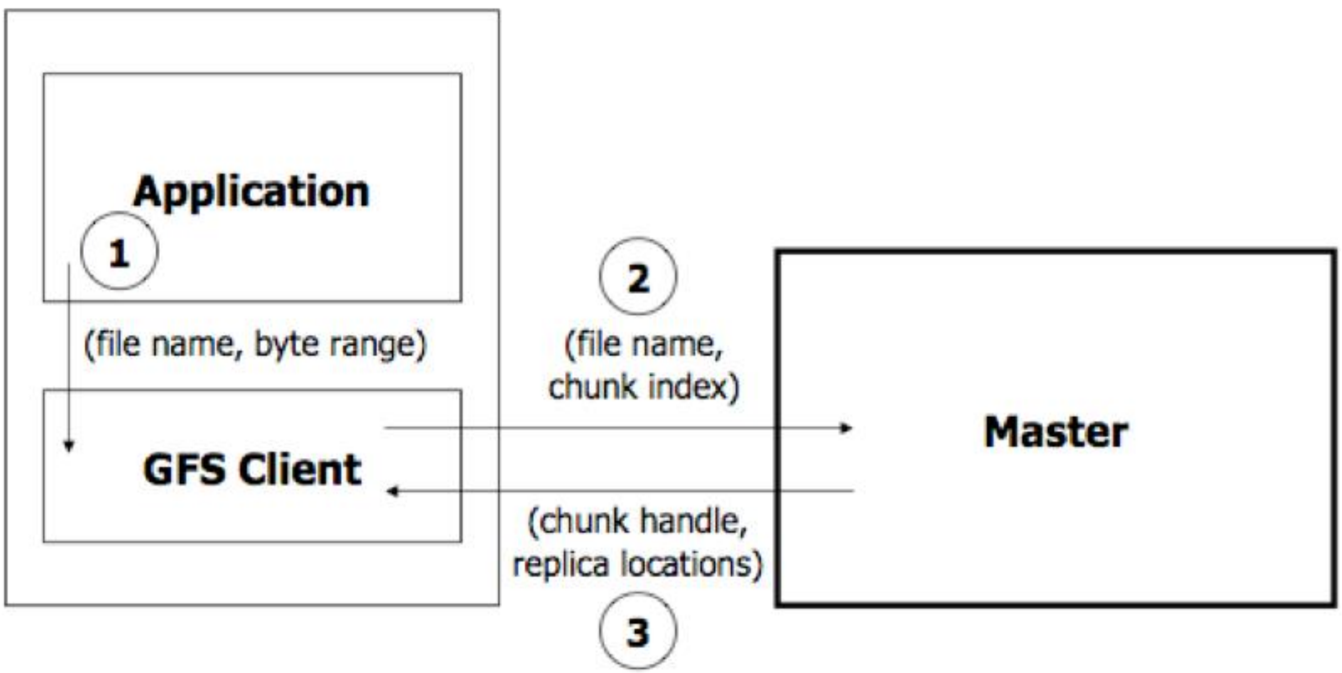
- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
  - give instructions, collect state, track cluster health
- Chunk creation, re-replication, rebalancing
  - balance space utilization and access speed
  - spread replicas across racks to reduce correlated failures
  - re-replicate data if redundancy falls below threshold
  - rebalance data to smooth out storage and request load
- 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

# Mutations

- Mutation = write or record append
  - Must be done for all replicas
- Goal: ***minimize*** master involvement
- Lease mechanism:
  - Master picks one replica as primary of a chunk; gives it a "lease" for mutations
- Data flow decoupled from control flow

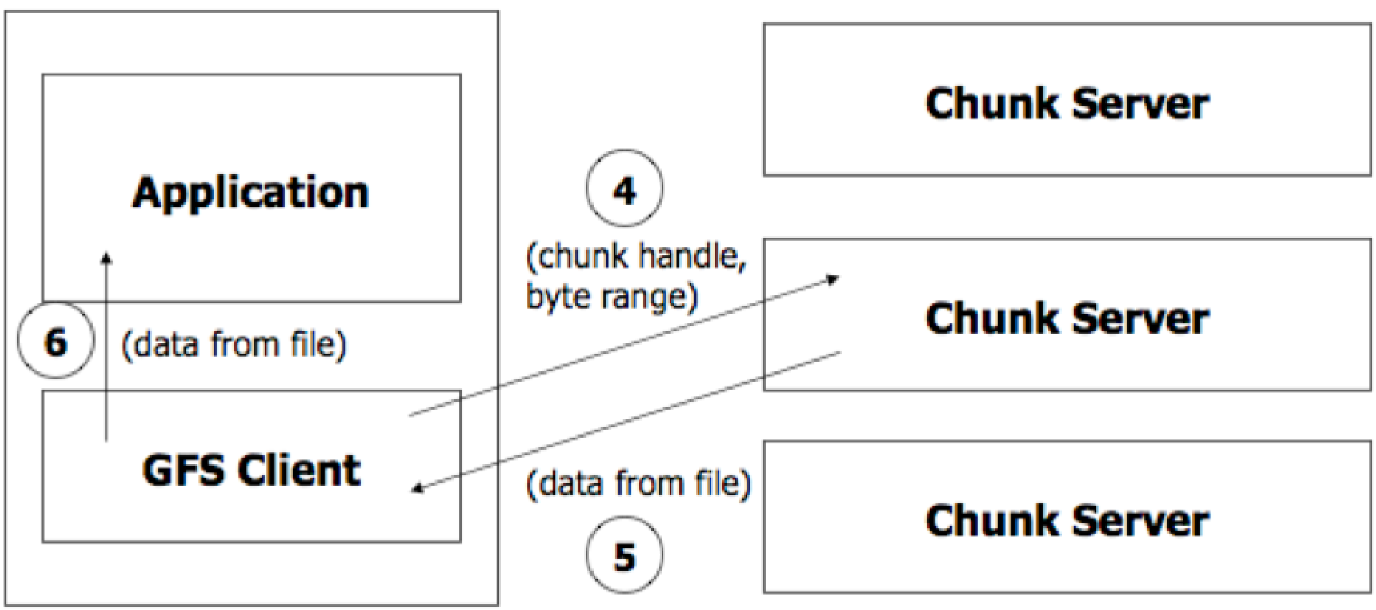
# Read Algorithm

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



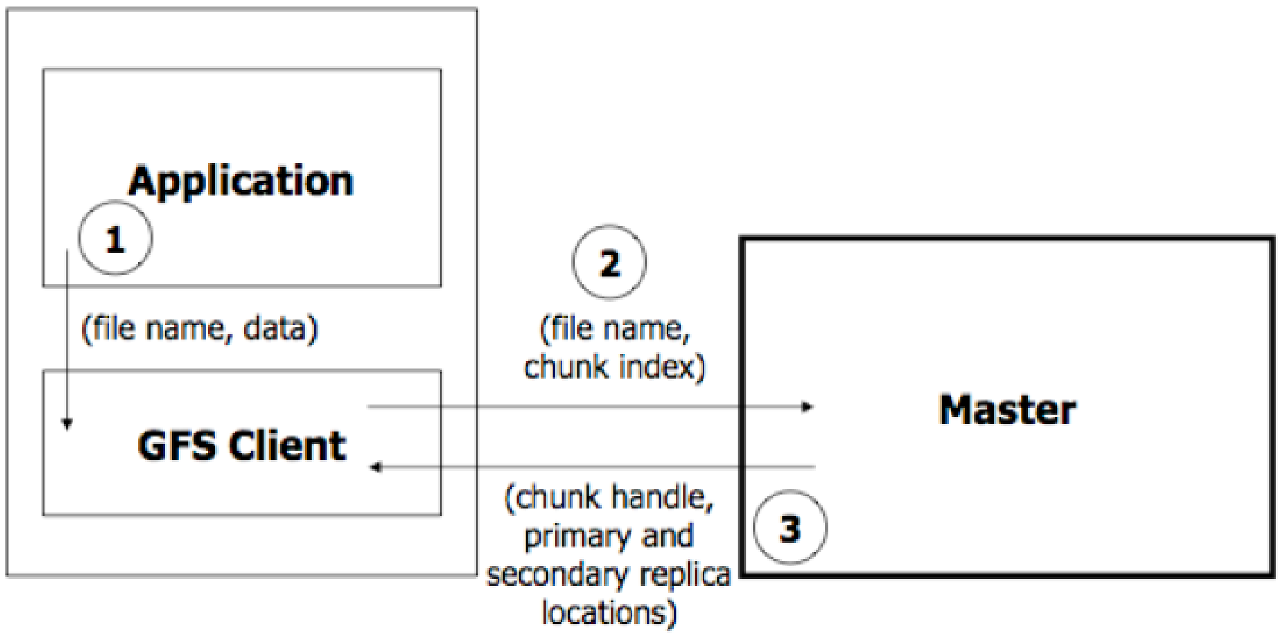
# Read Algorithm

4. Client picks a location and sends the request
5. Chunkserver sends requested data to the client
6. Client forwards the data to the application



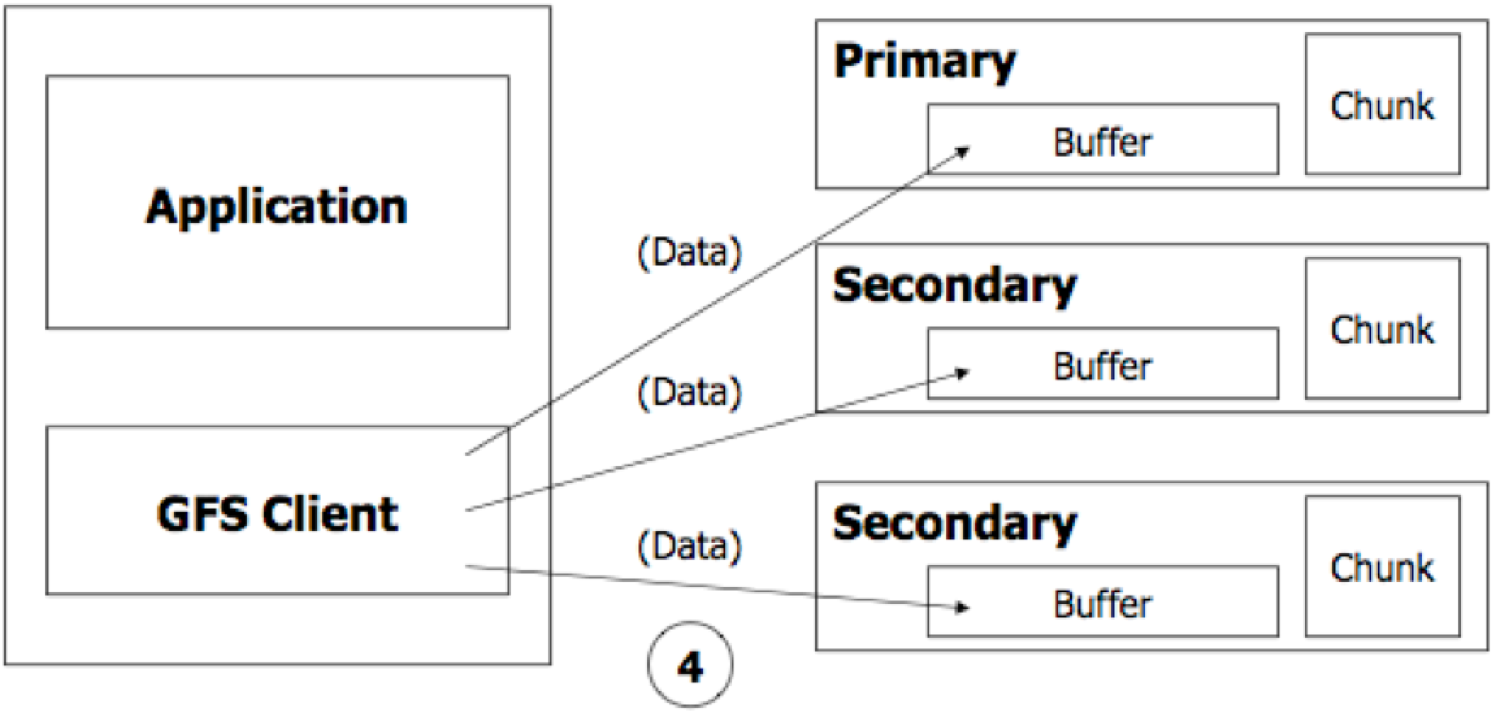
# Write Algorithm

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



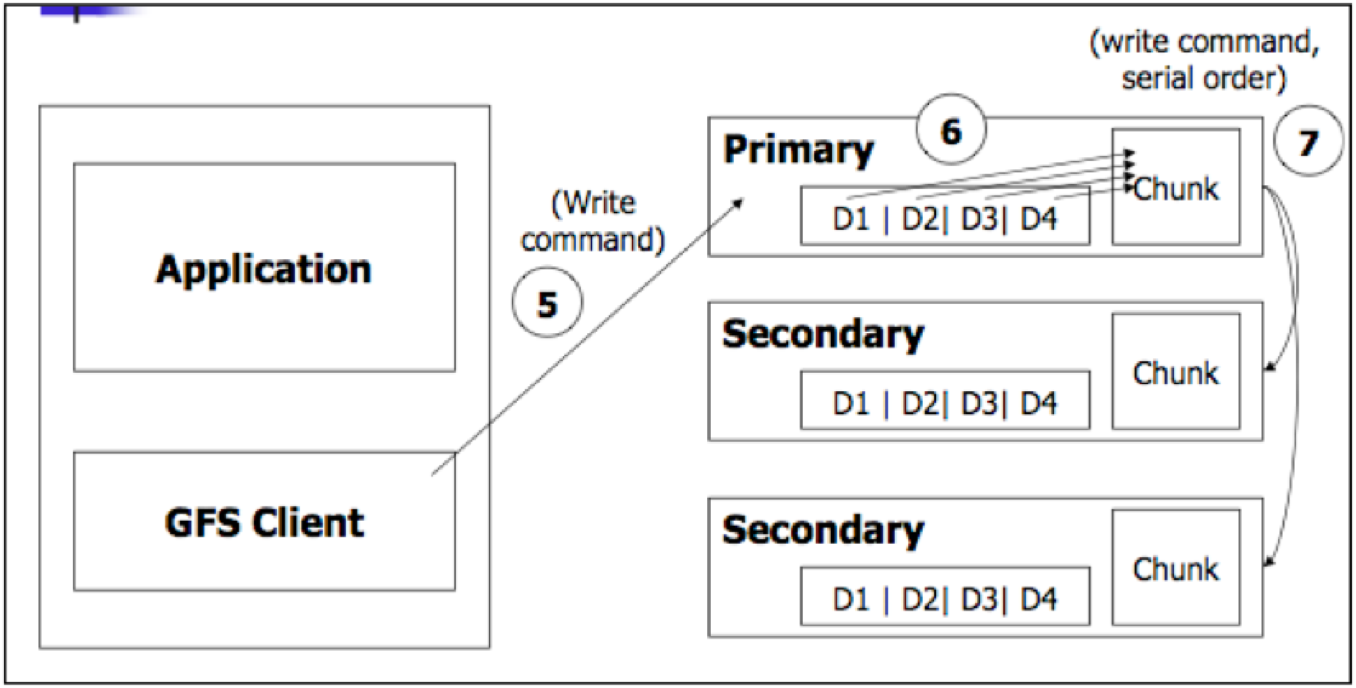
# Write Algorithm

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



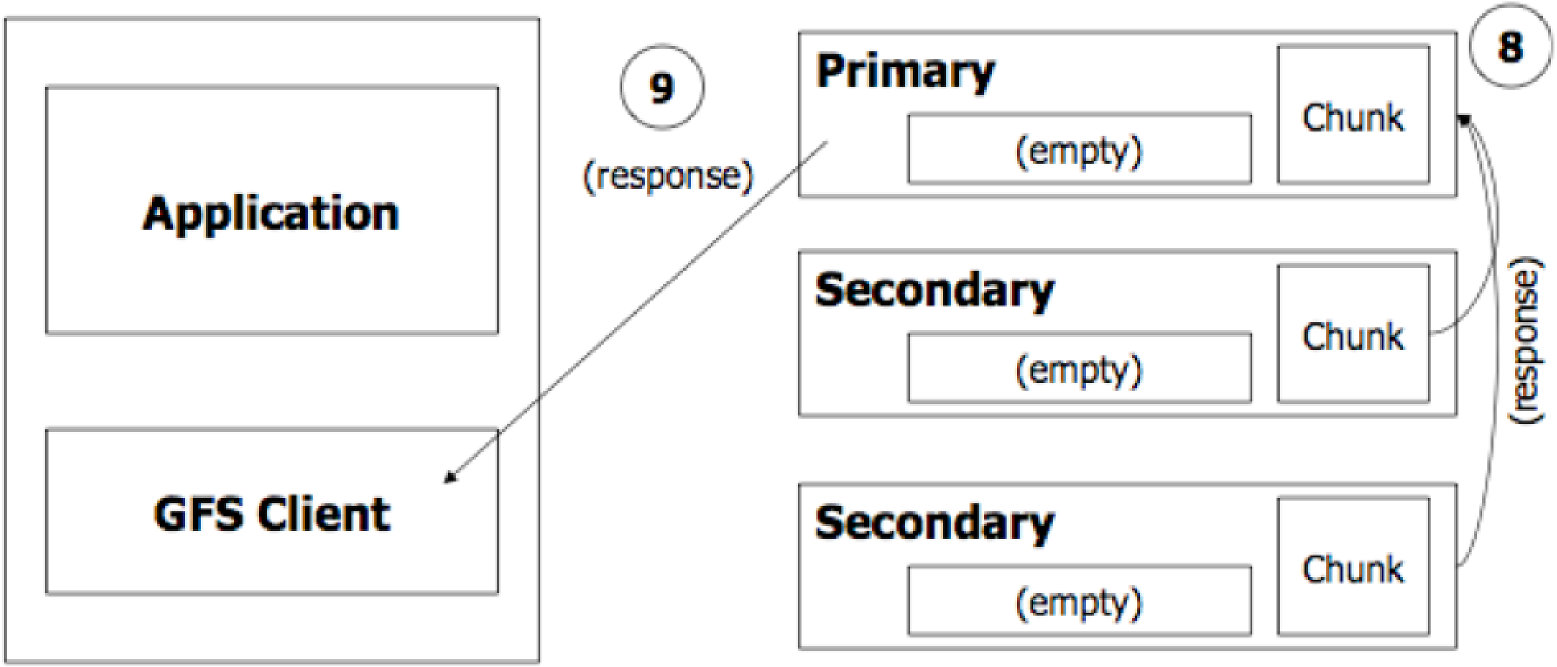
# Write Algorithm

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



# Write Algorithm

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



# Atomic Record Append

- GFS appends it to the file atomically at least once
  - GFS ***picks*** the offset
  - Works for concurrent writers
- Used heavily by Google apps
  - e.g., for files that serve as multiple-producer/single-consumer queues
  - Merge results from multiple machines into one file

# Garbage Collection

- Use garbage collection instead of immediate reclamation when a file is deleted:
  - The master only records chunks (files) that have been removed.
  - Attached with HeartBeat messages, chunkservers know which of their chunks that are orphaned and can be removed.
- Why garbage collection and not eager deletion?
  - Simple and reliable as the master doesn't send and manage deletion messages.
  - Batched operation for lower amortized cost and better timing.
  - The delay provides a safety net against accidental, irreversible deletion.

# Fault Tolerance

- High availability
  - Fast recovery
    - master and chunkservers restartable in a few seconds
  - Chunk replication
    - default: 3 replicas.
  - Shadow masters
- Data integrity
  - Checksum every 64KB block in each chunk

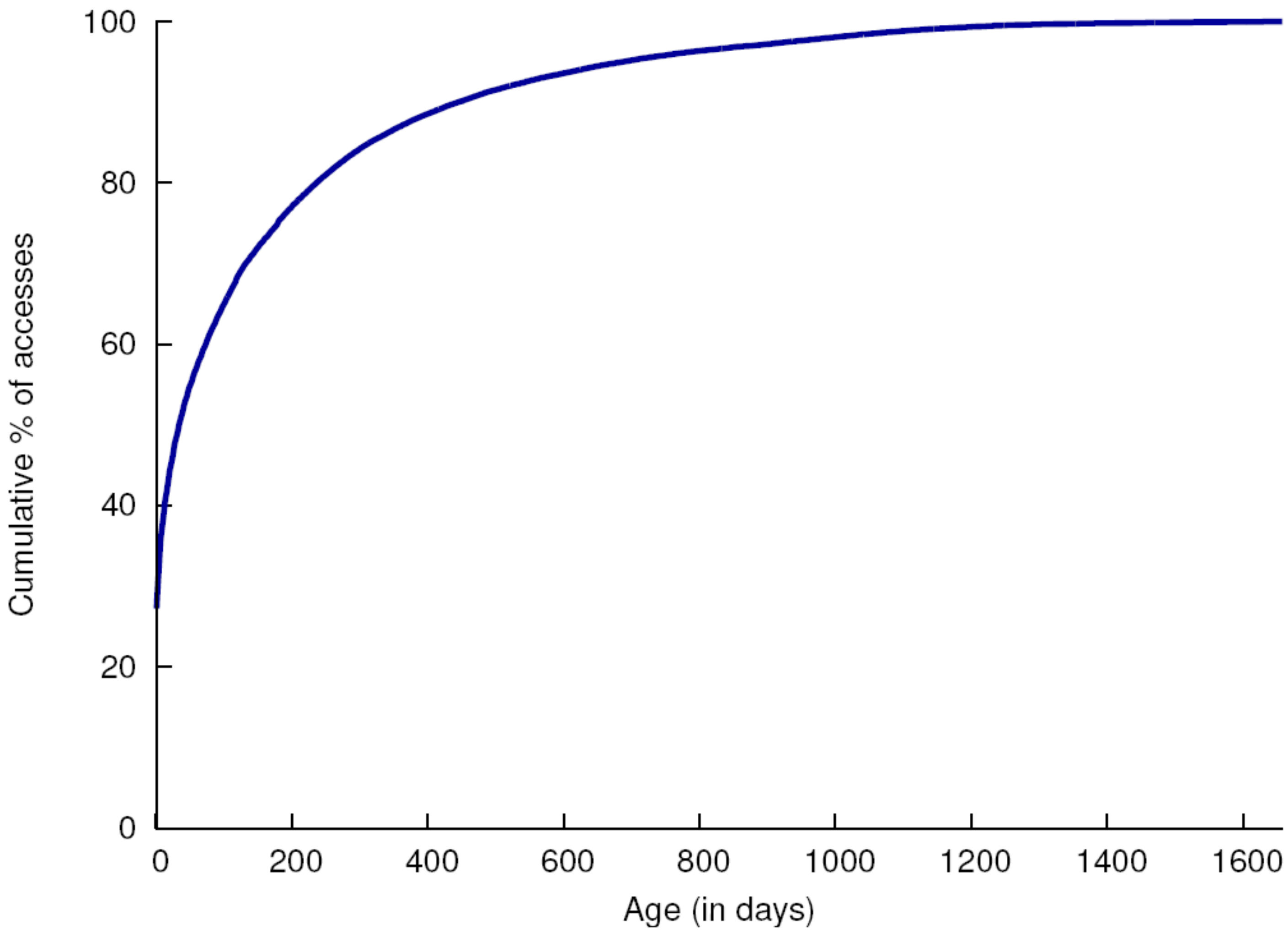
# Conclusion

- GFS demonstrates how to support large-scale processing workloads on commodity hardware
  - design **to tolerate frequent component failures**
  - optimize for **huge** files that are **mostly appended** and **read**
  - feel free to relax and extend FS interface as required
  - go for simple solutions (e.g., single master)
- GFS has met Google's storage needs, therefore good enough for them.

# Facebook's Photo Storage: Motivation

- Facebook stores over 260 billion images
  - 20 PB of data
- Users upload one billion new images each week
  - 60 TB of data
- Facebook serves over one million images per second at peak
- Two types of workloads for image serving
  - Profile pictures – heavy access, smaller size
  - Photo albums – intermittent access, higher at beginning, decreasing over time (long tail)
- Data is written once, read often, never modified, and rarely deleted

# Long Tail Issue




What can you read from the plot?

# Problem Description

Four main goals for photo serving method:

- High throughput and low latency
  - Current file systems need multiple disk accesses for a read
  - Only one data access for a read, metadata are reduced to fit in memory
- Fault-tolerant

  Haystack replicates each photo in geographically distinct locations
- Cost-effective
  - Save money over traditional approaches (reduce reliance on CDNs!)
- Simplicity
  - Make it easy to implement and maintain

# Typical Design

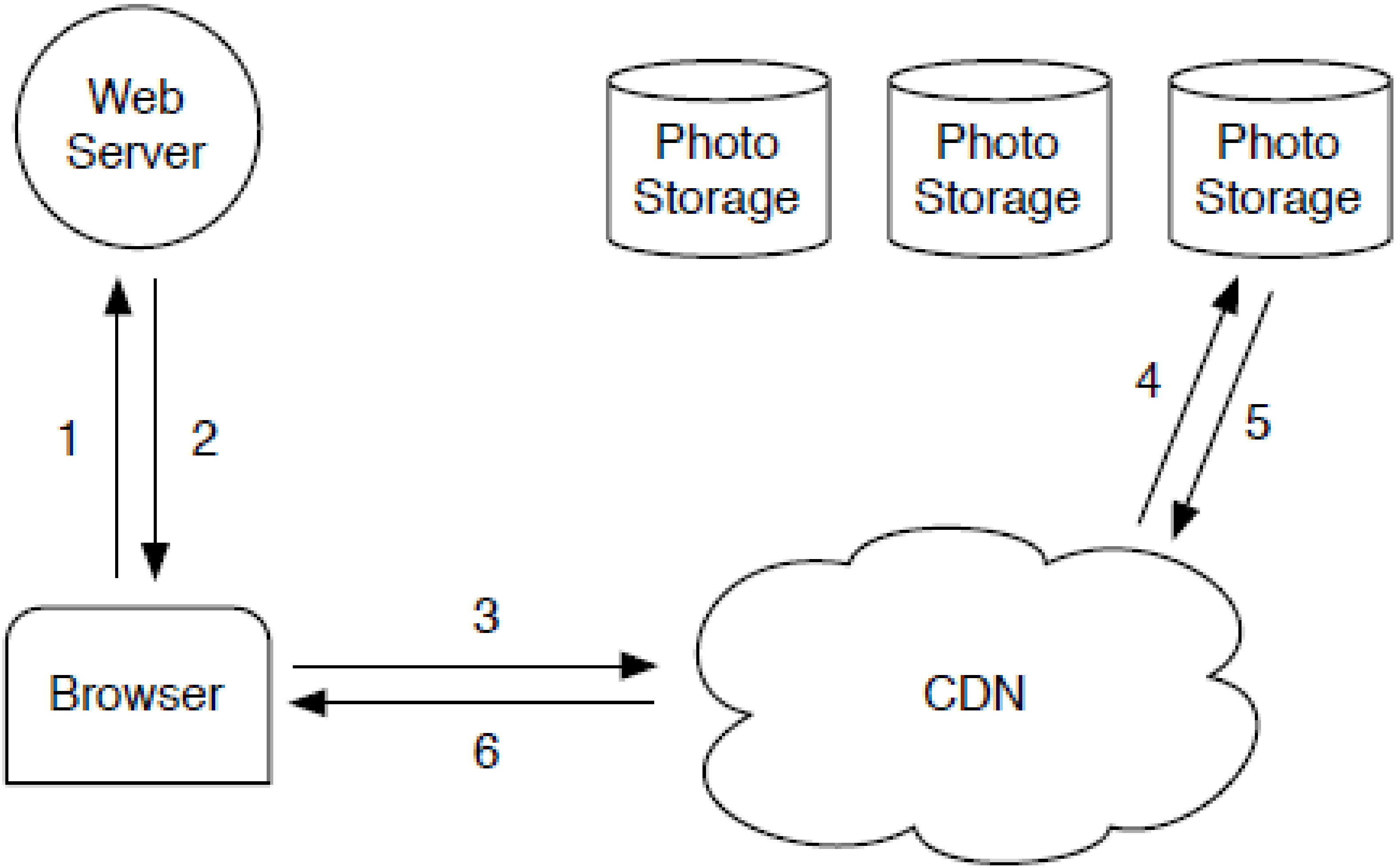



Figure 1: Typical Design

# Facebook's Old Design

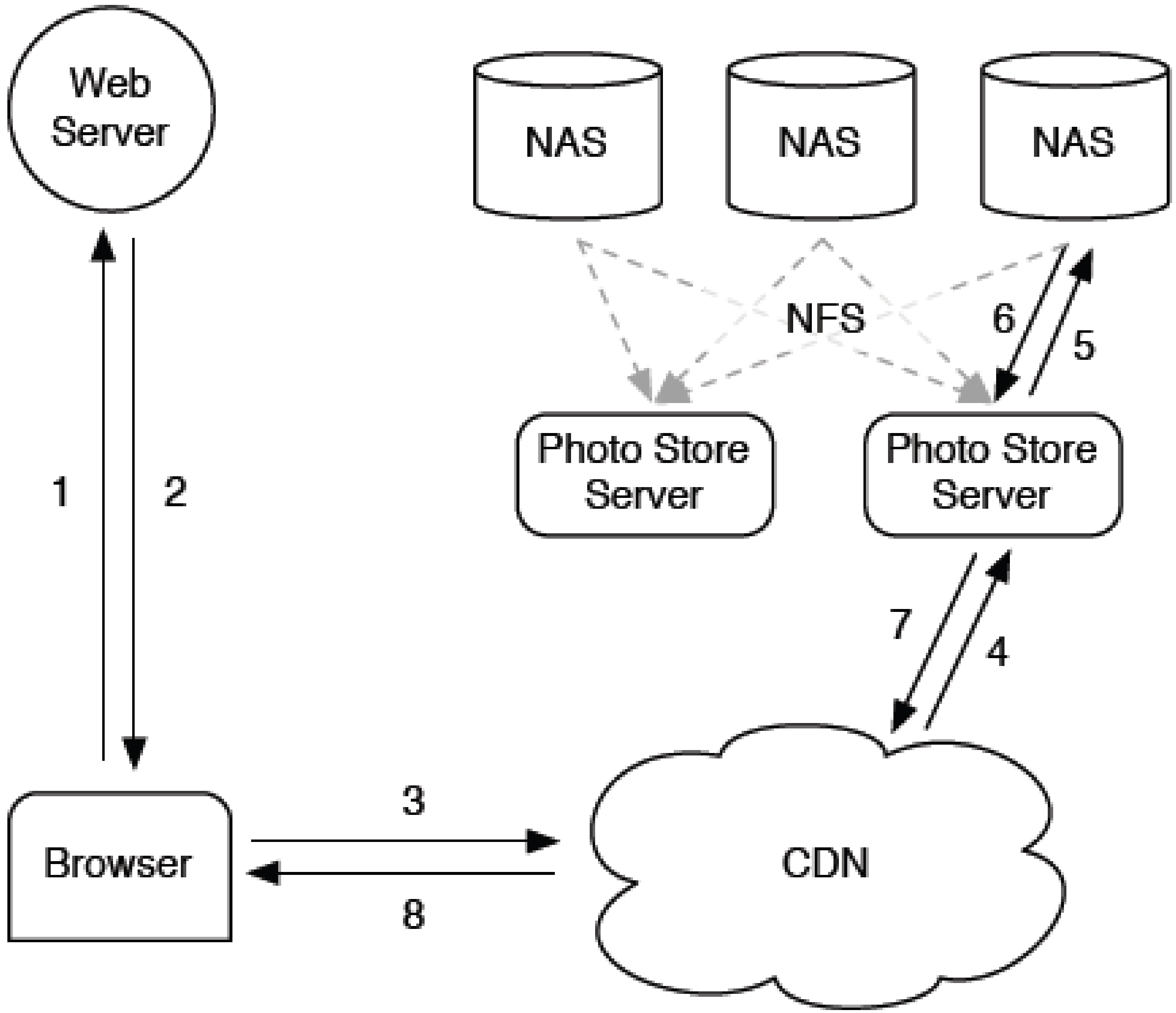



Figure 2: NFS-based Design

# Old Design

The old photo infrastructure consisted of several tiers:

- Upload tier receives users' photo uploads, scales the original images and saves them on the NFS storage tier.
- Photo serving tier receives HTTP requests for photo images and serves them from the NFS storage tier.
- NFS storage tier built on top of commercial storage appliances.

# Features of Old Design

- Since each image is stored in its own file, there is an enormous amount of metadata generated on the storage tier due to the namespace directories and file inodes.
- The amount of metadata far exceeds the caching abilities of the NFS storage tier, resulting in multiple I/O operations per photo upload or read request
- After optimization, there are three disk accesses: one to read the directory metadata into memory, a second to load the inode into memory, and a third to read the file contents)
- High degree of reliance on CDNs = expensive

# Design of Haystack

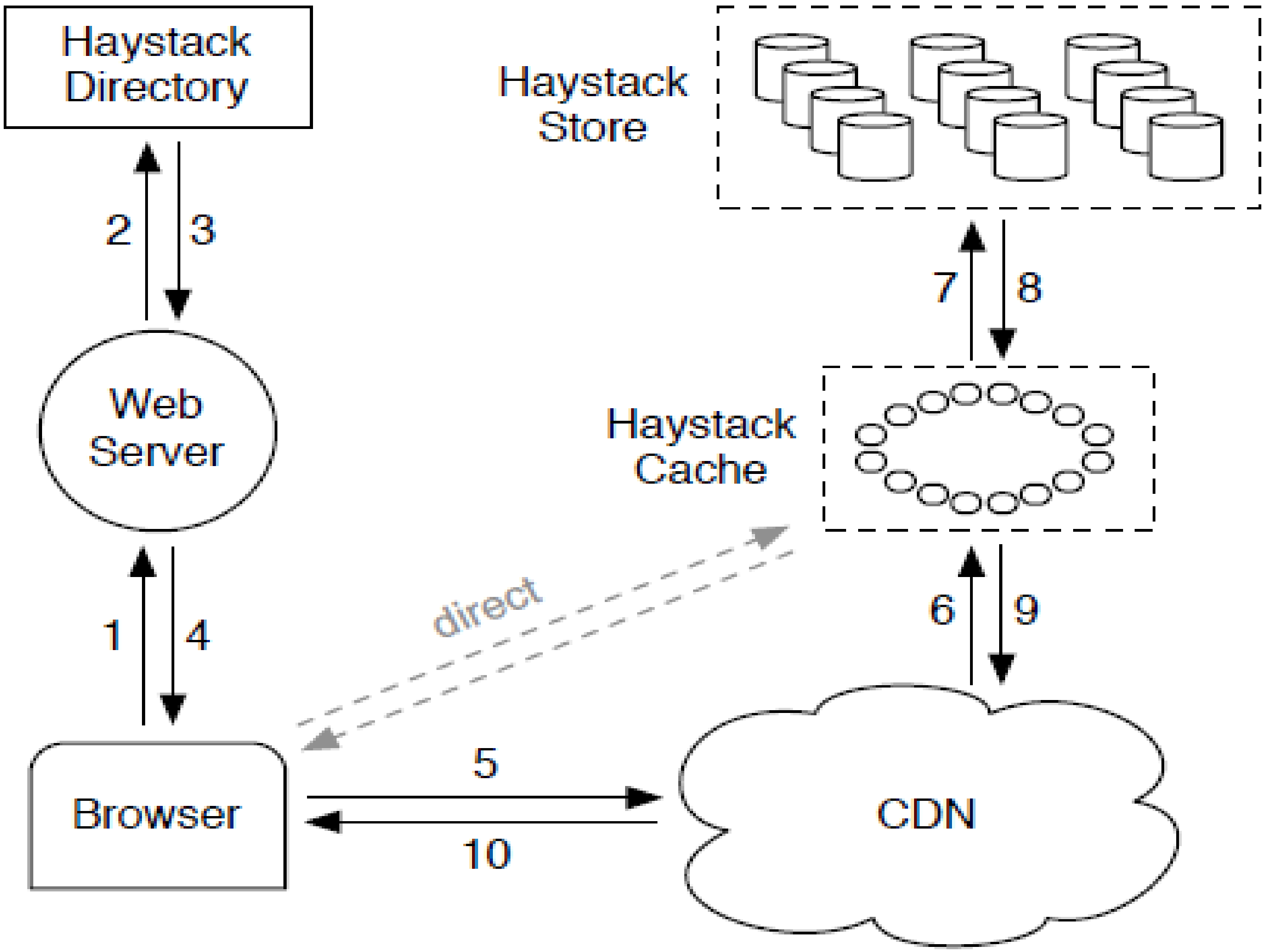



Figure 3: Serving a photo

# Step-through of Operation

User visits page

- Web server receives the request
- Uses Haystack Directory to construct URL for each photo
  - http://**<CDN>**/**<Cache>**/**<Machine id>**/**<Logical volume, Photo>**
  - From which CDN to request the photo
- This portion may be omitted if the photo is available directly from the Cache
- If CDN is unsuccessful, contacts the Cache

# Haystack

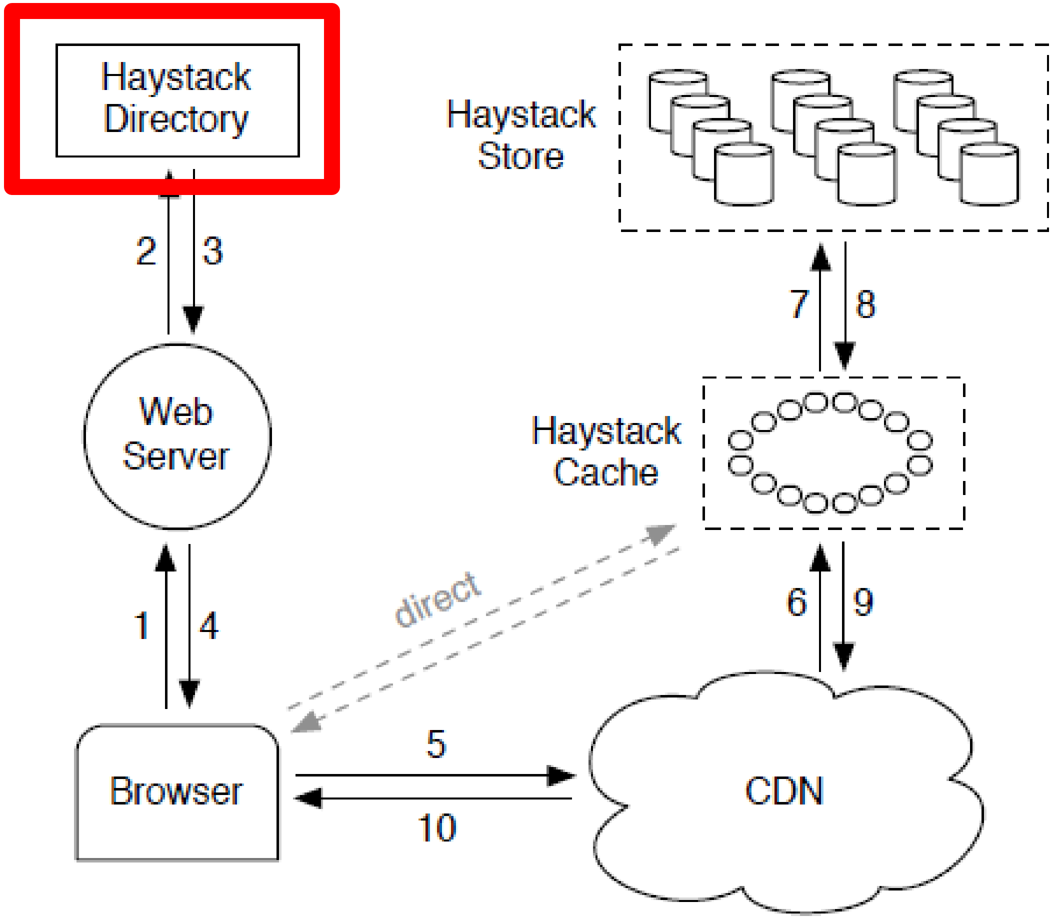



Figure 3: Serving a photo

# Haystack Directory

Four main functions…

- Provides a mapping from logical volumes to physical volumes
- Load balances writes across logical volumes
- Determines whether a photo request should be handled by the CDN or by the Haystack Cache
- Identifies logical volumes that are read-only
  - Operational reasons
  - Reached storage capacity

# Haystack

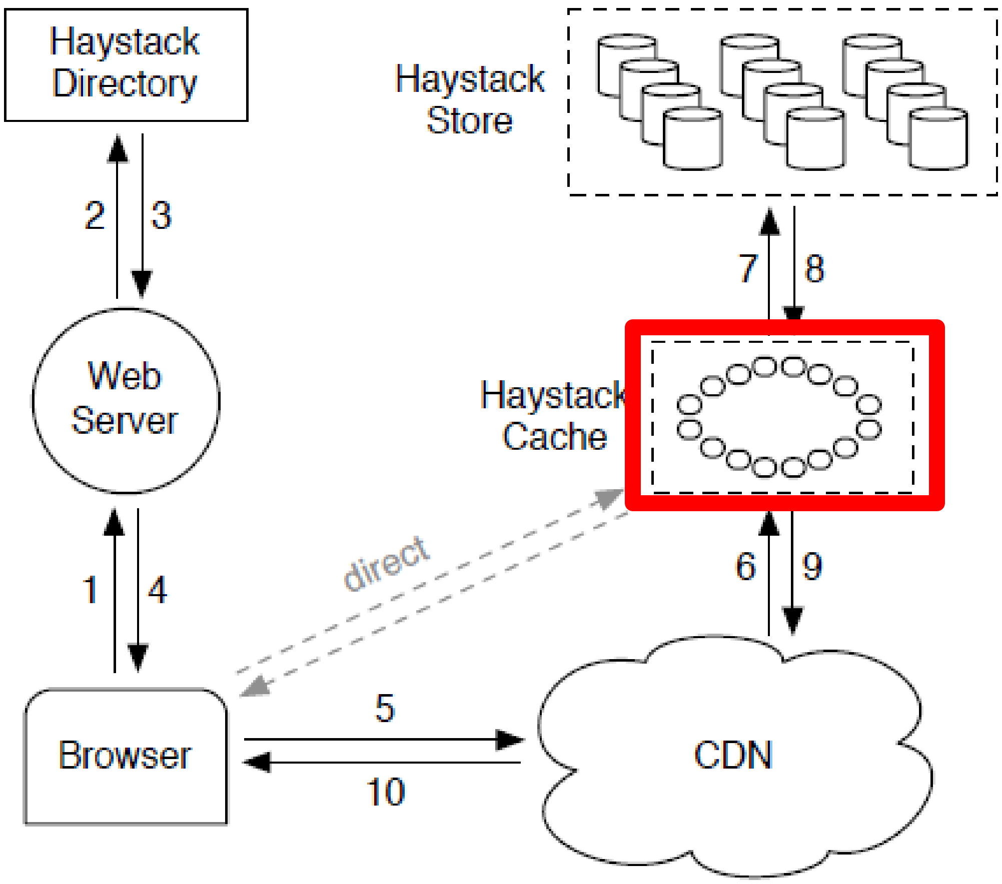



Figure 3: Serving a photo

# Haystack Cache

Distributed hash table, uses photo's id to locate cached data

Receives HTTP requests from CDNs and browsers

- If photo is in Cache, return the photo
- If photo is not in Cache, fetches photo from the Haystack Store and returns the photo

Add a photo to Cache if two conditions are met…

- The request comes directly from a browser, not the CDN
- The photo is fetched from a write-enabled store machine

# Cache Hit Rate

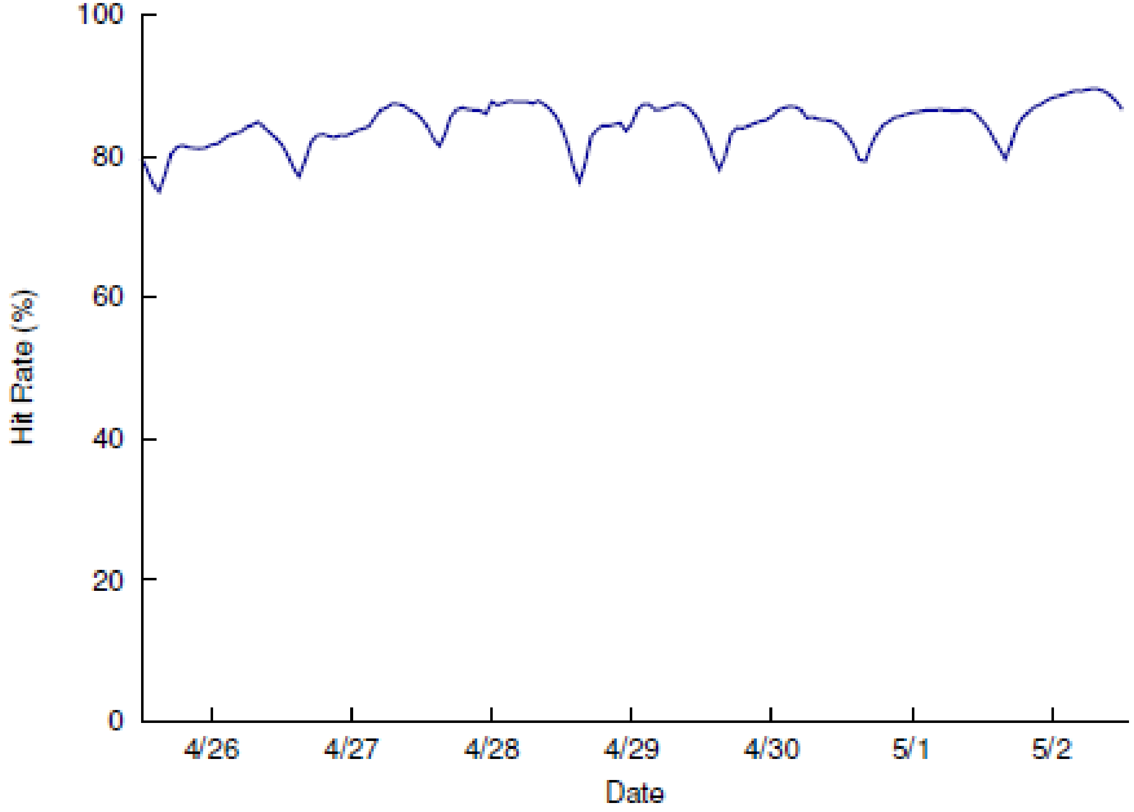



Figure 9: Cache hit rate for images that might be potentially stored in the Haystack Cache.

# Haystack

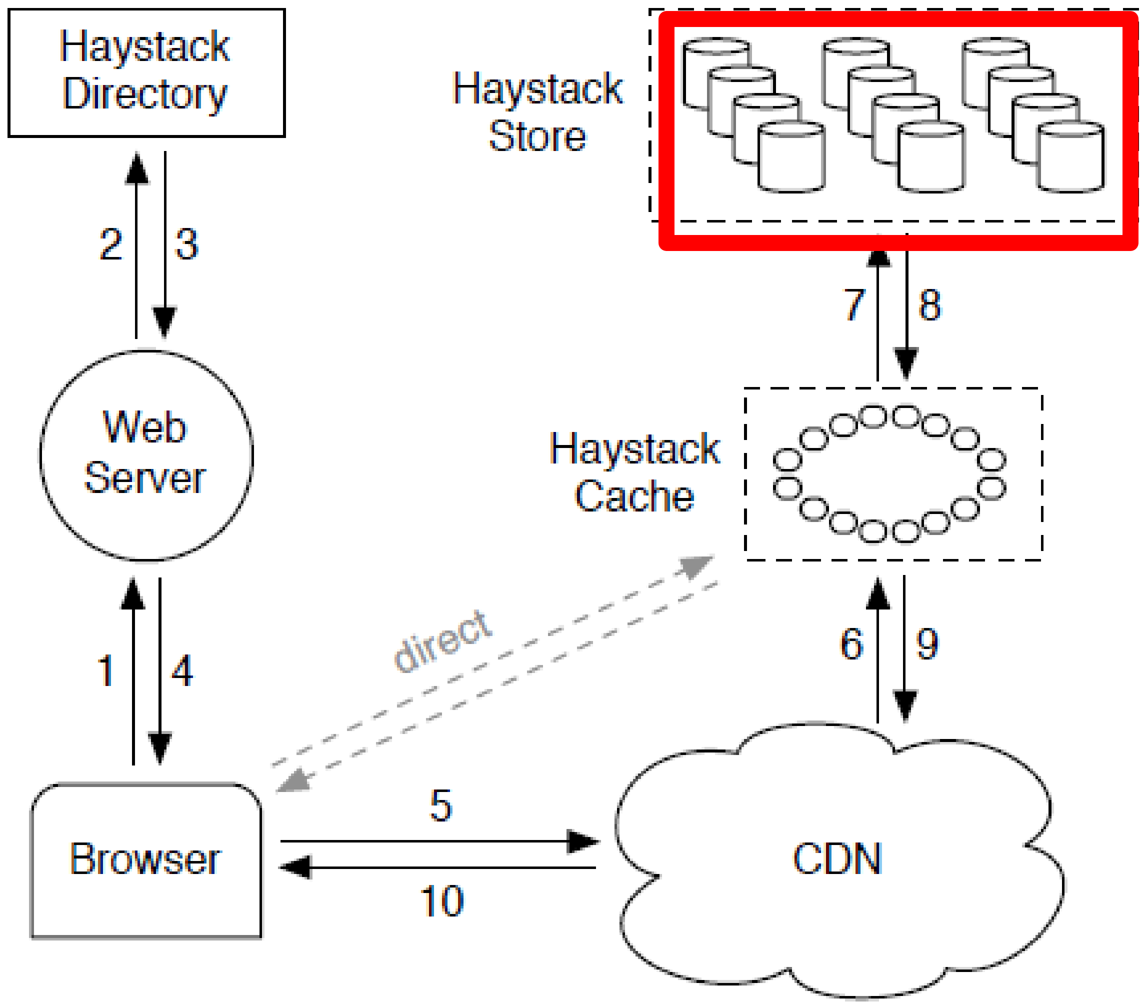
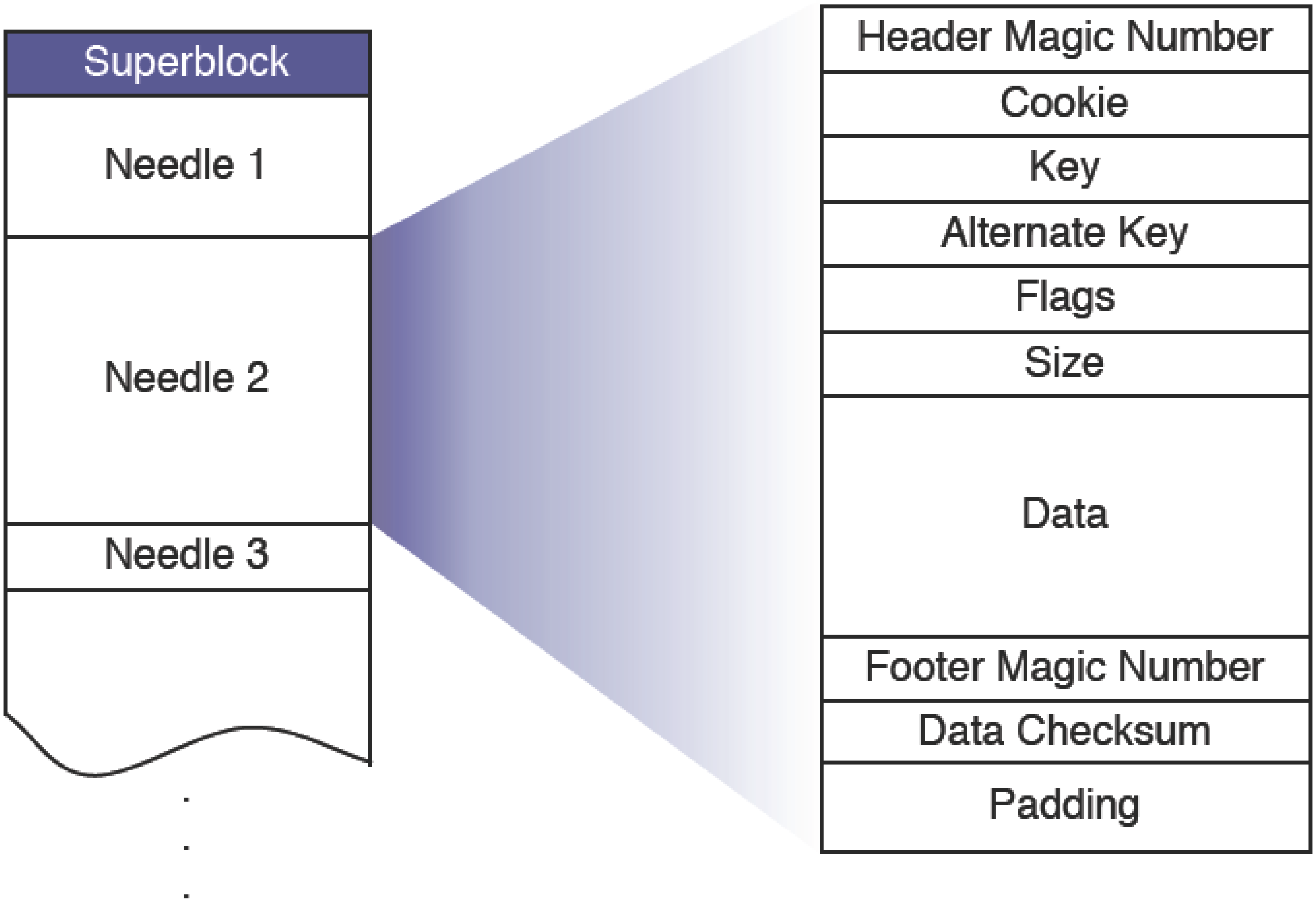



Figure 3: Serving a photo

# Layout of Haystack Store File

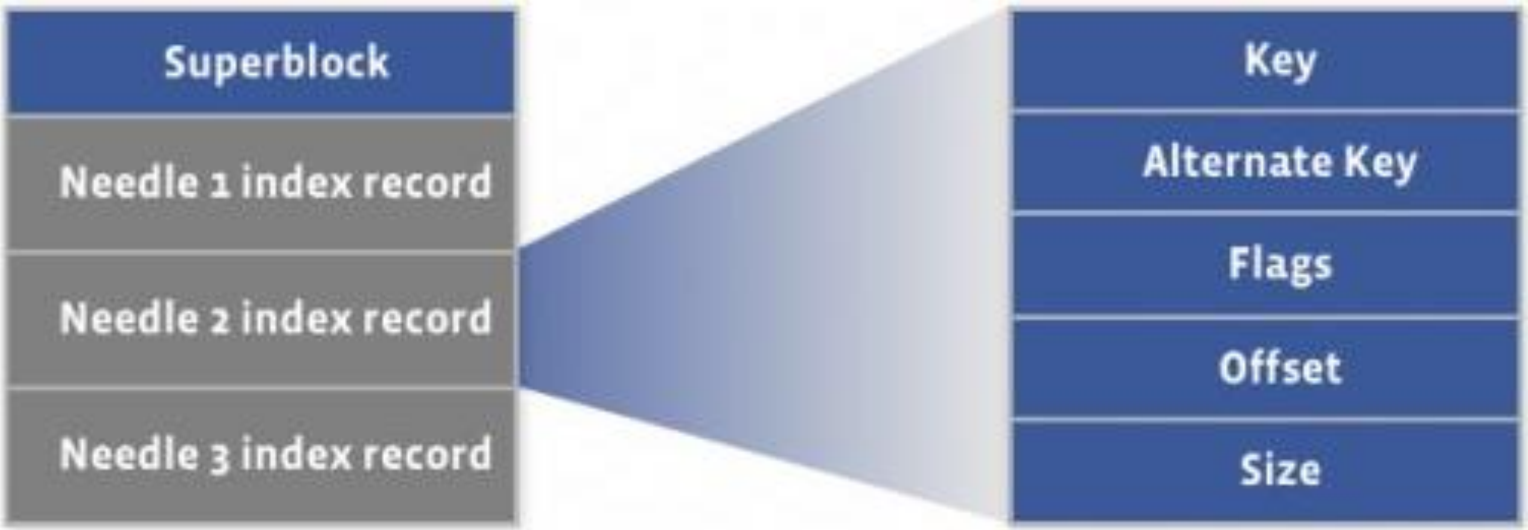


# A Closer Look at the Needles…

- A needle is uniquely identified by its <Offset, Key, Alternate Key, Cookie> tuple, where the offset is the needle offset in the haystack store.

| | |
|---|---|
| Header Magic Number | Magic number used to find the next possible needle during recovery |
| Cookie | Security cookie supplied by the client application to prevent brute force attack |
| Key | 64-bit object key |
| Alternate Key | 32-bit object alternate key |
| Flags | Currently only one signifying that the object has been removed |
| Size | Data size |
| Footer Magic Number | Magic number used to find the possible needle end during recovery |
| Data Checksum | Checksum for the data portion of the needle |
| Padding | Total needle size is aligned to 8 bytes |

# Haystack Index File




| | |
|---|---|
| Key | 64-bit object key |
| Alternate Key | 32-bit object alternate key |
| Flags | Currently unused |
| Offset | Needle offset in the haystack store |
| Size | Needle data size |

# Haystack Index File

The index file provides the minimal metadata required to locate a particular needle in the store

- Main purpose: allow quick loading of the needle metadata into memory without traversing the larger Haystack store file upon restarting
- Index is usually less than 1% the size of the store file

# Haystack Store

- Each Store machine manages multiple physical volumes
- Can access a photo quickly using only the id of the corresponding logical volume and the file offset of the photo
- Handles three types of requests…
  - Read
  - Write
  - Delete

# Haystack Store Read

- Cache machine supplies the logical volume id, key, alternate key, and cookie to the Store machine
  -- Purpose of the cookie?
- Store machine looks up the relevant metadata in its in-memory mappings
- Seeks to the appropriate offset in the volume file, reads the entire needle
- Verifies cookie and integrity of the data
- Returns data to the Cache machine

# Haystack Store Write

- Web server provides logical volume id, key, alternate key, cookie, and data to Store machines
- Store machines synchronously append needle images to physical volume files
- Update in-memory mappings as needed

# Haystack Store Delete

- Store machine sets the delete flag in both the in-memory mapping and in the volume file
- Space occupied by deleted needles is lost!
  - How to reclaim?
    - Compaction!
    - Important because 25% of photos get deleted in a given year.

# Haystack Advantages

Reduced disk I/O

- 10 TB/node -> 10 GB of metadata
  - This amount is easily cacheable!

Simplified metadata

- No directory structures/file names
  - 64-bit ID
- Results in easier lookups

Single photo serving and storage layer

- Direct I/O path between client and storage
- Results in higher bandwidth

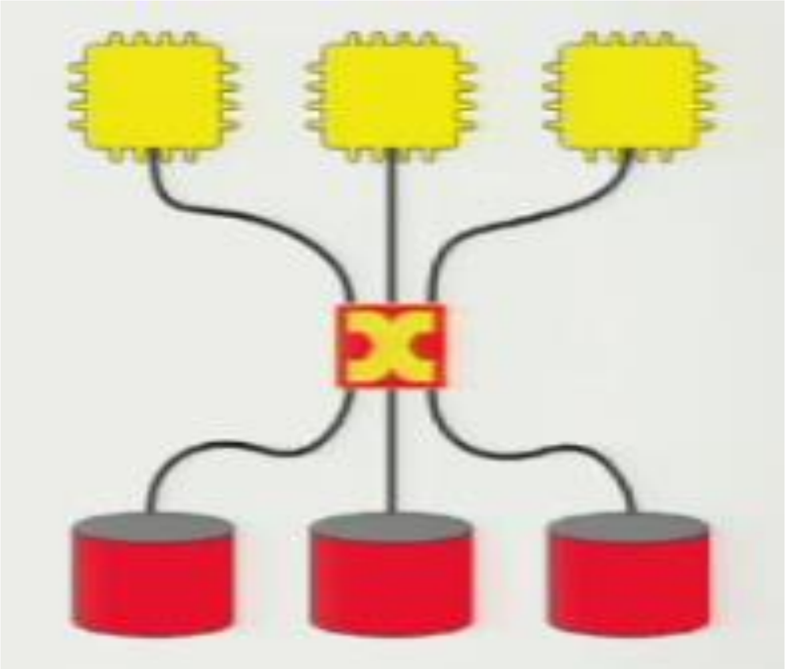
# Microsoft's Flat Storage System

Flat Datacenter Storage (FDS) is a high-performance, fault-tolerant, large-scale, locality-oblivious blob store.

E. B. Nightingale et al. "**Flat Datacenter Storage**", OSDI'12.

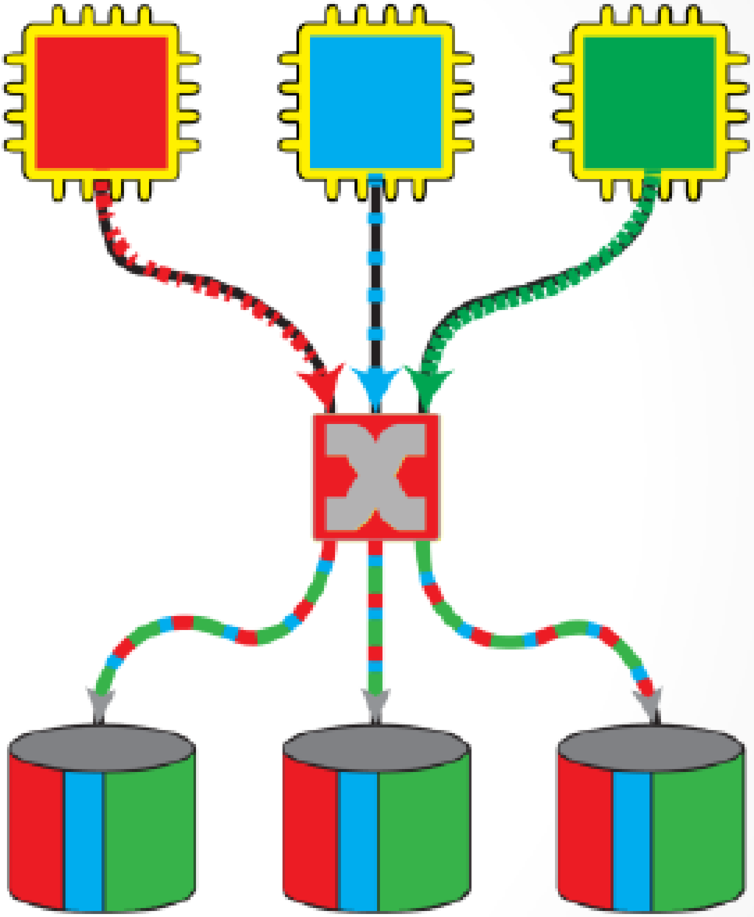
Context: why do we need a <u>locality-oblivious</u> FDS?
Here is what we need for a single machine or computer

- bunch of processors
- bunch of disks
- controller



# Writing

- Fine-grained write striping → statistical multiplexing → high disk utilization
- Good performance and disk efficiency



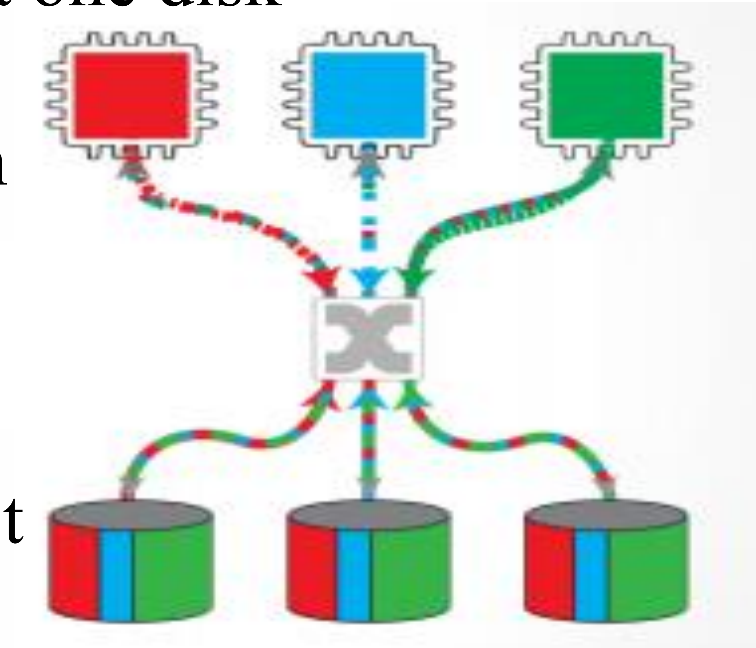
# Reading

We get full **performance** out of the disks because all the disks stay busy even if some processes consume data slowly and other quickly

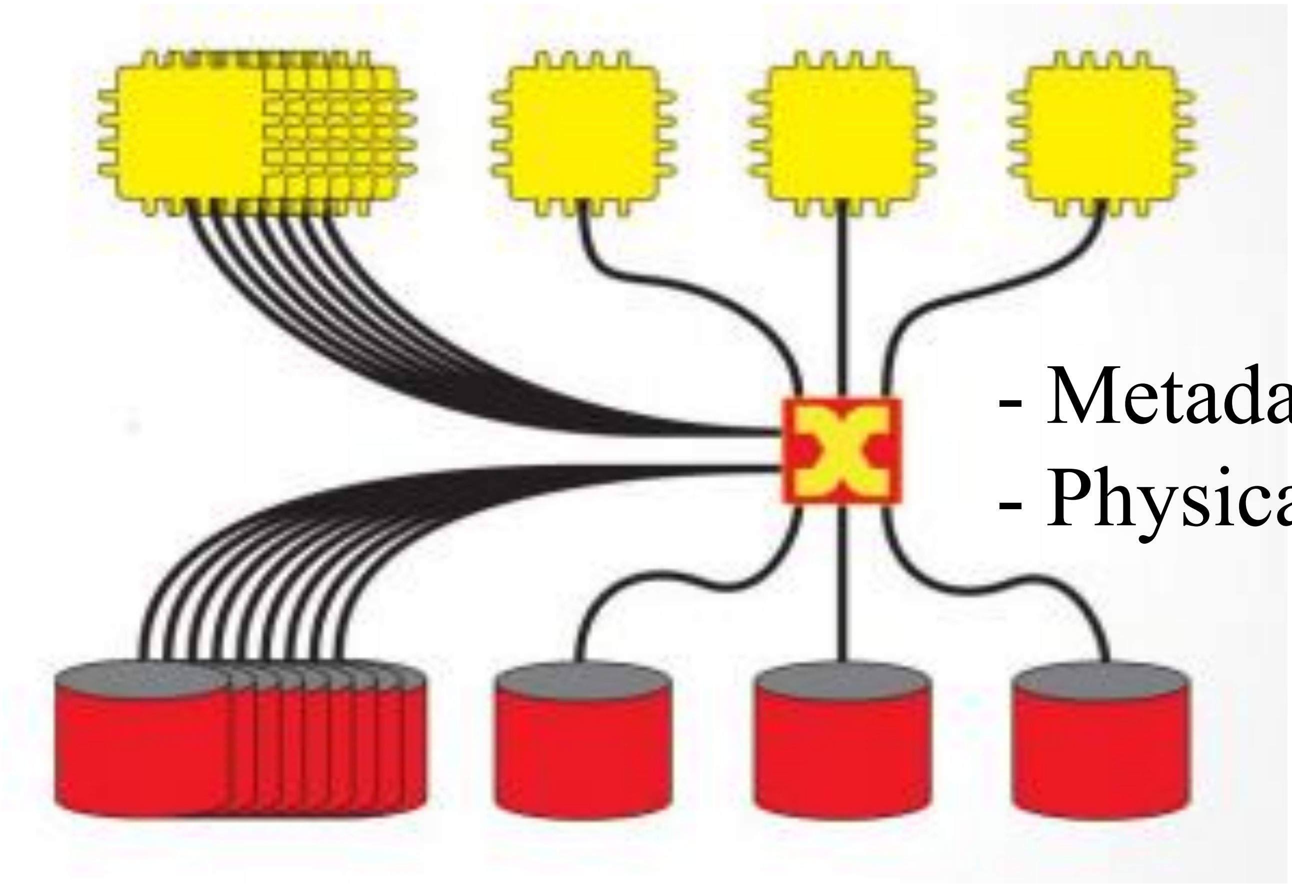
**Programmers** can pretend there's just one disk

If the need is to attack a large problem in **parallel** the input doesn't need to be partitioned in advance.)

Another benefit is that is easy to adjust the **ratio** of processors and disks



# How much can this architecture scale?



- Metadata management
- Physically eouting data

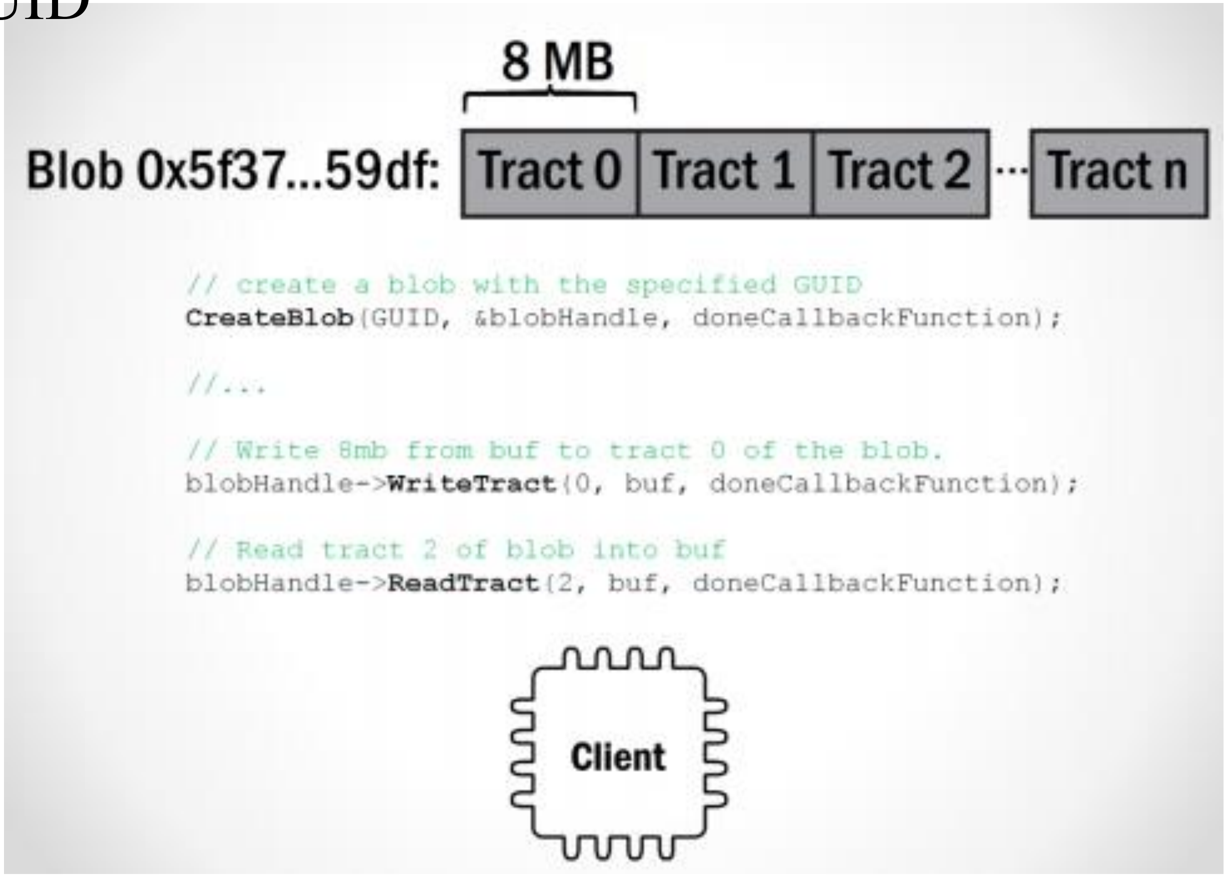
# Blobs and tracts

In FDS, data is stored in blobs

A blob is named with a GUID

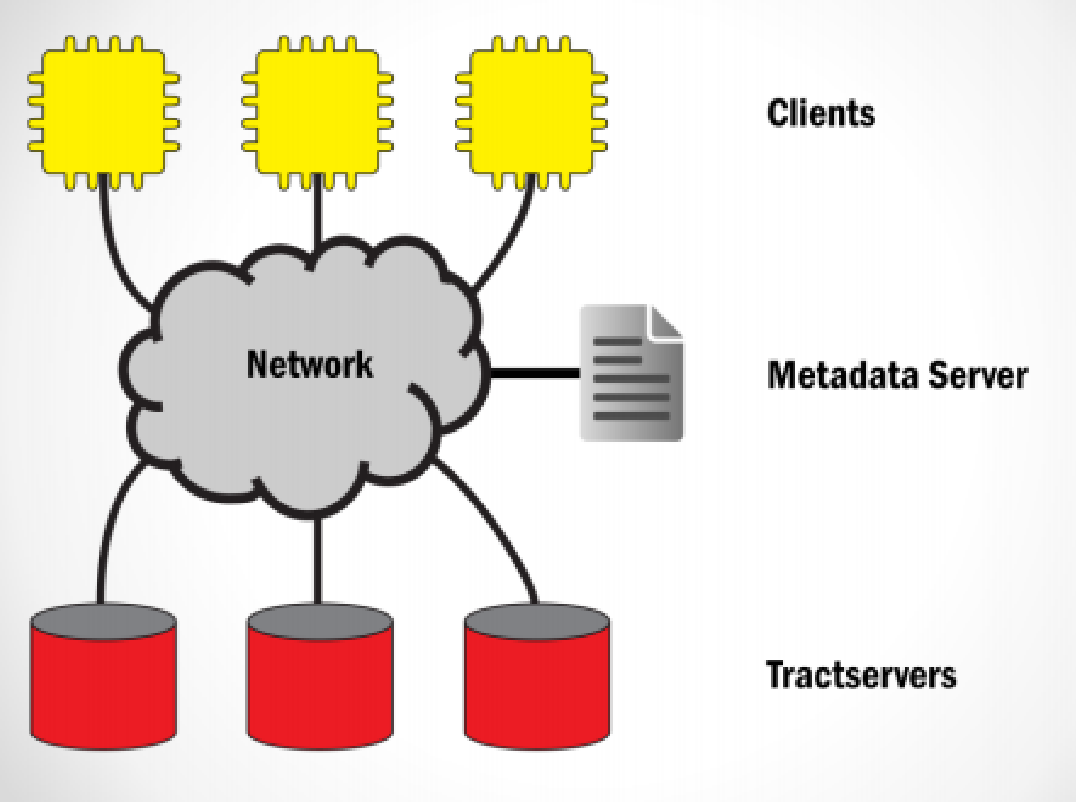
Reads from and writes to a blob are done in units called tracts.

Each tract within a blob is numbered sequentially starting from 0.

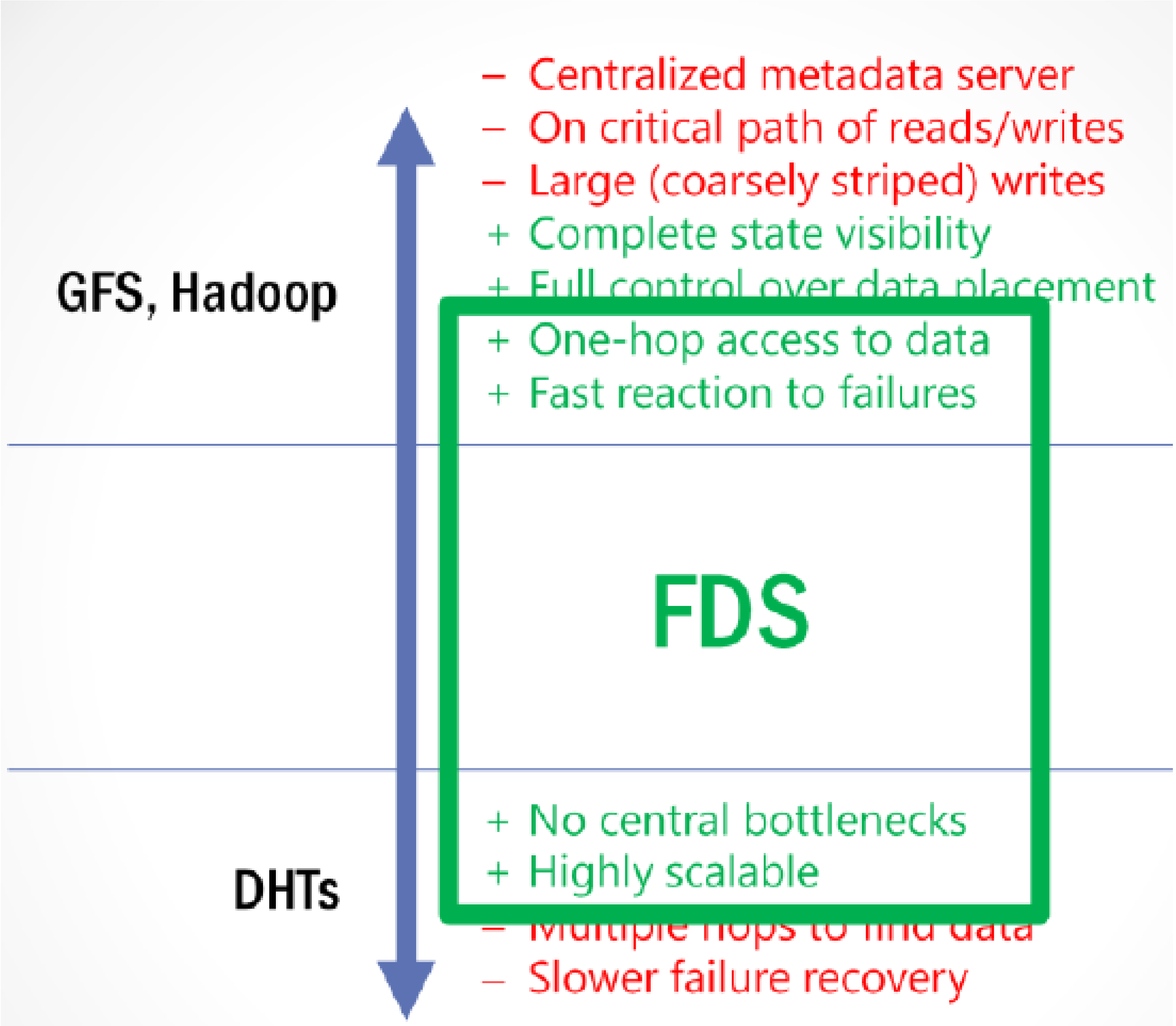


# Design

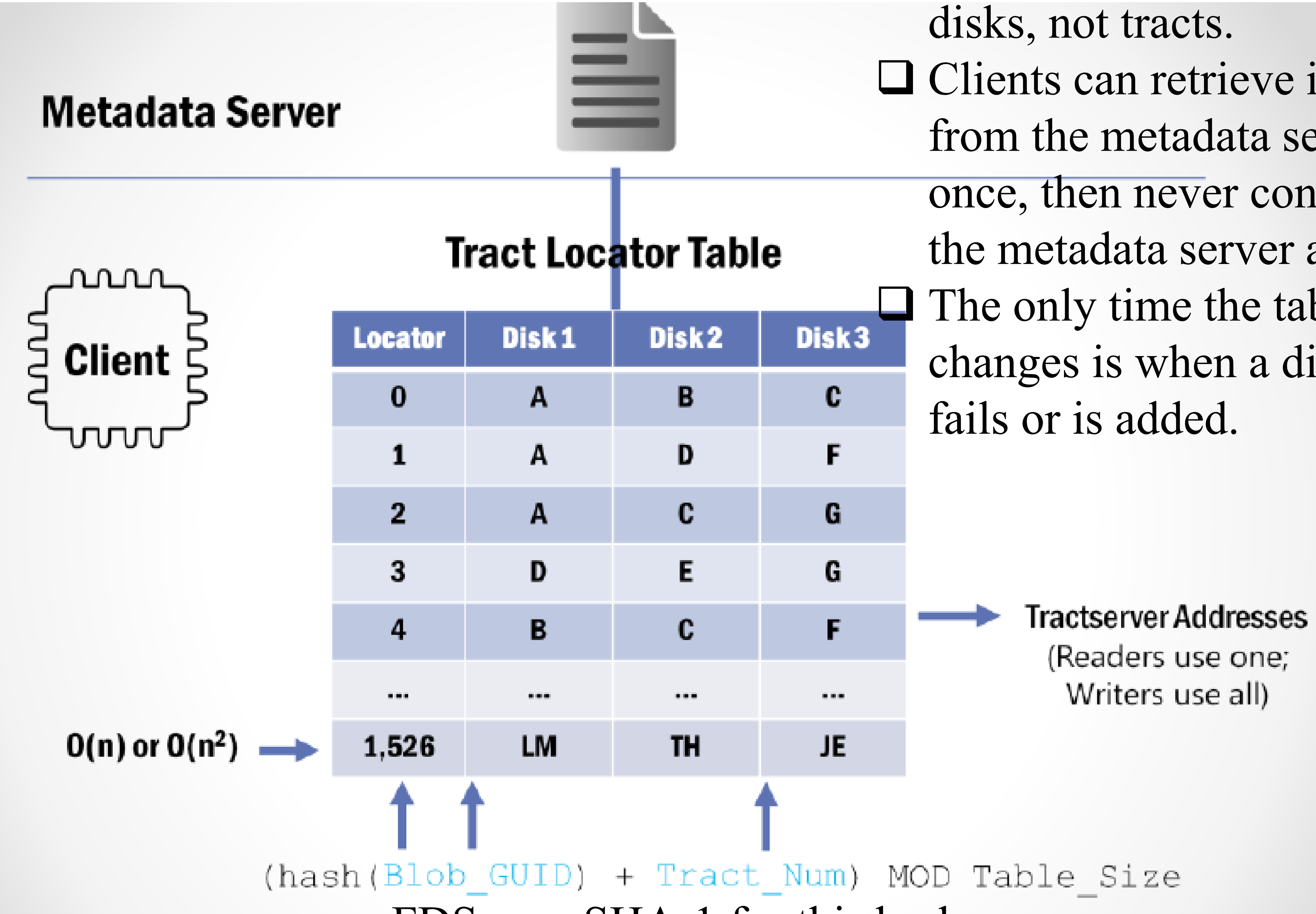
The metadata server coordinates the cluster and helps clients meet with tractservers.




How does the client know which tractserver should be used to read or write a tract?”


– Centralized metadata server
– On critical path of reads/writes
– Large (coarsely striped) writes
+ Complete state visibility
+ Full control over data placement
+ One-hop access to data
+ Fast reaction to failures
GFS, Hadoop
FDS
+ No central bottlenecks
+ Highly scalable
– Multiple hops to find data
– Slower failure recovery
DHTs

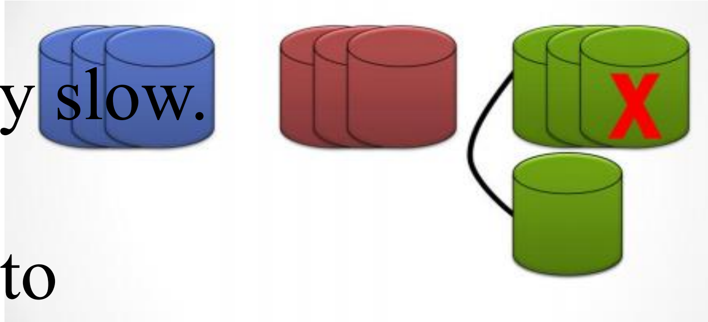
- The table only contains disks, not tracts.
- Clients can retrieve it from the metadata server once, then never contact the metadata server again.
- The only time the table changes is when a disk fails or is added.




| Locator | Disk 1 | Disk 2 | Disk 3 |
|---|---|---|---|
| 0 | A | B | C |
| 1 | A | D | F |
| 2 | A | C | G |
| 3 | D | E | G |
| 4 | B | C | F |
| ... | ... | ... | ... |
| 1,526 | LM | TH | JE |

FDS uses SHA-1 for this hash.

# Failure Recovery



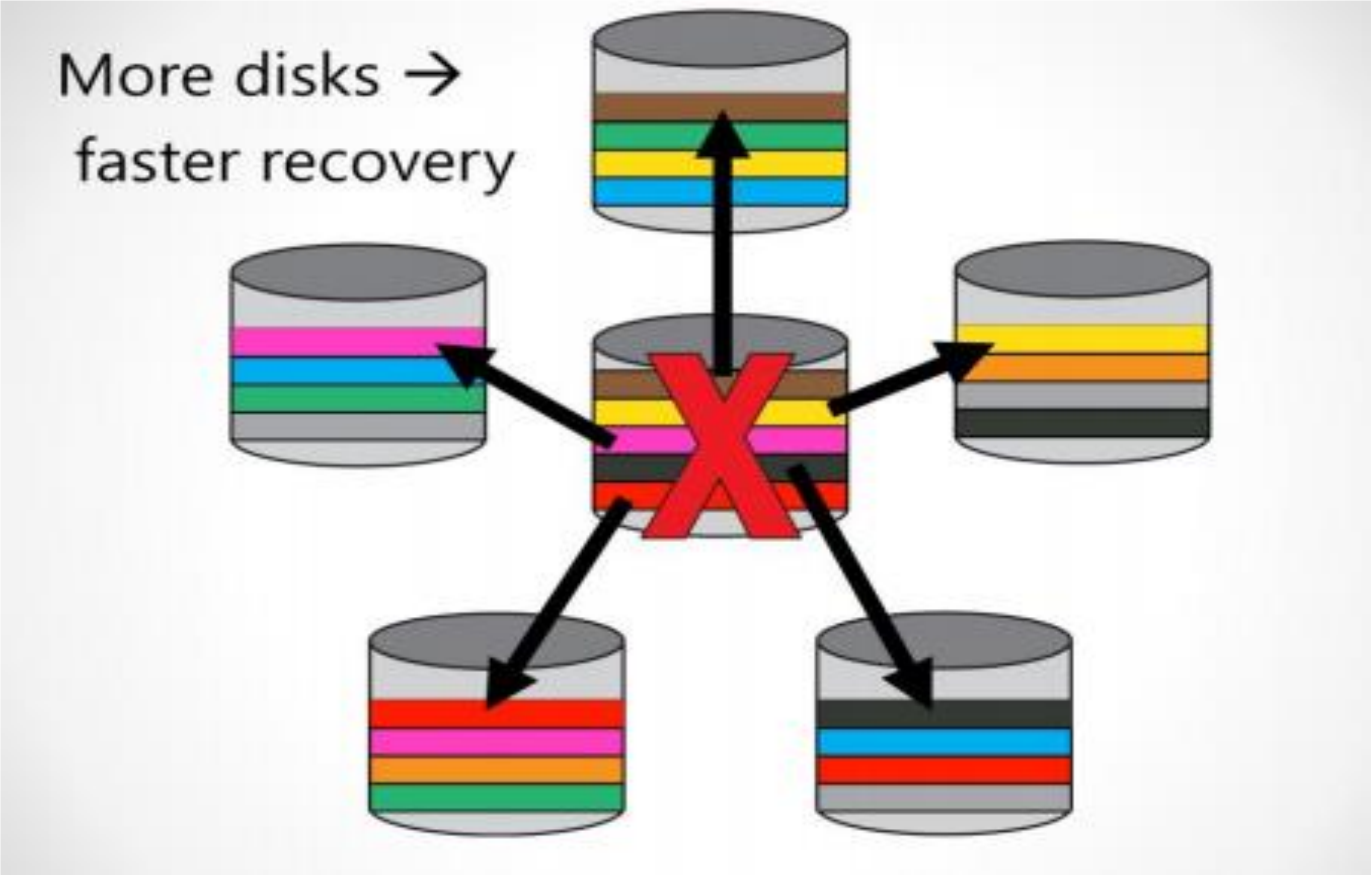
This simple method of replication is very slow.

In FDS, when a disk dies, the goal isn't to reconstruct an exact duplicate of the disk that died.

FDS wants to make sure that somewhere in the system, extra copies of the lost data get made.
It doesn't matter where.

When a disk dies all the other disks contain some backup copies of that disk's data. Every disk sends (in parallel) a copy of its small part of the lost data to some other disk that has some free space.
Recovery's speed grows linearly whith N

# FDS Recovery Solution



# Constructing the Table for Quick Recovery

| Locator | Disk 1 | Disk 2 | Disk 3 |
|---|---|---|---|
| 1 | A | B | C |
| 2 | A | C | Z |
| 3 | A | D | H |
| 4 | A | E | M |
| 5 | A | F | C |
| 6 | A | G | P |
| ... | ... | ... | ... |
| 648 | Z | W | H |
| 649 | Z | X | L |
| 650 | Z | Y | C |

- All **disk pairs** appear in the table
- *n* disks each recover 1/*n*th of the lost data in parallel

| Locator | Disk 1 | Disk 2 | Disk 3 |
|---|---|---|---|
| 1 | ~~A~~ M | B | C |
| 2 | ~~A~~ S | C | Z |
| 3 | ~~A~~ R | D | H |
| 4 | ~~A~~ D | E | M |
| 5 | ~~A~~ S | F | C |
| 6 | ~~A~~ N | G | P |
| ... | ... | ... | ... |
| 648 | Z | W | H |
| 649 | Z | X | L |
| 650 | Z | Y | C |

- All **disk pairs** appear in the table
- *n* disks each recover 1/*n*th of the lost data in parallel

# Ceph: A Scalable, High-Performance Distributed File System

Scalability

- Storage capacity, throughput, client performance. Emphasis on HPC.

Reliability

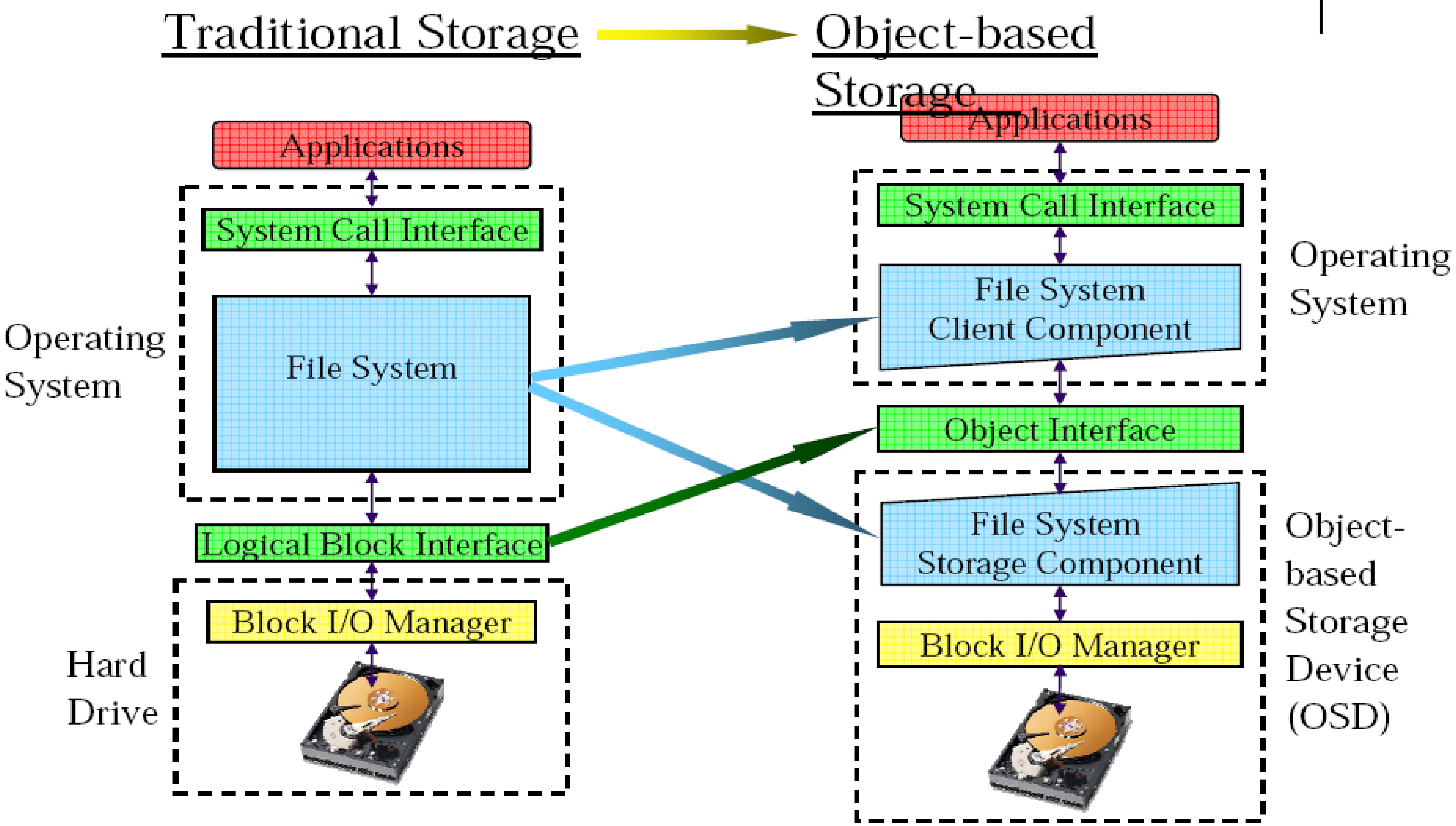
- "…failures are the norm rather than the exception…"

Performance

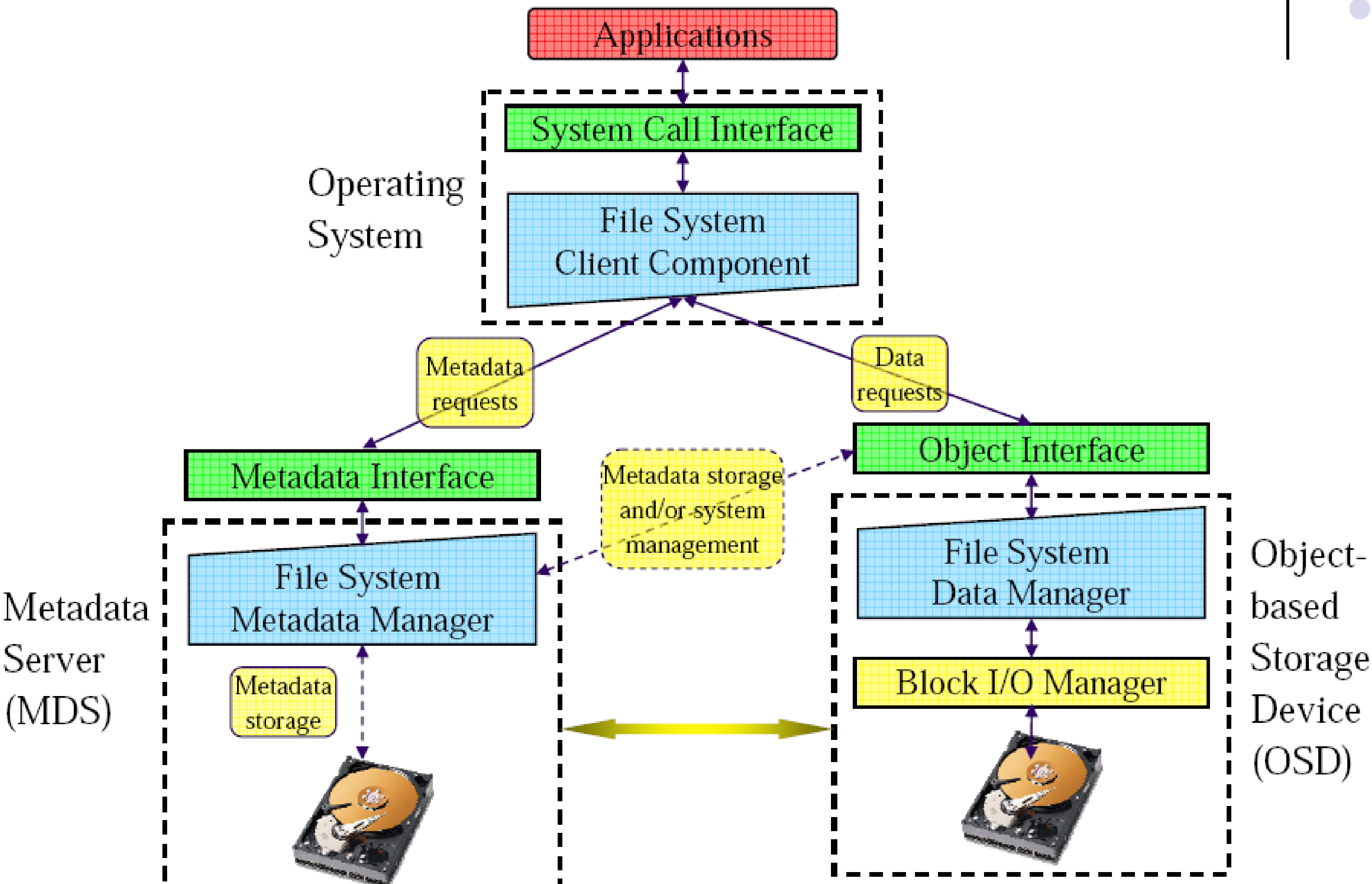
- Dynamic workloads

S. A. Weil et al., "**Ceph: A Scalable, High-Performance Distributed File System**", in OSDI'06

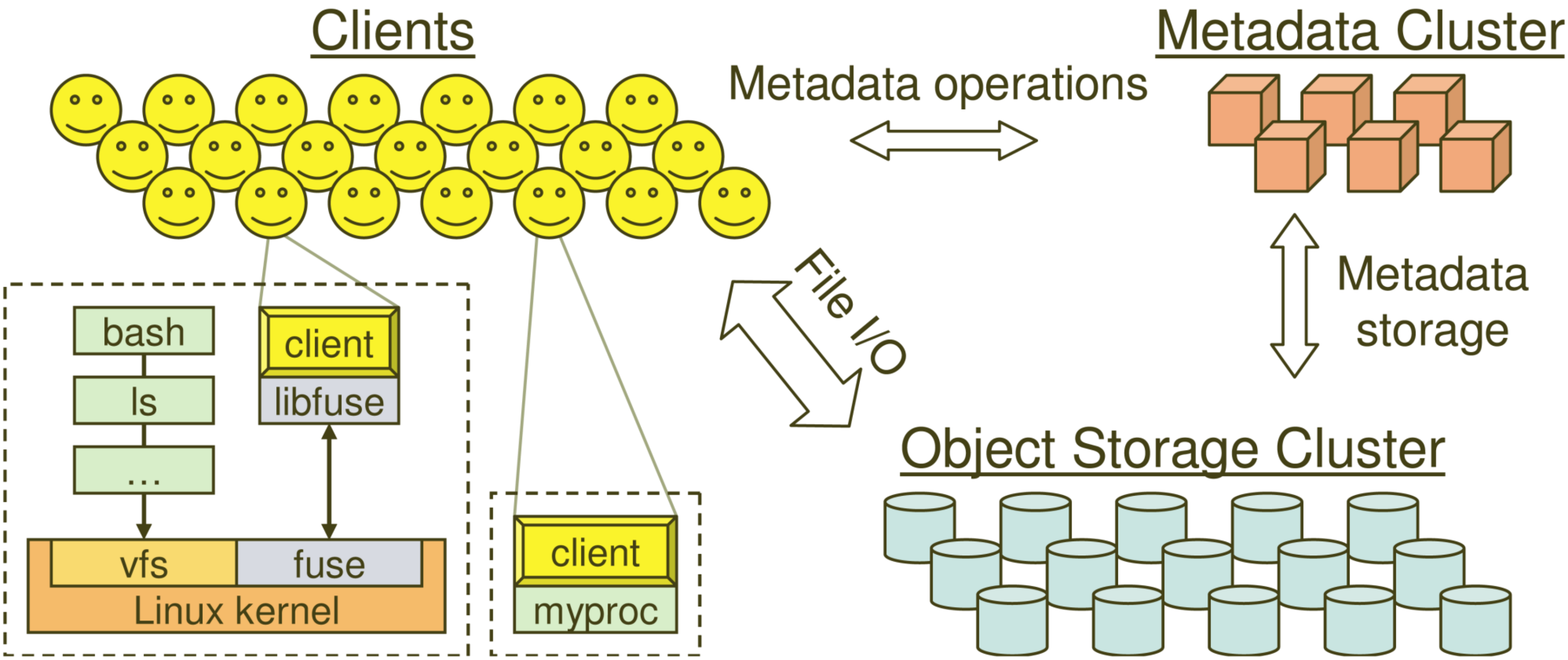
# First Key Idea: Object-based Storage



# Second Key Idea: Decoupled Data and Metadata



# System Overview



# Key Features

Decoupled data and metadata

- CRUSH
  - Files striped onto predictably named objects
  - CRUSH maps objects to storage devices

Dynamic Distributed Metadata Management

- Dynamic subtree partitioning
  - Distributes metadata amongst MDSs

Object-based storage

- OSDs handle migration, replication, failure detection and recovery

# Client Operation

Ceph interface

- Nearly POSIX
- Decoupled data and metadata operation

User space implementation

- FUSE or directly linked

# Client Access Example

1. Client sends *open* request to MDS
2. MDS returns capability, file inode, file size and stripe information (map file data into objects)
3. Client read/write directly from/to OSDs
4. MDS manages the capability
5. Client sends *close* request, relinquishes capability, provides MDS with the new file size

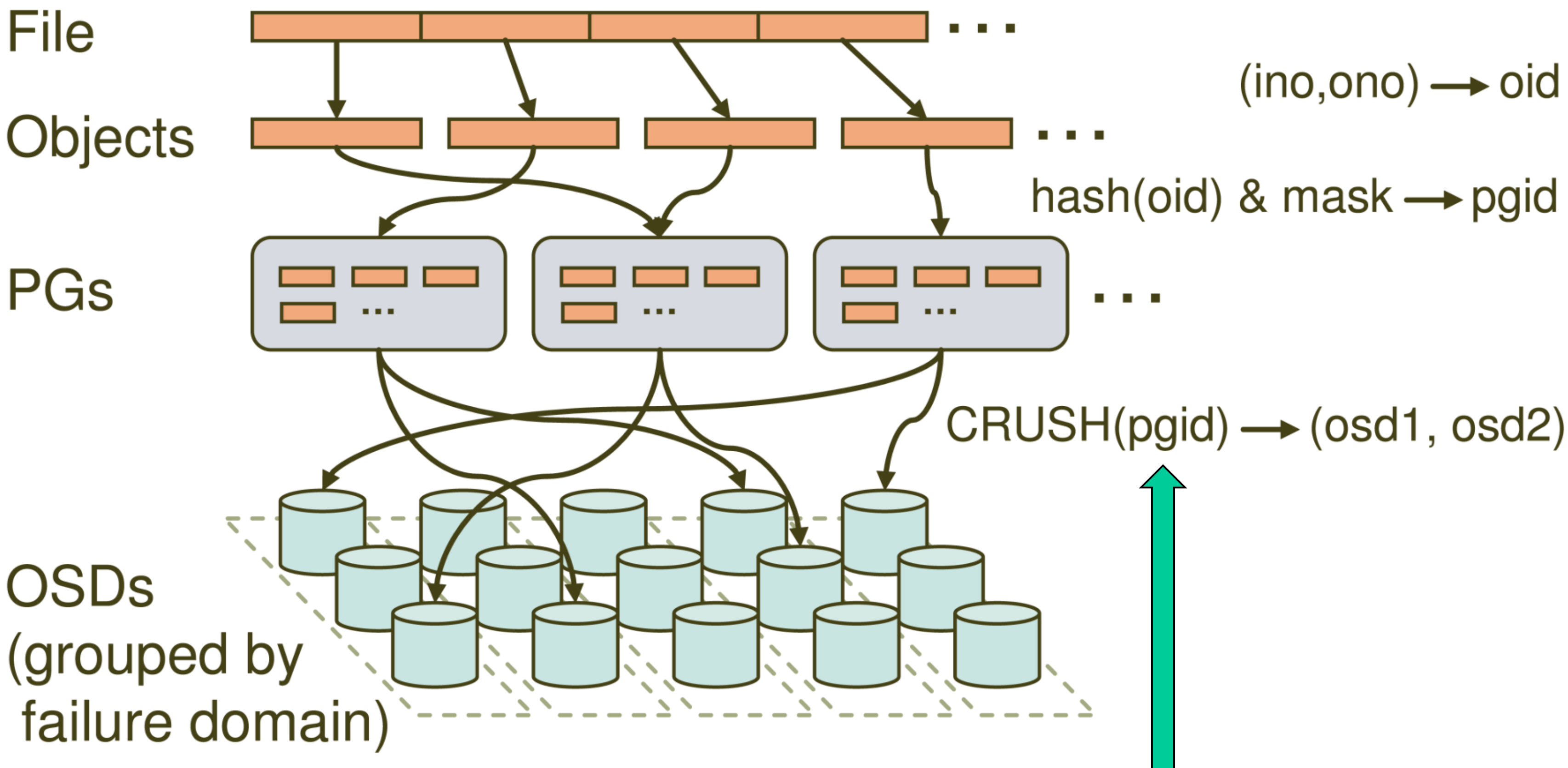
# Distributed Object Storage

Files are split across objects

Objects are members of placement groups

Placement groups are distributed across OSDs.

# Distributed Object Storage




CRUSH takes the placement group and an *OSD cluster map*: a compact, hierarchical description of the devices comprising the storage cluster.

# CRUSH

CRUSH(x) → ($osd_{n1}$, $osd_{n2}$, $osd_{n3}$)

- Inputs
  - x is the placement group
  - Hierarchical cluster map
  - Placement rules
- Outputs a list of OSDs

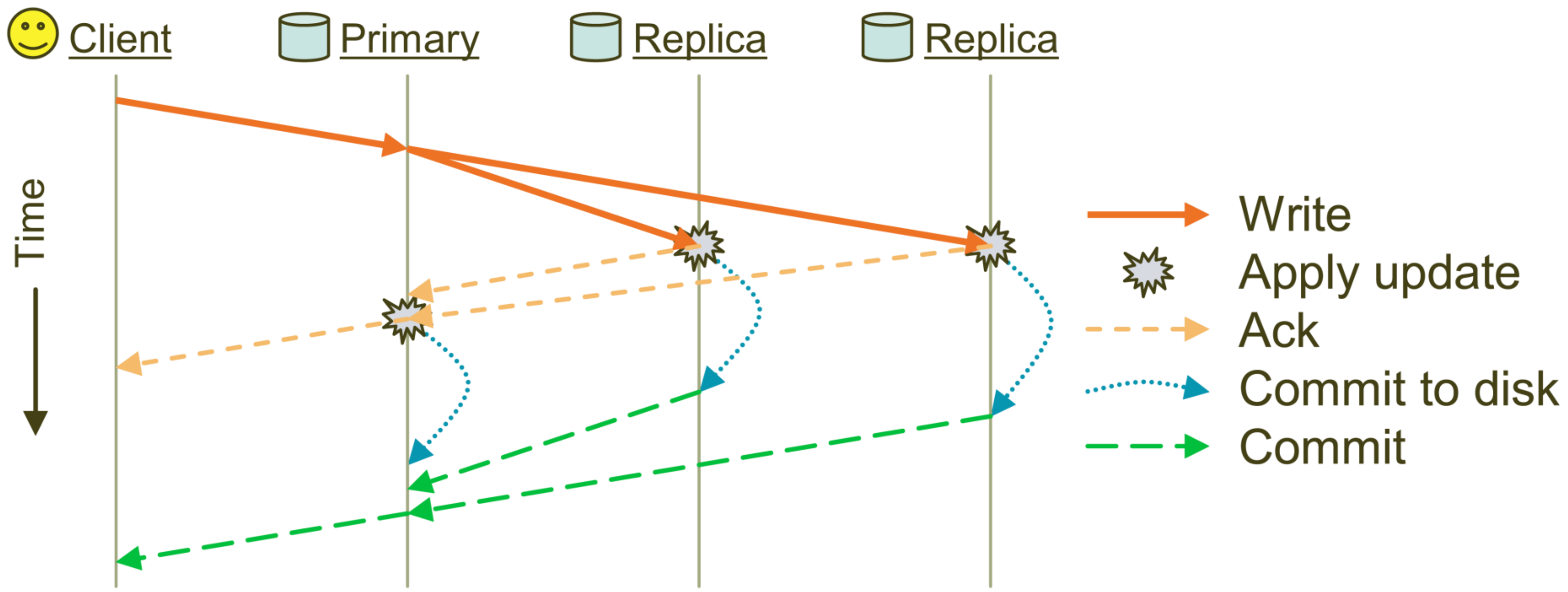
Advantages

- Anyone can calculate object location
- Cluster map infrequently updated

# Replication

Objects are replicated on OSDs in terms of placement groups, each of which is mapped to an ordered list of n OSDs (for n-way replication).

Client is oblivious to replication



# Acronyms

**CRUSH**: Controlled Replication Under Scalable Hashing
**EBOFS**: Extent and B-tree based Object File System
**HPC**: High Performance Computing
**MDS**: MetaData server
**OSD**: Object Storage Device
**PG**: Placement Group
**POSIX**: Portable Operating System Interface for uniX
**RADOS**: Reliable Autonomic Distributed Object Store

# Summary of Ceph

- Scalability, Reliability, Performance
- Separation of data and metadata

  -- CRUSH data distribution function
- Object based storage