
# Big Data and Internet Thinking

Chentao Wu
Associate Professor
Dept. of Computer Science and Engineering
wuct@cs.sjtu.edu.cn

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

# Download lectures

- ftp://public.sjtu.edu.cn
- User: wuct
- Password: wuct123456

- http://www.cs.sjtu.edu.cn/~wuct/bdit/

# Schedule

- lec1: Introduction on big data, cloud computing & IoT
- lec2: Parallel processing framework (e.g., MapReduce)
- lec3: Advanced parallel processing techniques (e.g., YARN, Spark)
- lec4: Cloud & Fog/Edge Computing
- lec5: Data reliability & data consistency
- lec6: Distributed file system & objected-based storage
- lec7: Metadata management & NoSQL Database
- lec8: Big Data Analytics

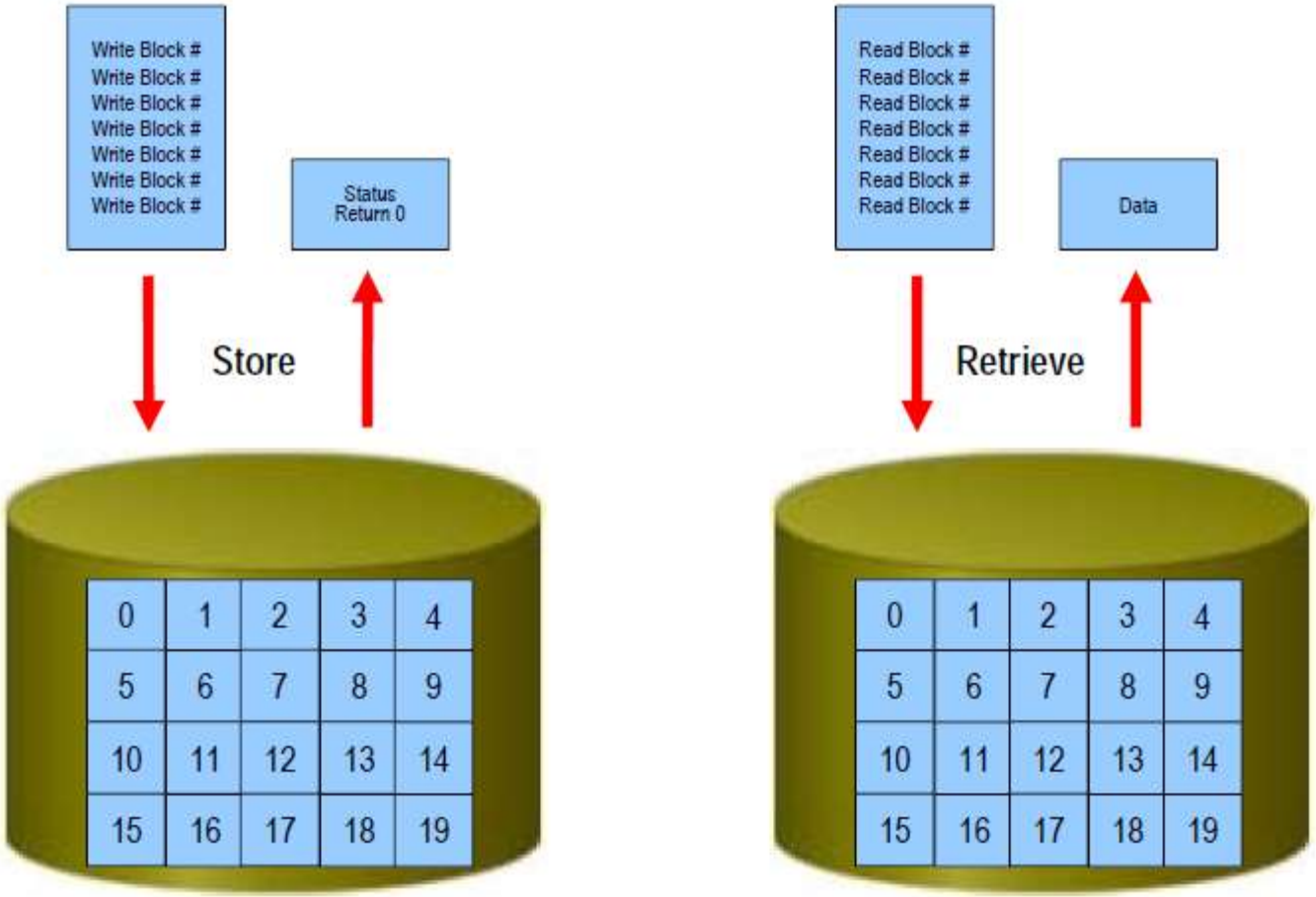
# Collaborators

Google 谷歌

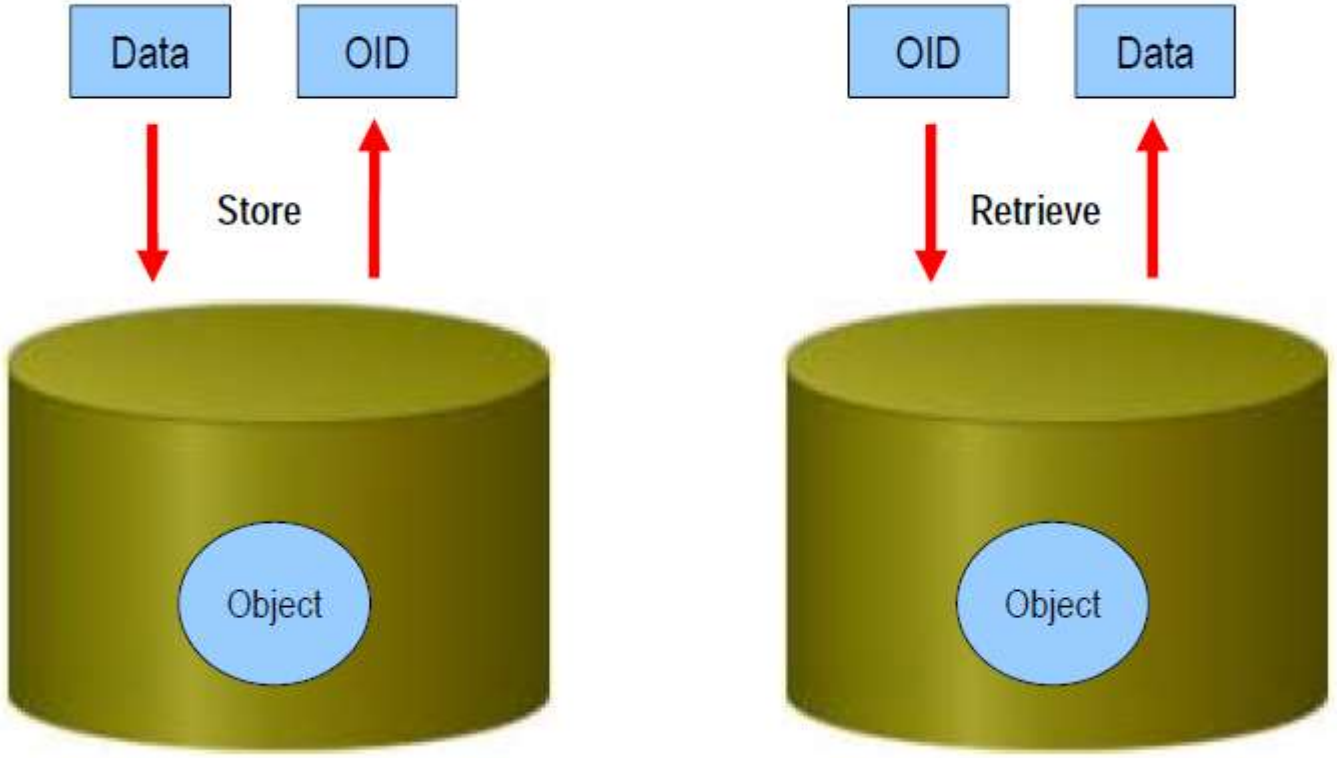
阿里云 aliyun.com

DELL EMC

# The Block Paradigm

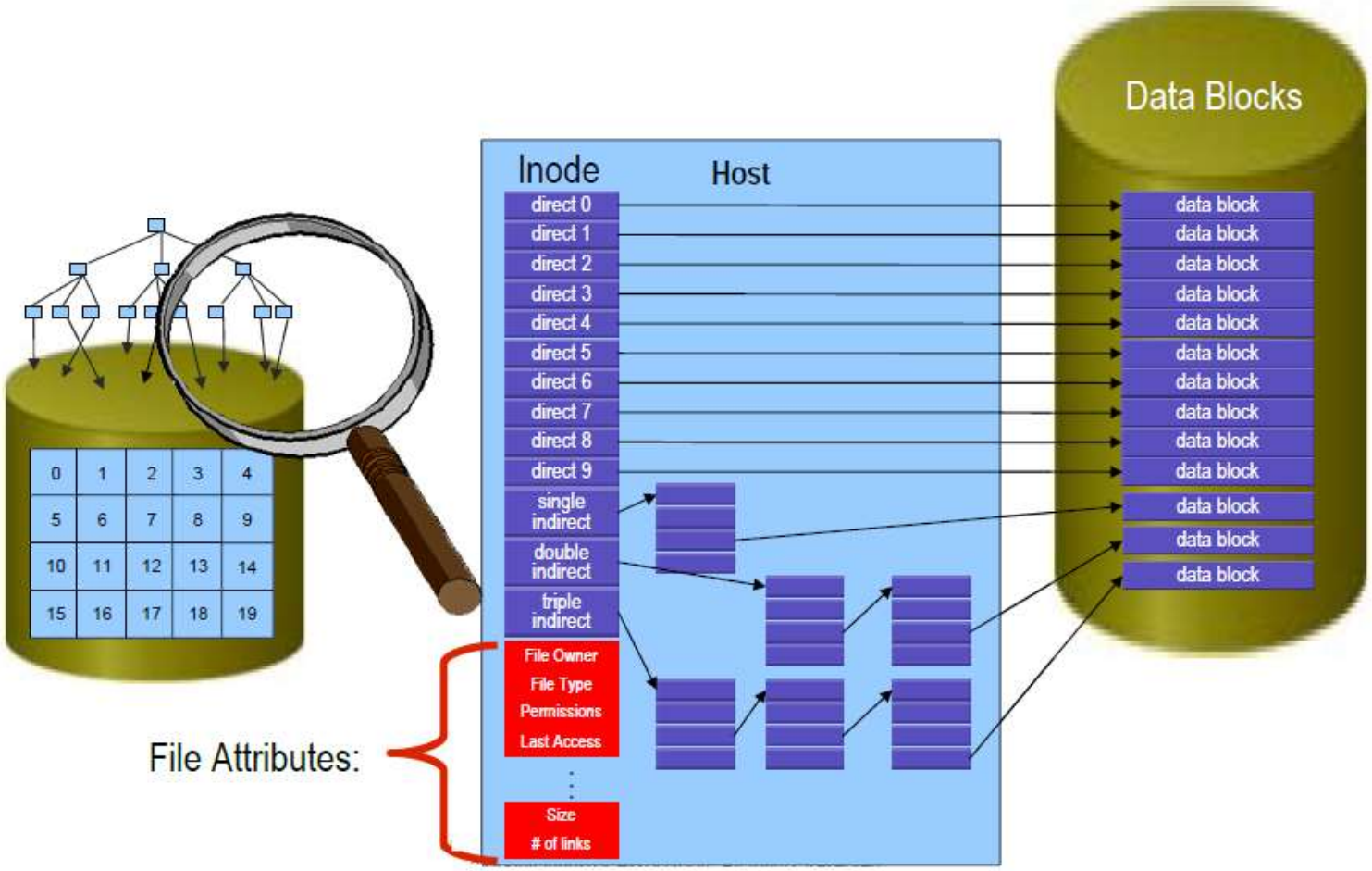


# The Object Paradigm

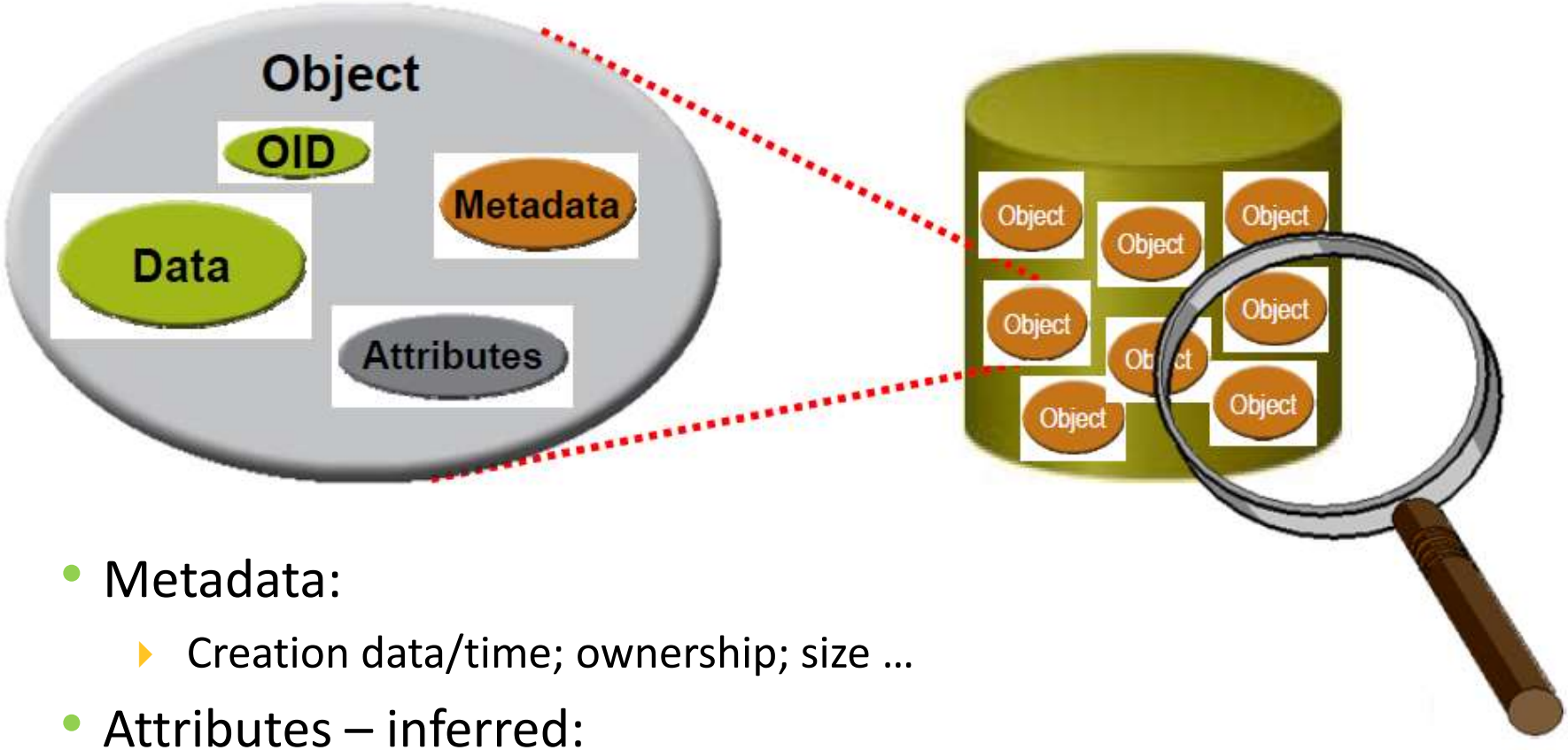


# File Access via Inodes

- Inodes contain file attributes

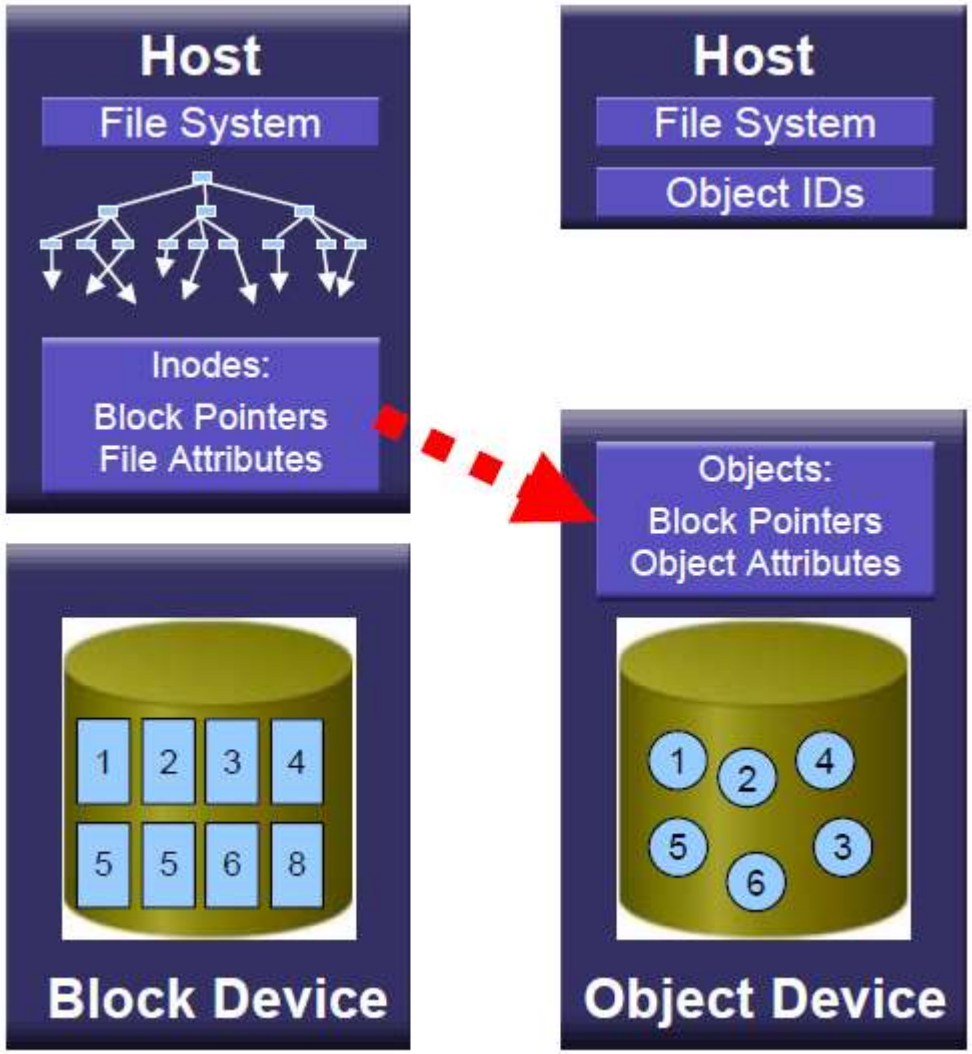


# Object Access




- Metadata:
  - Creation data/time; ownership; size …
- Attributes – inferred:
  - Access patterns; content; indexes …
- Attributes – user supplied:
  - Retention; QoS …

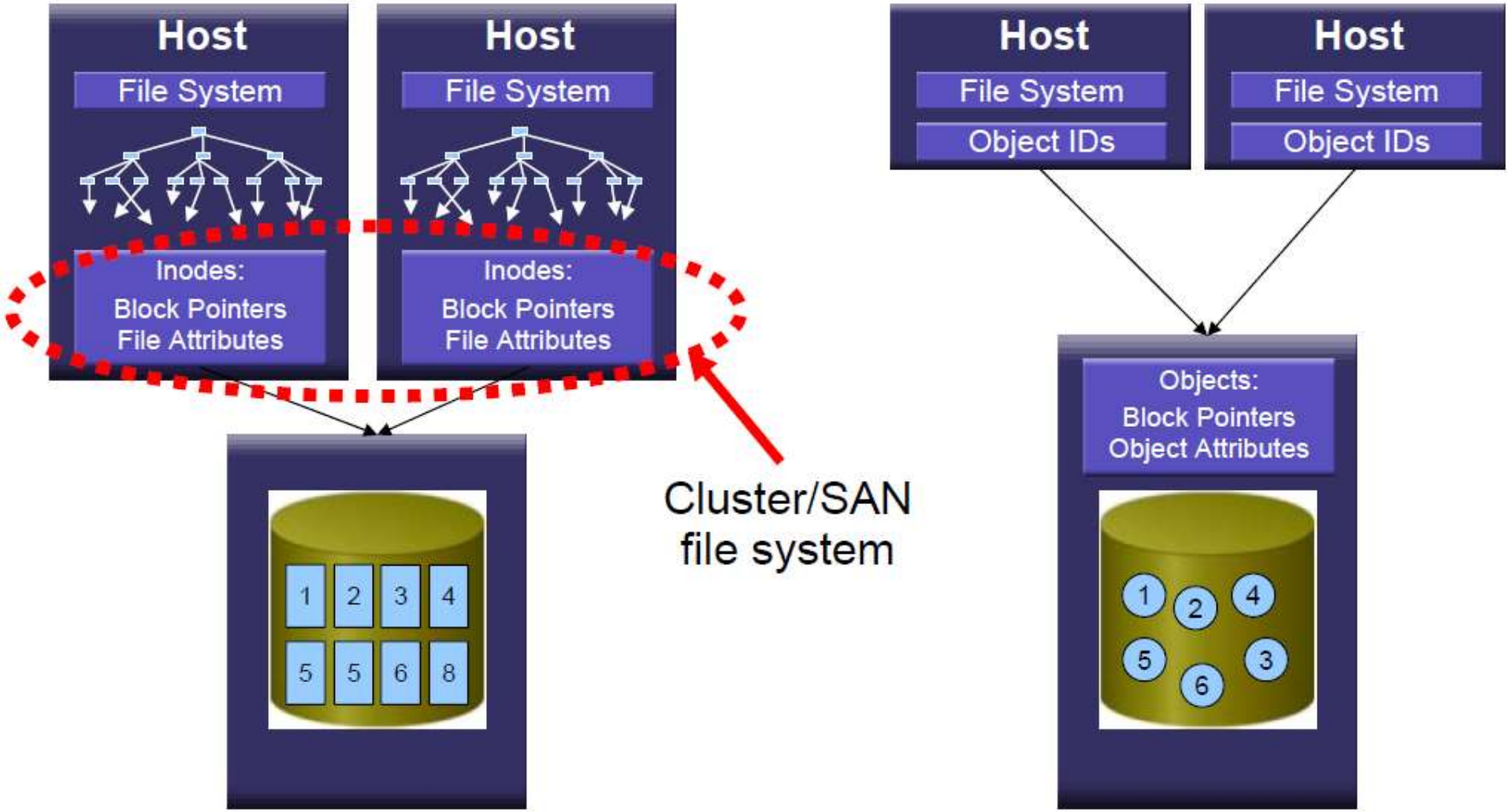
# Object Autonomy




- Storage becomes autonomous
  - Capacity planning
  - Load balancing
  - Backup
  - QoS, SLAs
  - Understand data/object grouping
  - Aggressive prefetching
  - Thin provisioning
  - Search
  - Compression/Deduplication
  - Strong security, encryption
  - Compliance/retention
  - Availability/replication
  - Audit
  - Self healing

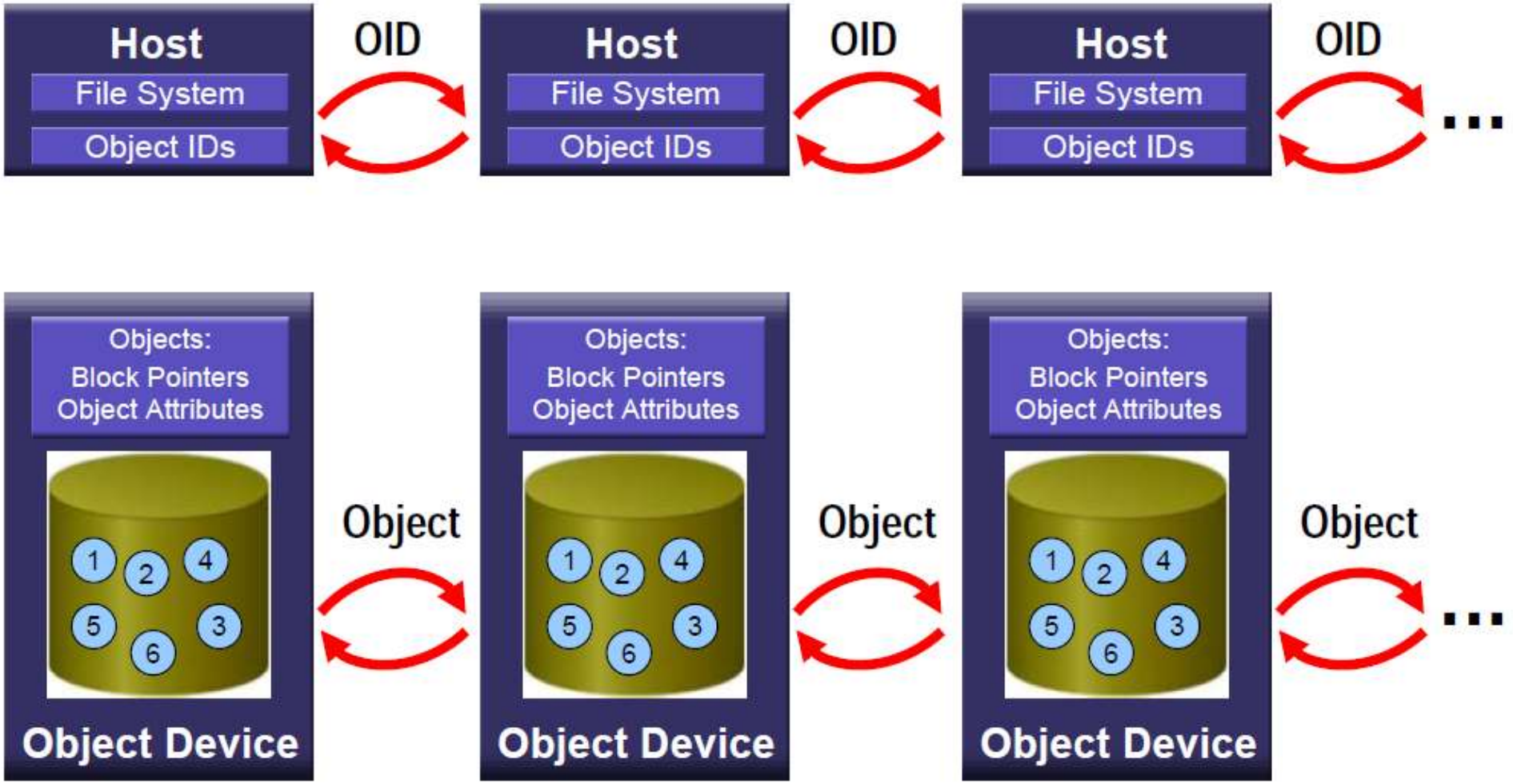
# Data Sharing

## homogeneous/heterogeneous

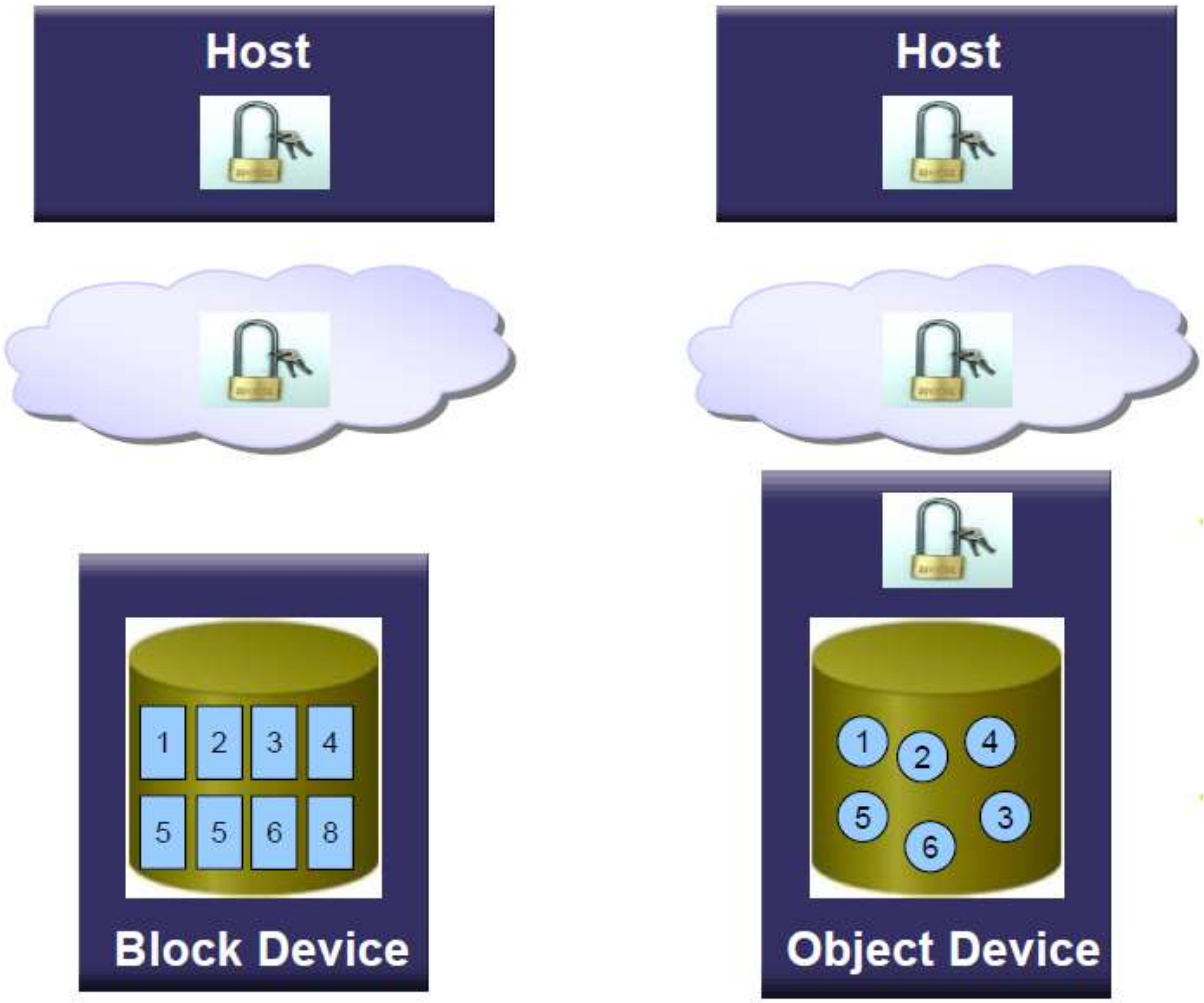


# Data Migration

## homogeneous/heterogeneous



# Strong Security Additional layer

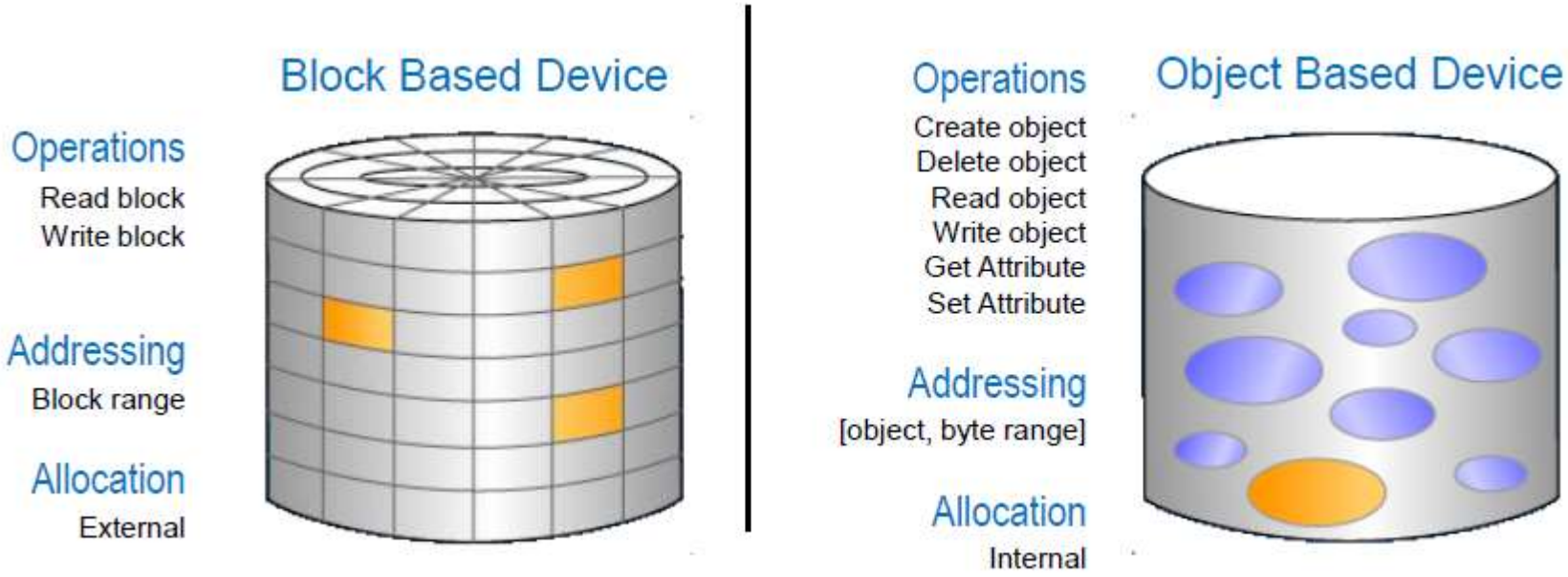



- Strong security via external service
  - Authentication
  - Authorization
  - …
- Fine granularity
  - Per object

# 2 Object-based Storage Devices

# Data Access (Block-based vs. Object-based Device)

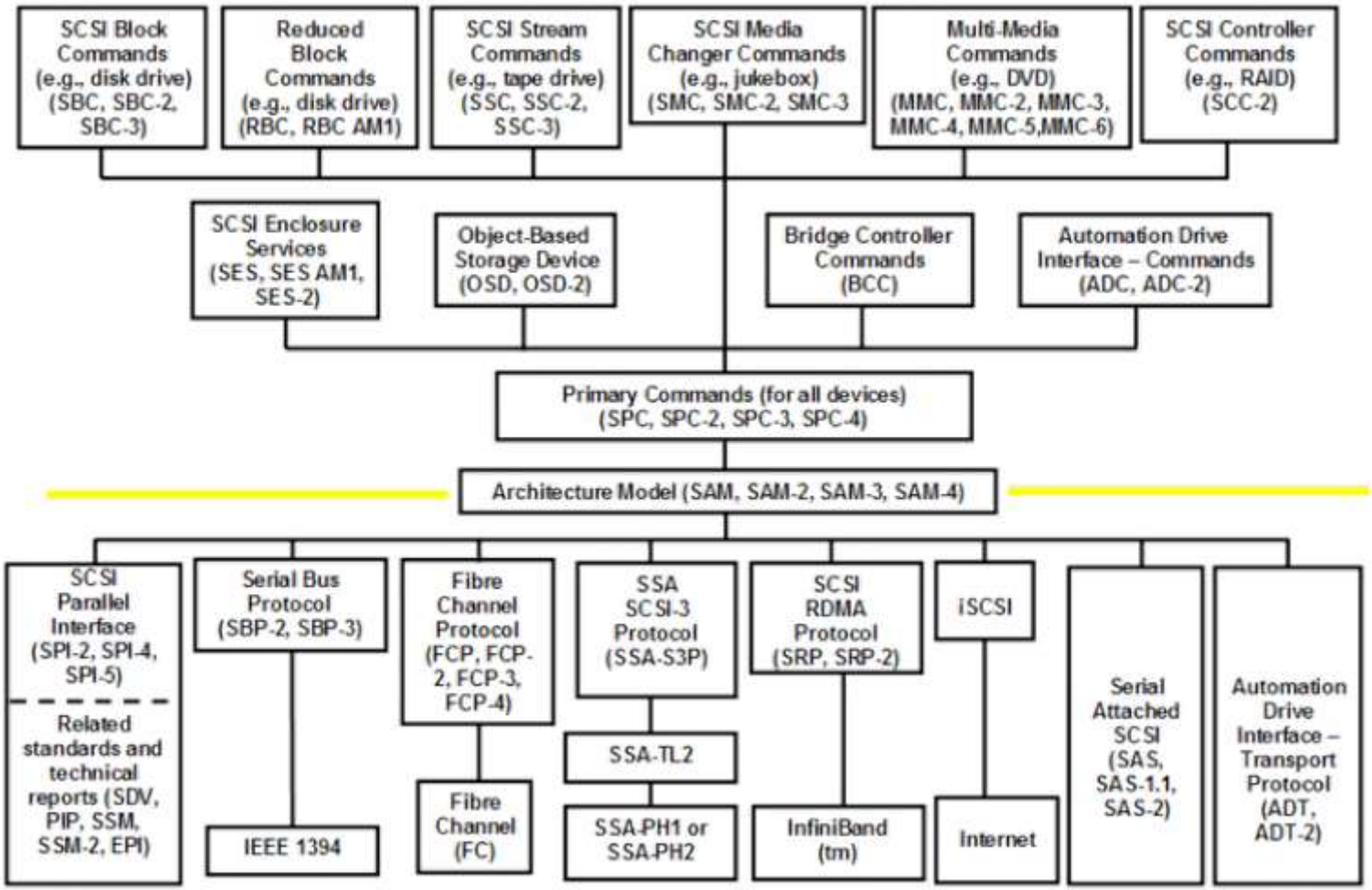
- Objects contain both data and attributes
  - Operations: create/delete/read/write objects, get/set attributes



# OSD Standards (1)

- ANSI INCITS T10 for OSD (the SCSI Specification, www.t10.org)
  - ANSI INCITS 458
  - OSD-1 is basic functionality
    - Read, write, create objects and partitions
    - Security model, Capabilities, manage shared secrets and working keys
  - OSD-2 adds
    - Snapshots
    - Collections of objects
    - Extended exception handling and recovery
  - OSD-3 adds
    - Device to device communication
    - RAID-[1,5,6] implementation between/among devices

# OSD Standards (2)



# OSD Forms

- Disk array/server subsystem
  - Example: custom-built HPC systems predominantly deployed in national labs
- Storage bricks for objects
  - Example: commercial supercomputing offering
- Object Layer Integrated in Disk Drive

# OSDs: like disks, only different

| | Disk | OSD |
|---|---|---|
| Model | Array of blocks<br>• Number never changes<br>• Size never changes | Objects<br>• Created and deleted<br>• Grow and shrink |
| Operations | Read/write disk blocks | Create/delete object<br>Read/write object blocks |
| Security | Zoning, LUN masking<br>•Applies to entire device | Capability-based<br>•Applies to each object and op |
| Typical transports | Fibre Channel, SCSI, iSCSI | iSCSI, **ONC-RPC over TCP/IP** |

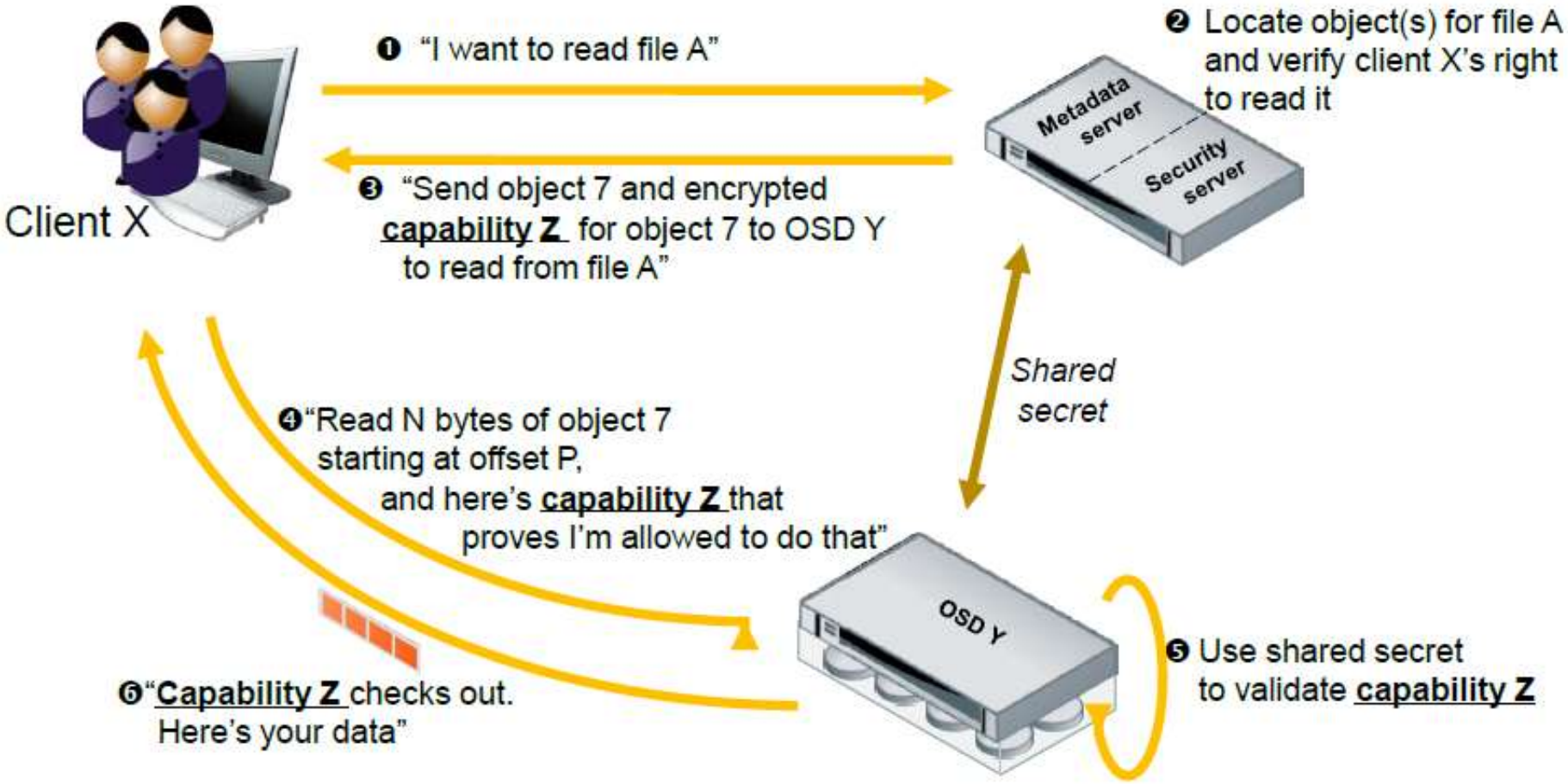
# OSDs: like a file server, only different

| | File server | OSD |
|---|---|---|
| Model | Files | Objects |
| Naming | Human-readable names in a hierarchical directory tree | Two level name space:<br>64 bit object "name" in a 64 bit partition "name" |
| Operations | File: create, delete, rename<br>File byte range: read, write, append, truncate | Object: create, delete,<br>Object block range: read, write, append, truncate |
| Security | User \| group \| world × `rwx` or access control lists<br>• Checked at initial file access | Digitally signed ***capabilities***<br>• Checked for every I/O request |

# OSD Capabilities (1)

- Unlike disks, where access is granted on an all or nothing basis, OSDs grant or deny access to individual objects based on **Capabilities**
- A Capability must accompany each request to read or write an object
  - Capabilities are cryptographically signed by the Security Manager and verified (and enforced) by the OSD
  - A Capability to access an object is created by the Security Manager, and given to the client (application server) accessing the object
  - Capabilities can be revoked by changing an attribute on the object

# OSD Capabilities (2)



# OSD Security Model

- OSD and File Server know a secret key
  - Working keys are periodically generated from a master key
- File server authenticates clients and makes access control policy decisions
  - Access decision is captured in a capability that is signed with the secret key
  - Capability identifies object, expire time, allowed operations, etc.
- Client signs requests using the capability signature as a signing key
  - OSD verifies the signature before allowing access
  - OSD doesn't know about the users, Access Control Lists (ACLs), or whatever policy mechanism the File Server is using

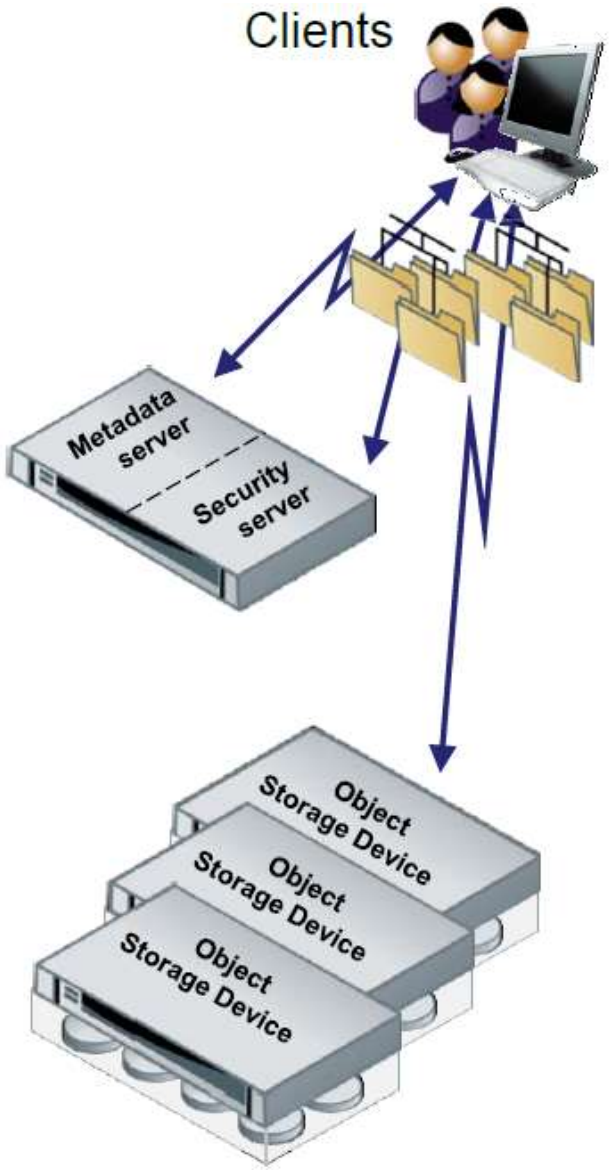
# 3 Object-based File Systems

# Why not just OSD = file system?

- Scaling
  - What if there's more data than the biggest OSD can hold?
  - What if too many clients access an OSD at the same time?
  - What if there's a file bigger than the biggest OSD can hold?
- Robustness
  - What happens to data if an OSD fails?
  - What happens to data if a Metadata Server fails?
- Performance
  - What if thousands of objects are access concurrently?
  - What if big objects have to be transferred really fast?

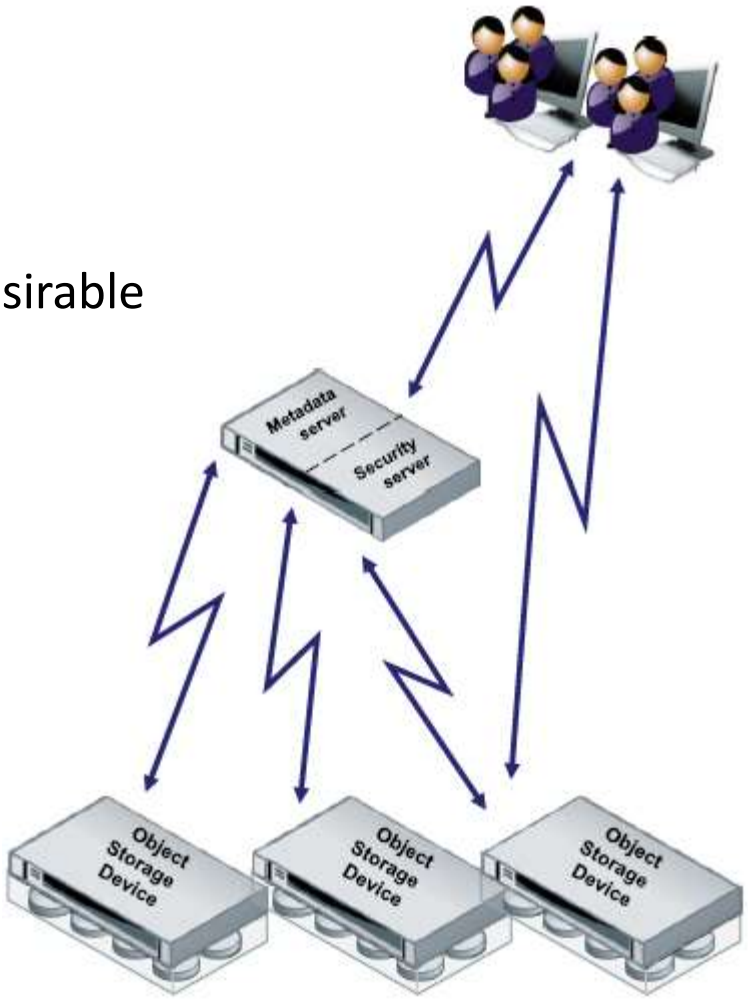
# General Principle

- Architecture
  - File = one or more groups of objects
    - Usually on different OSDs
  - Clients access Metadata Servers to locate data
  - Clients transfer data directly to/from OSDs
- Address
  - Capacity
  - Robustness
  - Performance



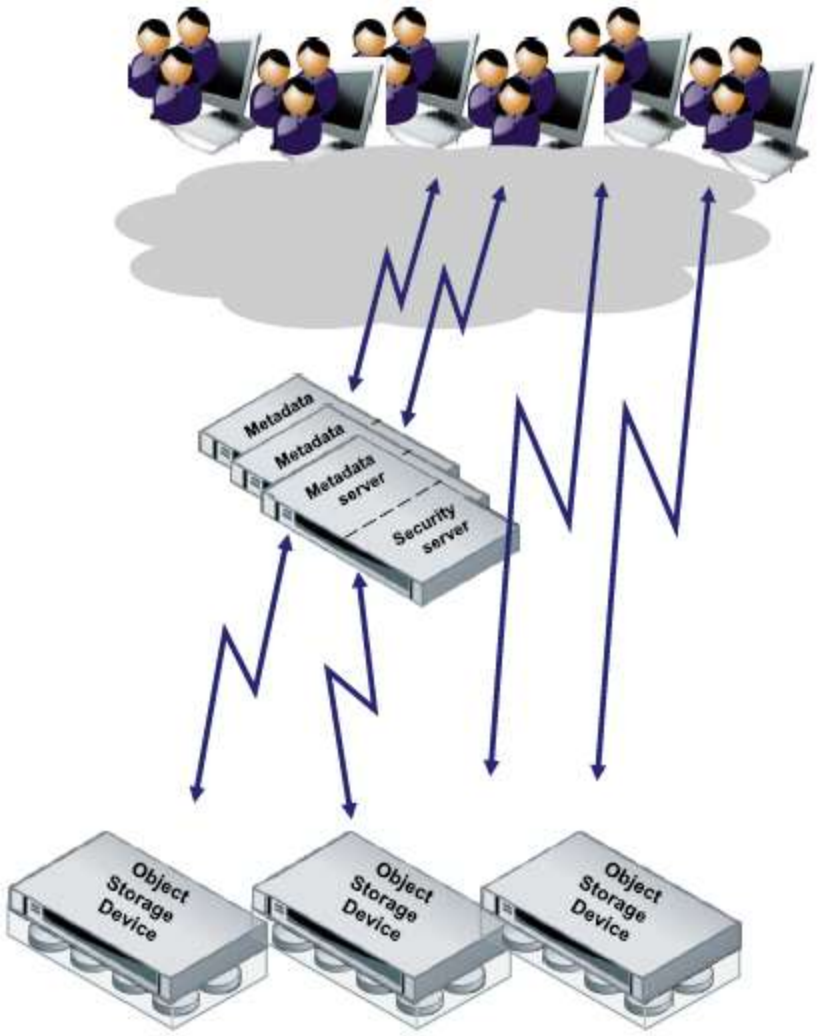
# Capacity

- Add OSDs
  - Increase total system capacity
  - Support bigger files
    - Files can span OSDs if necessary or desirable



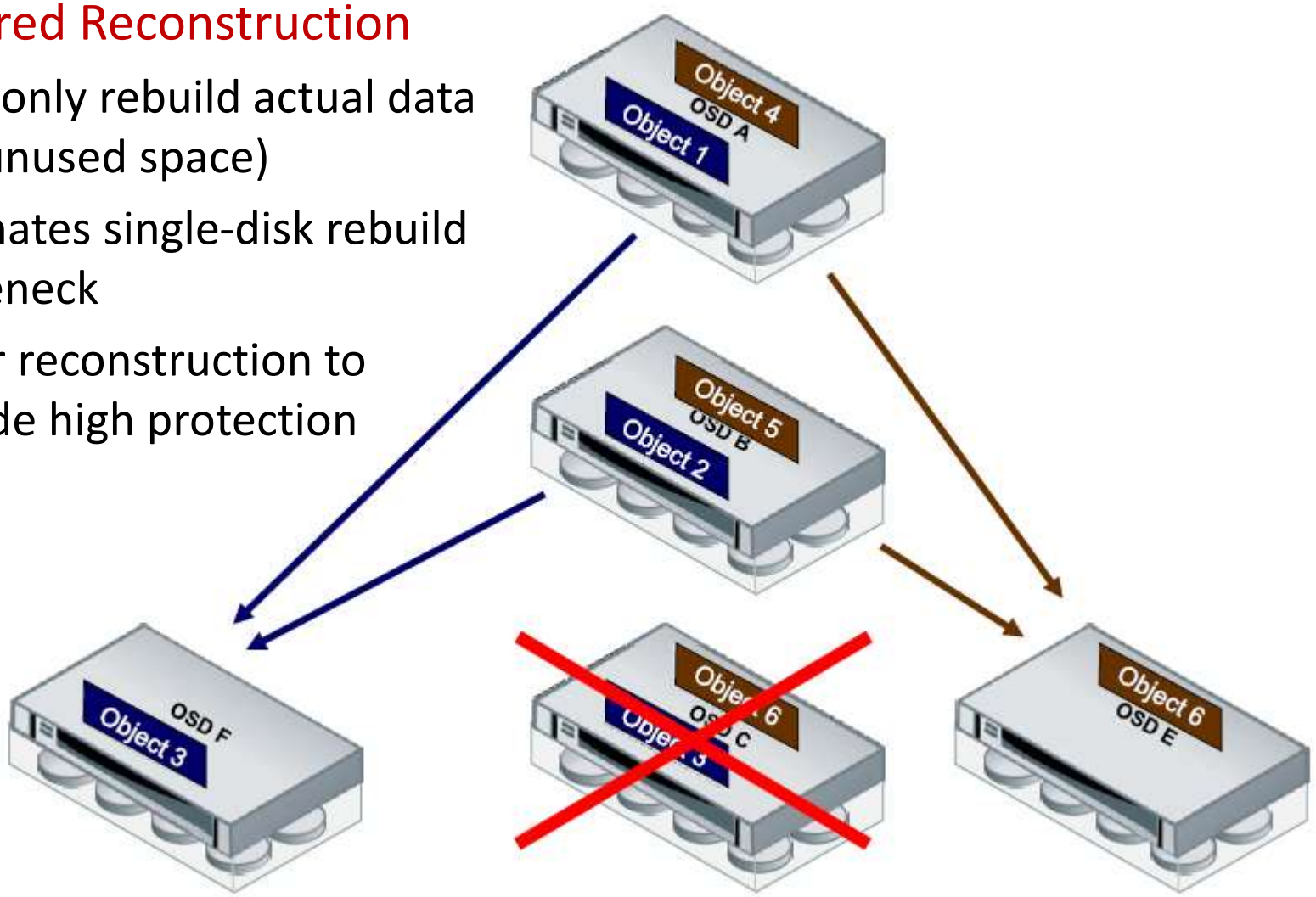
# Robustness

- Add metadata servers
  - Resilient metadata services
  - Resilient security services
- Add OSDs
  - Failed OSD affects small percentage of system resources
  - Inter-OSD mirroring and RAID
  - Near-online file system checking



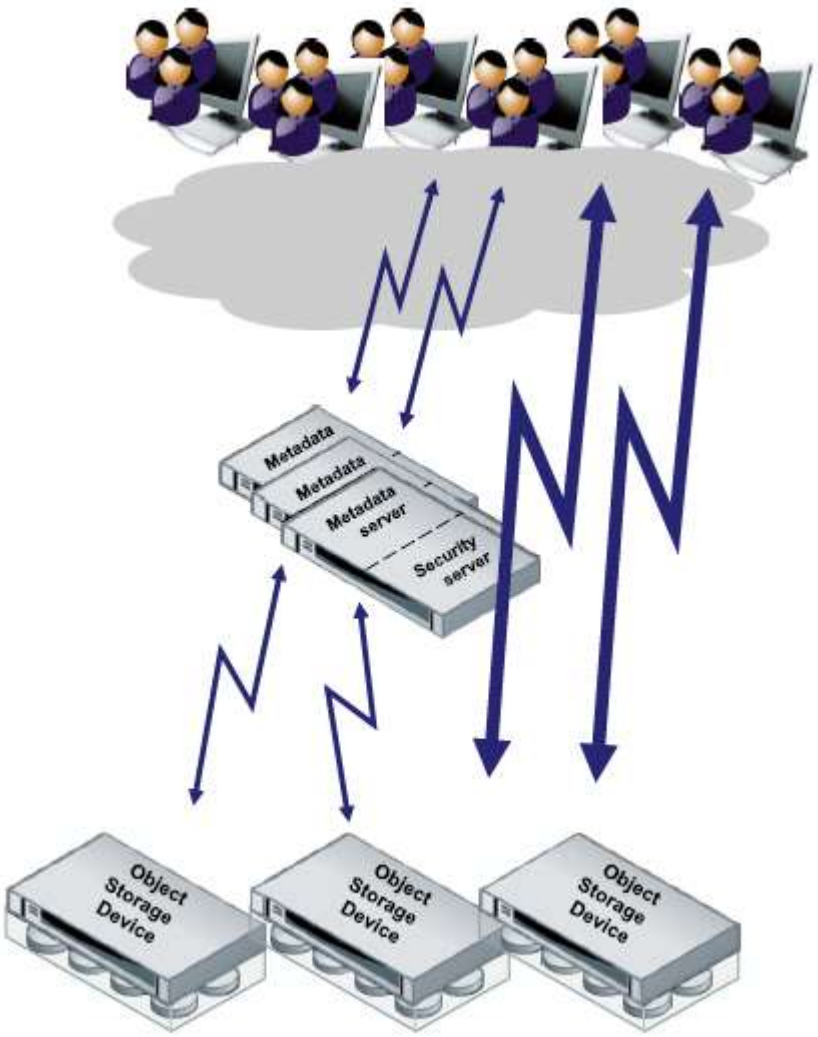
# Advantage of Reliability

- Declustered Reconstruction
  - OSDs only rebuild actual data (not unused space)
  - Eliminates single-disk rebuild bottleneck
  - Faster reconstruction to provide high protection



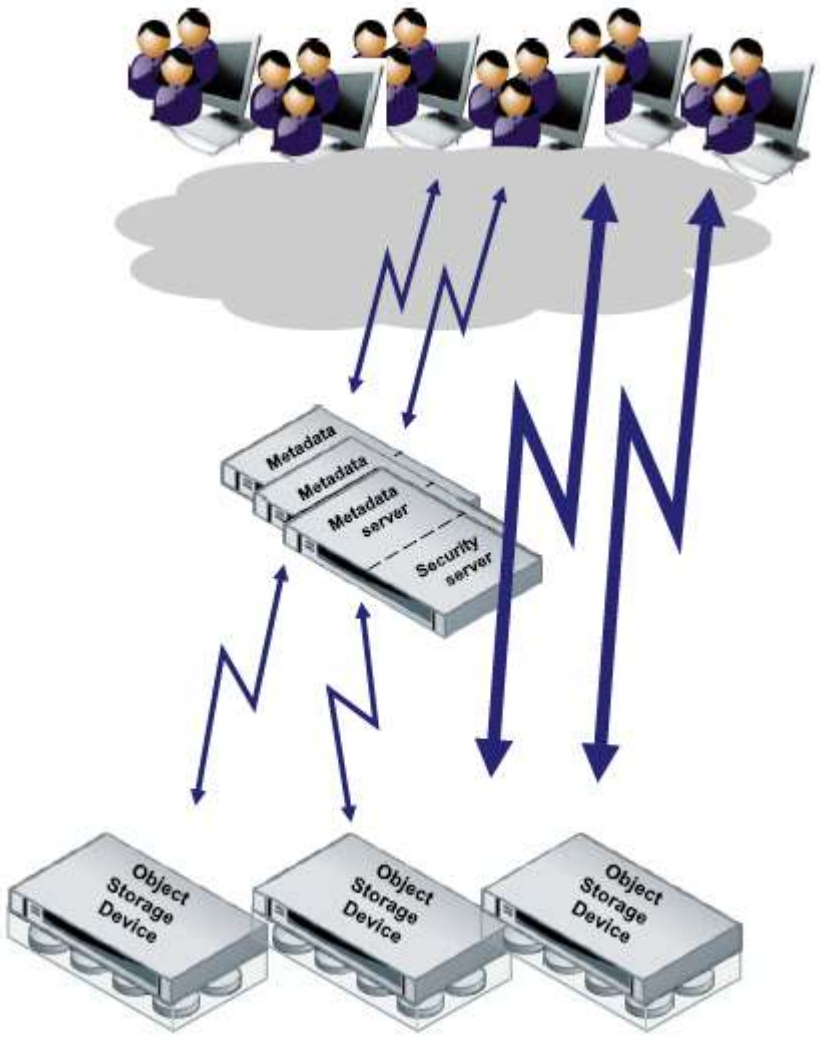
# Performance

- Add metadata servers
  - More concurrent metadata operations
    - Getattr, Readdir, Create, Open, …
- Add OSDs
  - More concurrent I/O operations
  - More bandwidth directly between clients and data

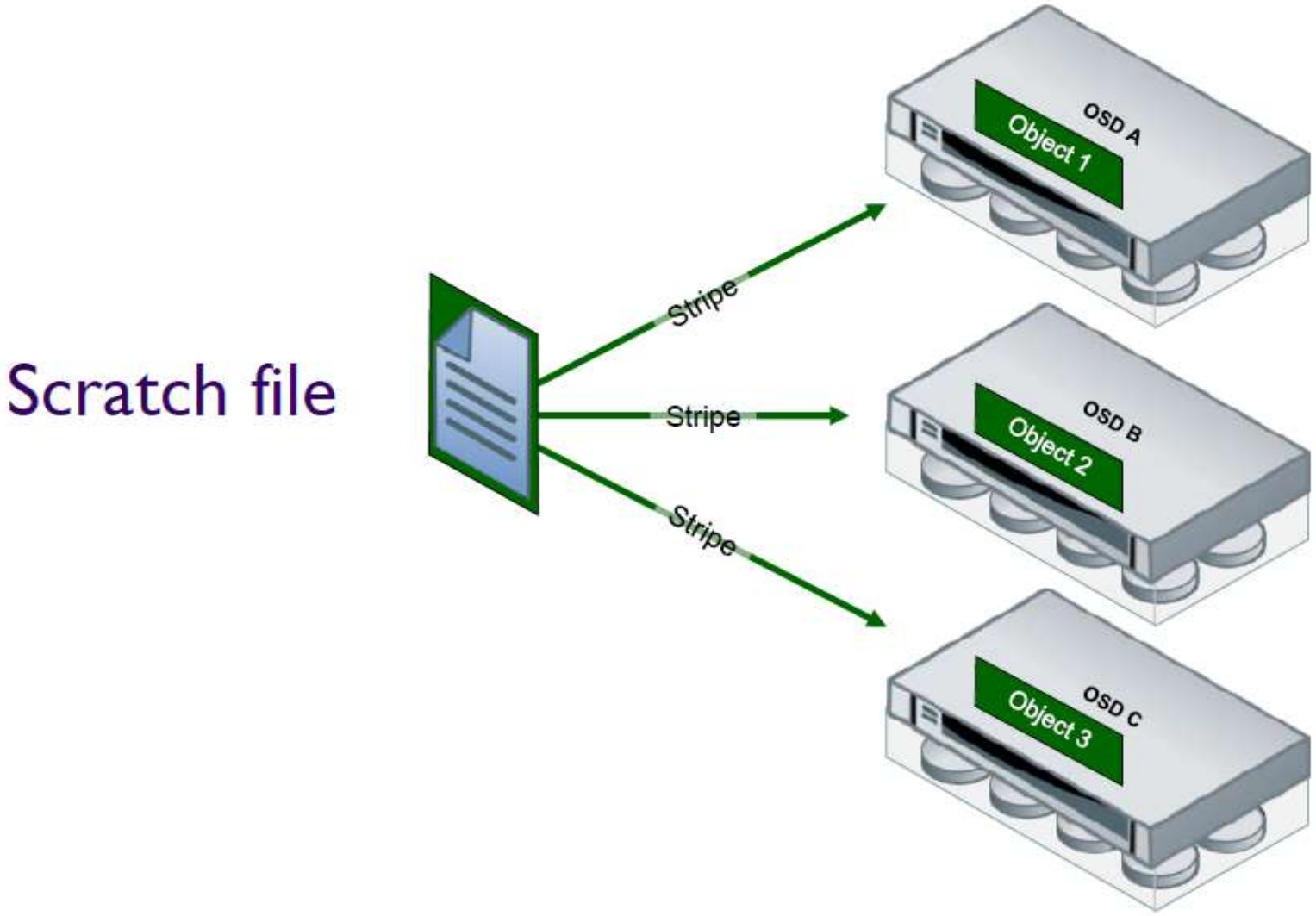


# Additional Advantages

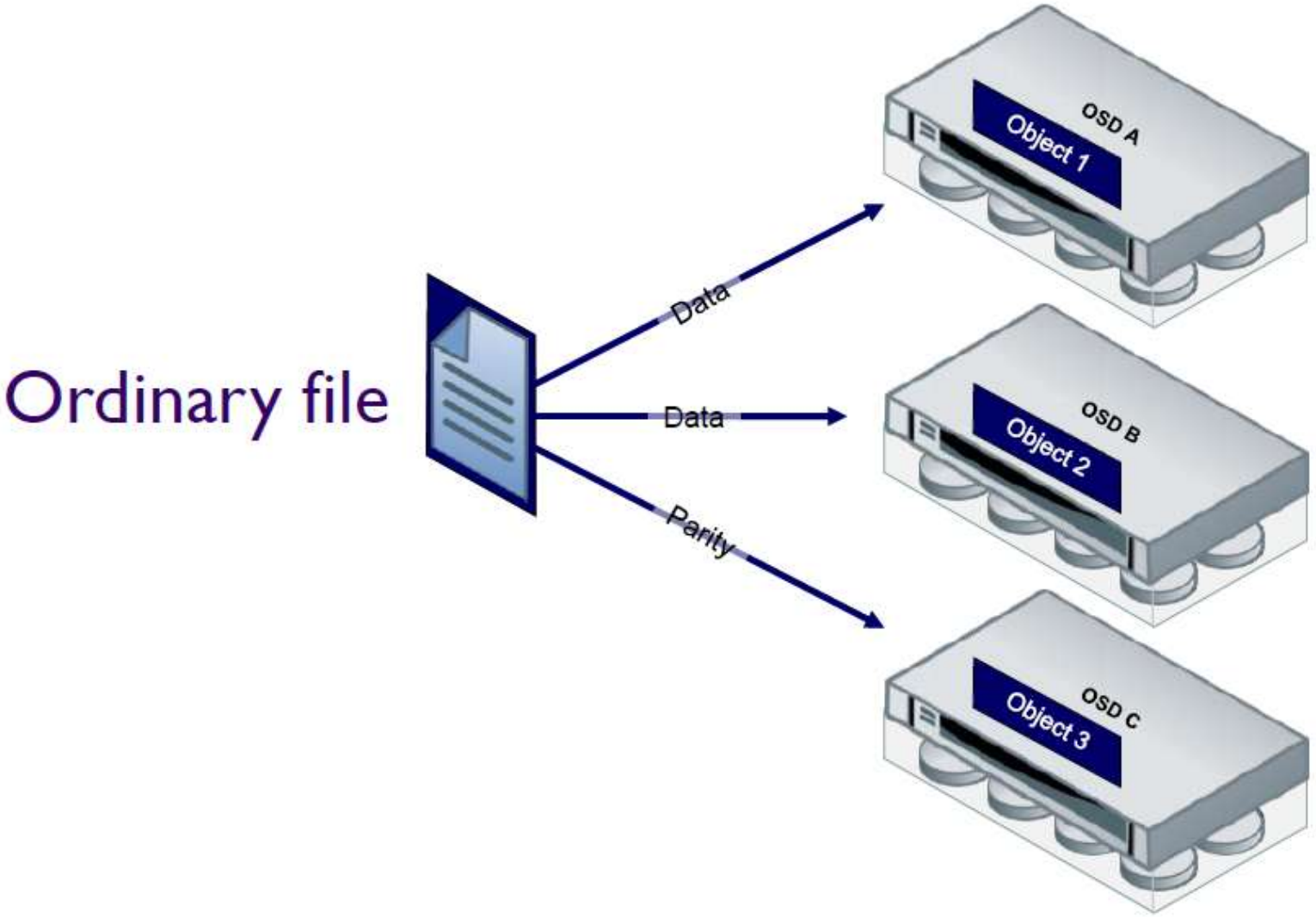
- Optimal data placement
  - Within OSD: proximity of related data
  - Load balancing across OSDs
- System-wide storage pooling
  - Across multiple file systems
- Storage tiering
  - Per-file control over performance and resiliency



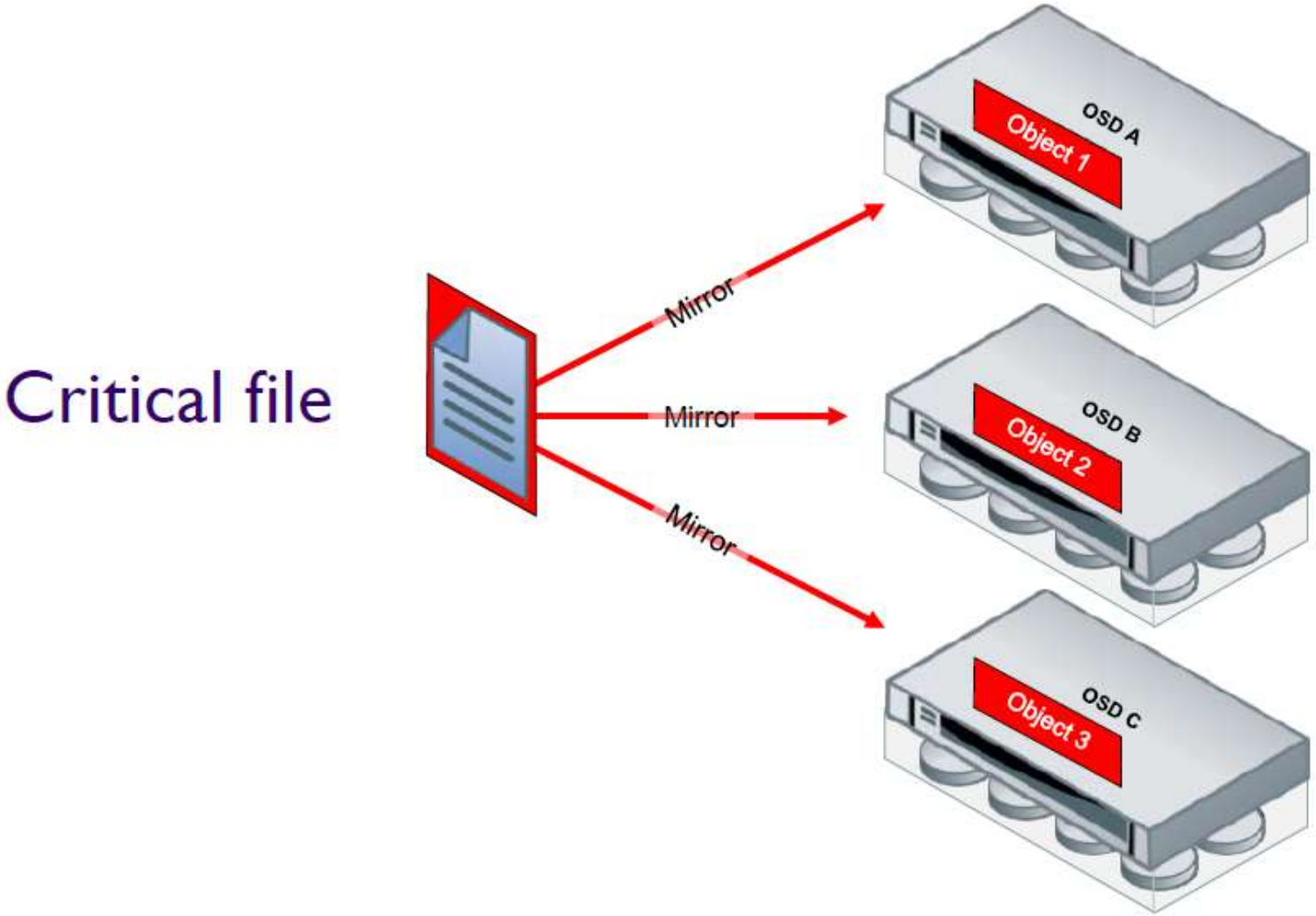
# Per-file tiering in OSDs: striping



# Per-file tiering in OSDs: RAID-4/5/6

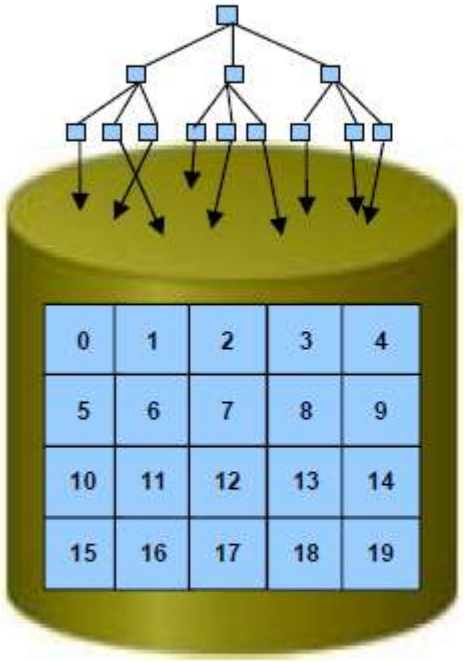


# Per-file tiering in OSDs: mirroring(RAID-1)



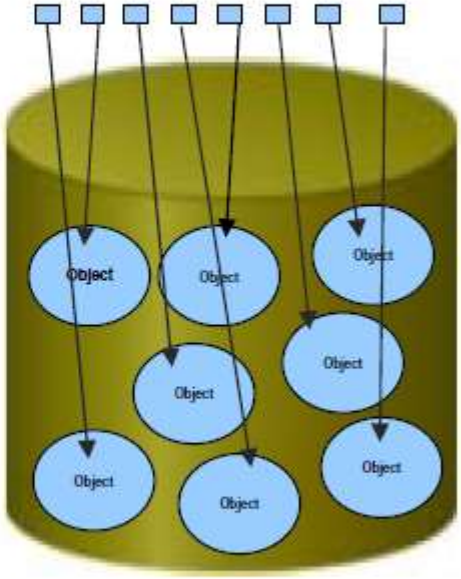
# Flat namespace

File names / inodes



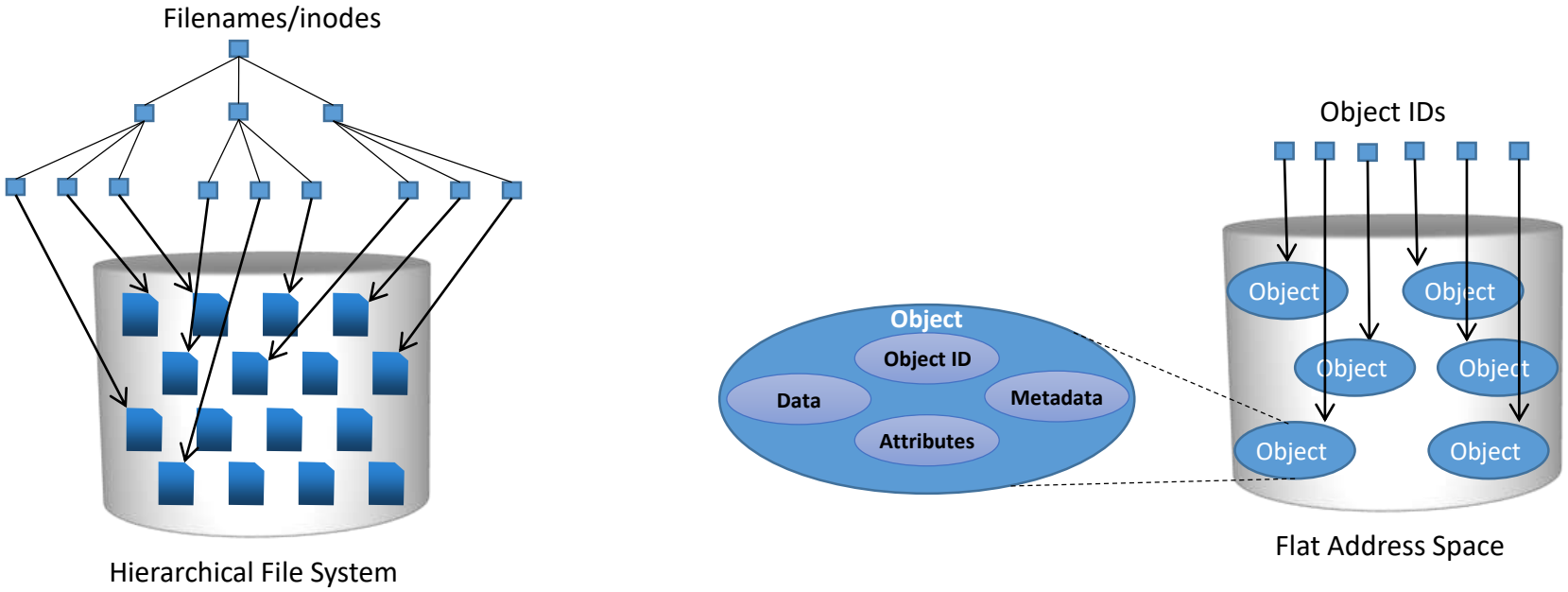

Traditional
Hierarchical

Objects / OIDs



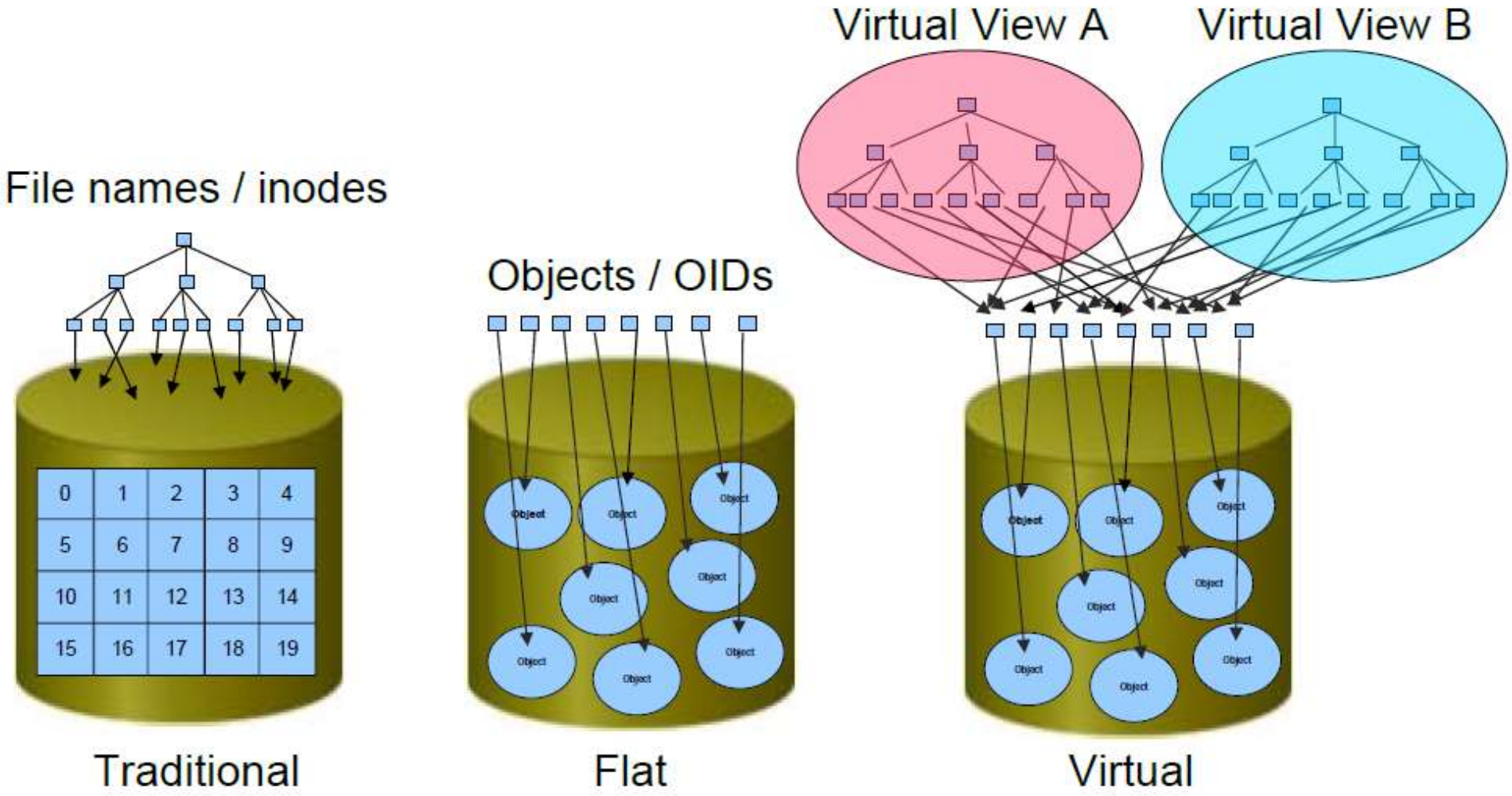

Flat

# Hierarchical File System Vs. Flat Address Space

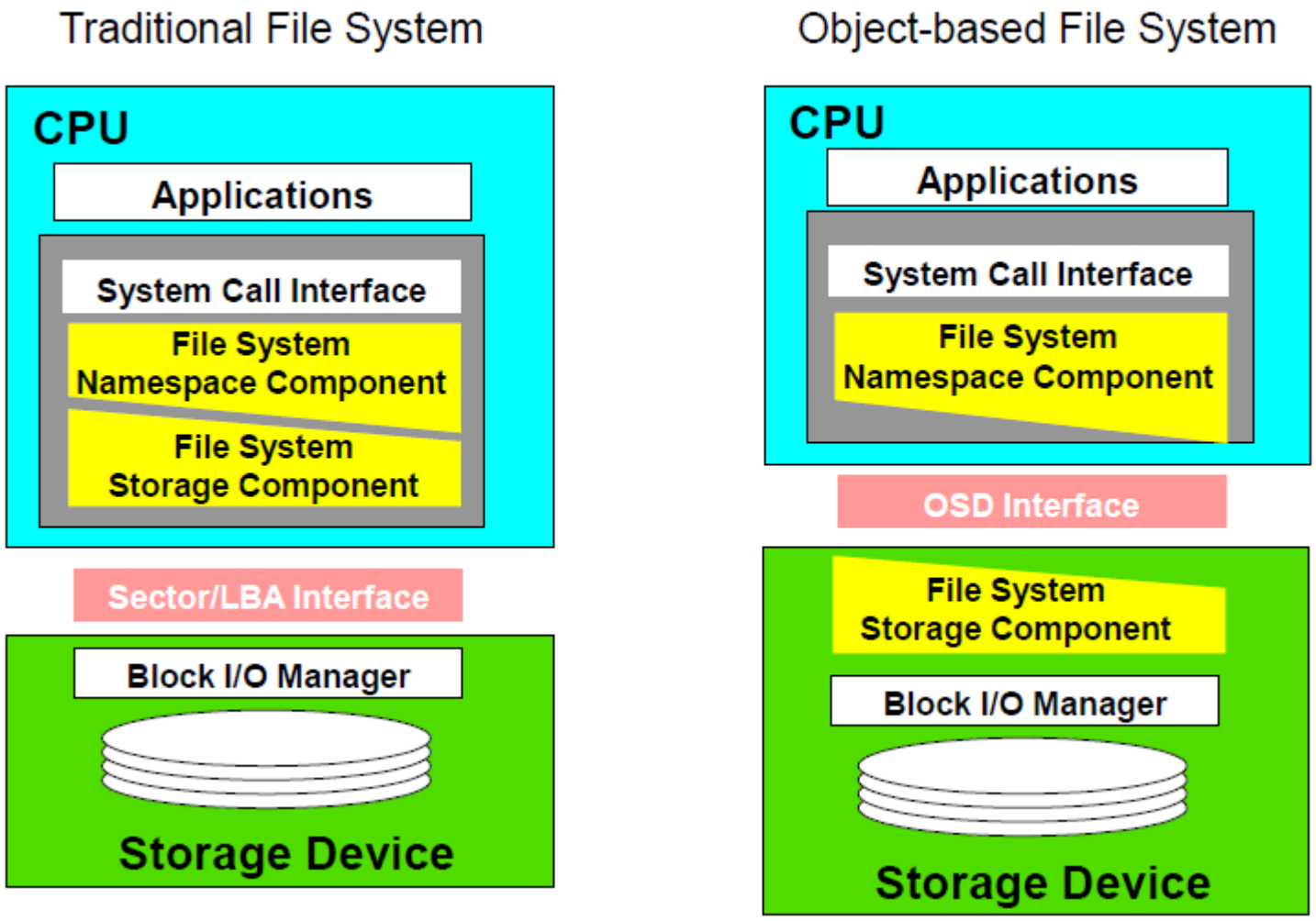



- Hierarchical file system organizes data in the form of files and directories
- Object-based storage devices store the data in the form of objects
  - It uses flat address space that enables storage of large number of objects
  - An object contains user data, related metadata, and other attributes
  - Each object has a unique object ID, generated using specialized algorithm

# Virtual View / Virtual File Systems



# Traditional FS Vs. Object-based FS (1)

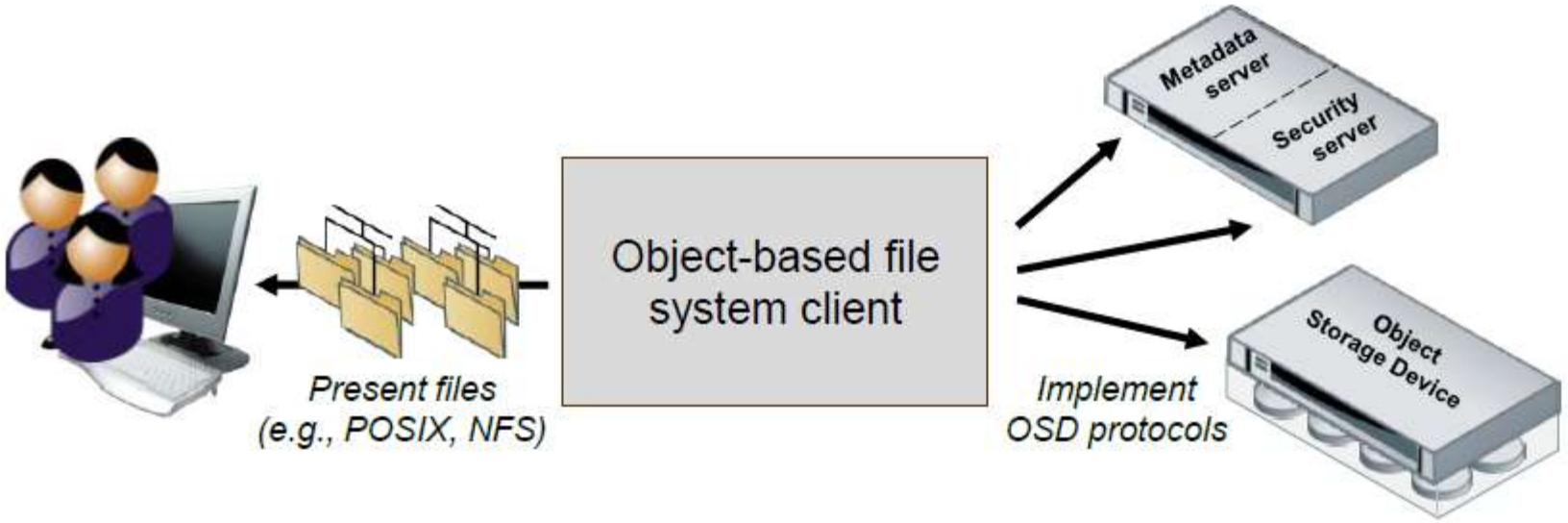


# Traditional FS Vs. Object-based FS (2)

- File system layer in host manages
  - Human readable namespace
  - User authentication, permission checking, Access Control Lists (ACLs)
  - OS interface
- Object Layer in OSD manages
  - Block allocation and placement
  - OSD has better knowledge of disk geometry and characteristic so it can do a better job of file placement/optimization than a host-based file system

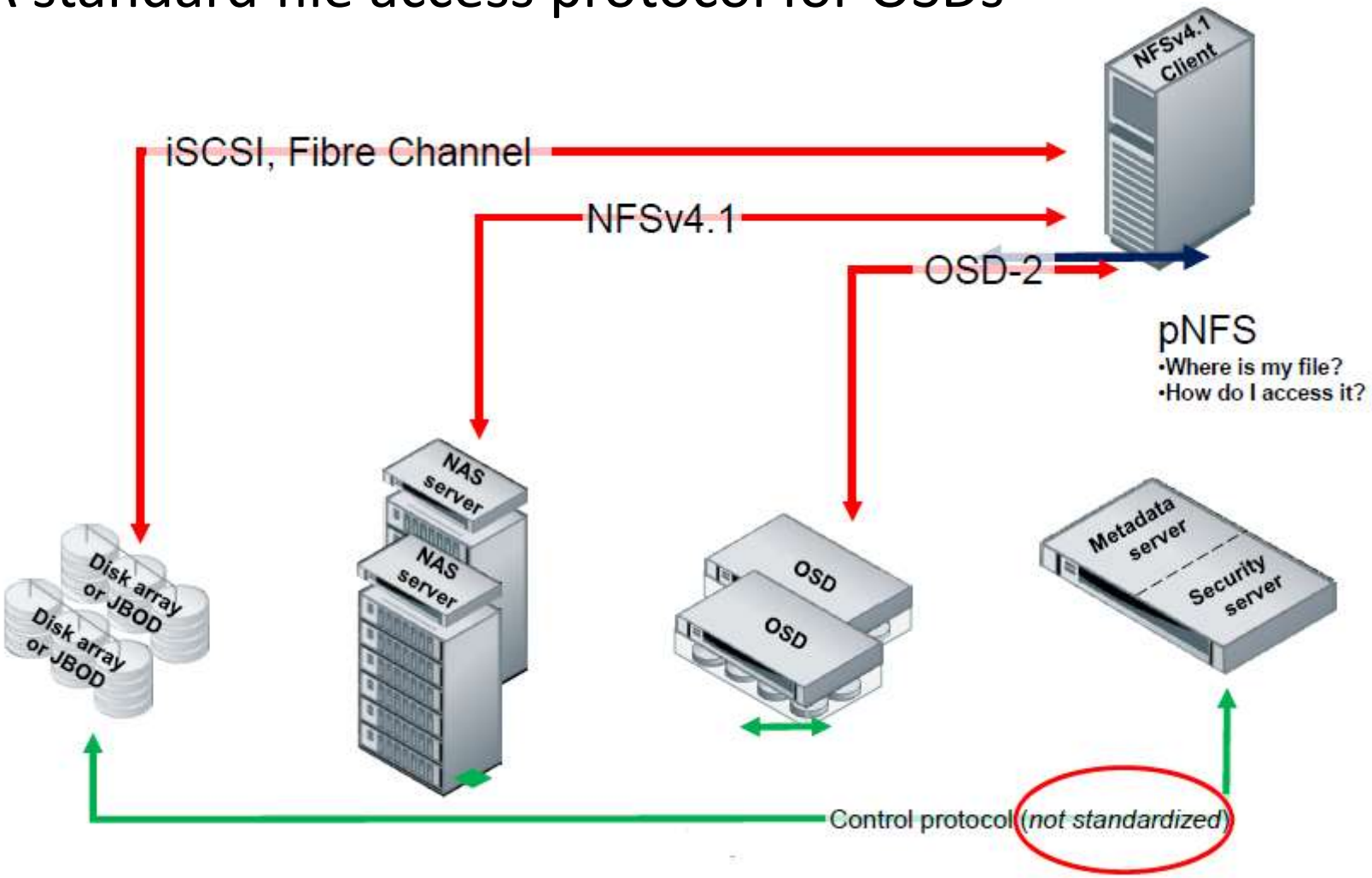
# Accessing Object-based FS

- Typical Access
  - SCSI (block), NFS/CIFS (file)
- Needs a client component
  - Proprietary
  - Standard

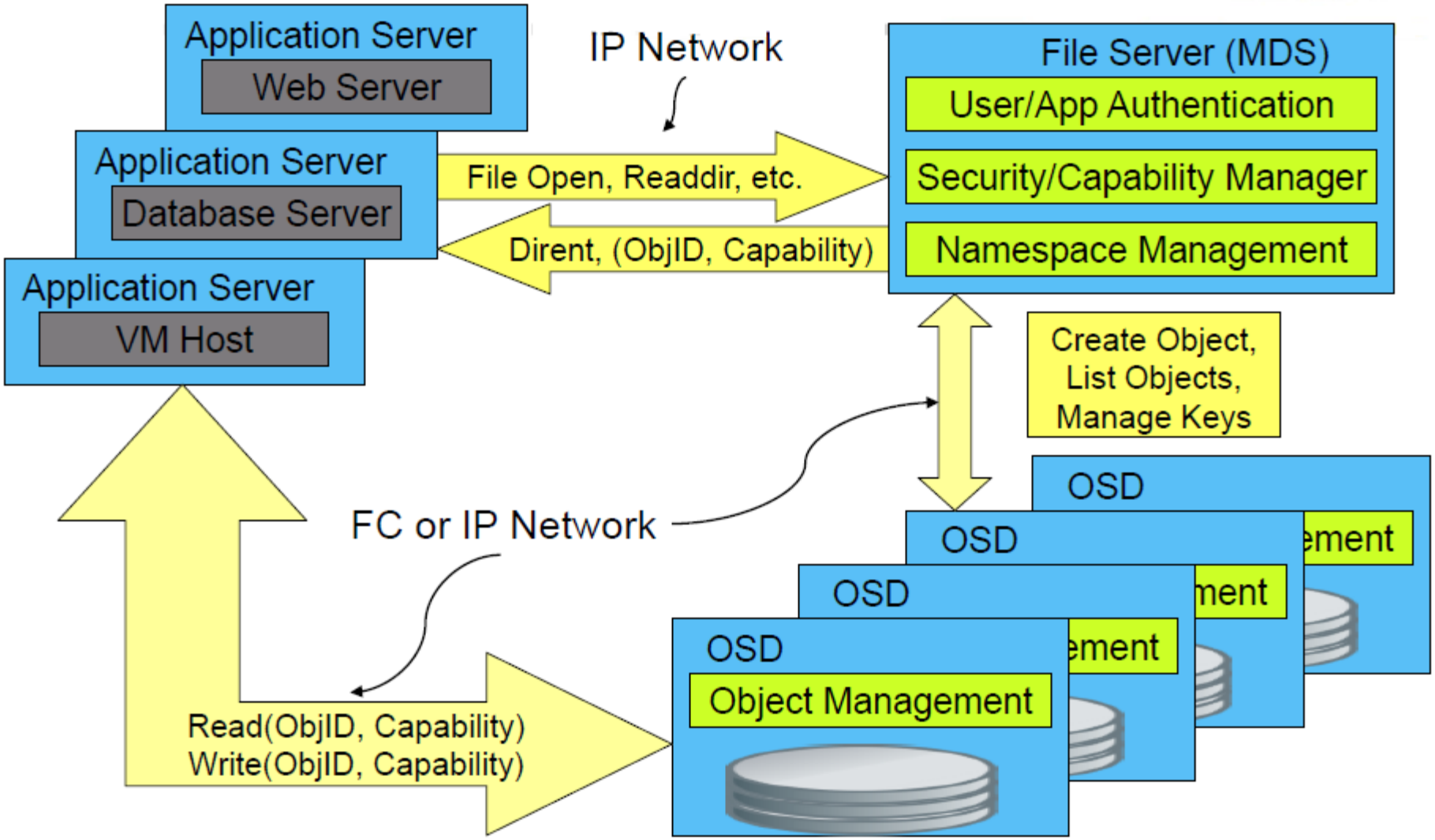


# Standard→ NFS v4.1

- A standard file access protocol for OSDs



# Scaling Object-based FS (1)



# Scaling Object-based FS (2)

- App servers (clients) have direct access to storage to read/write file data securely
  - Contrast with SAN where security is lacking
  - Contrast with NAS where server is a bottleneck
- File system includes multiple OSDs
  - Grow the file system by adding an OSD
  - Increase bandwidth at the same time
  - Can include OSDs with different performance characteristics (SSD, SATA, SAS)
- Multiple File Systems share the same OSDs
  - Real storage pooling

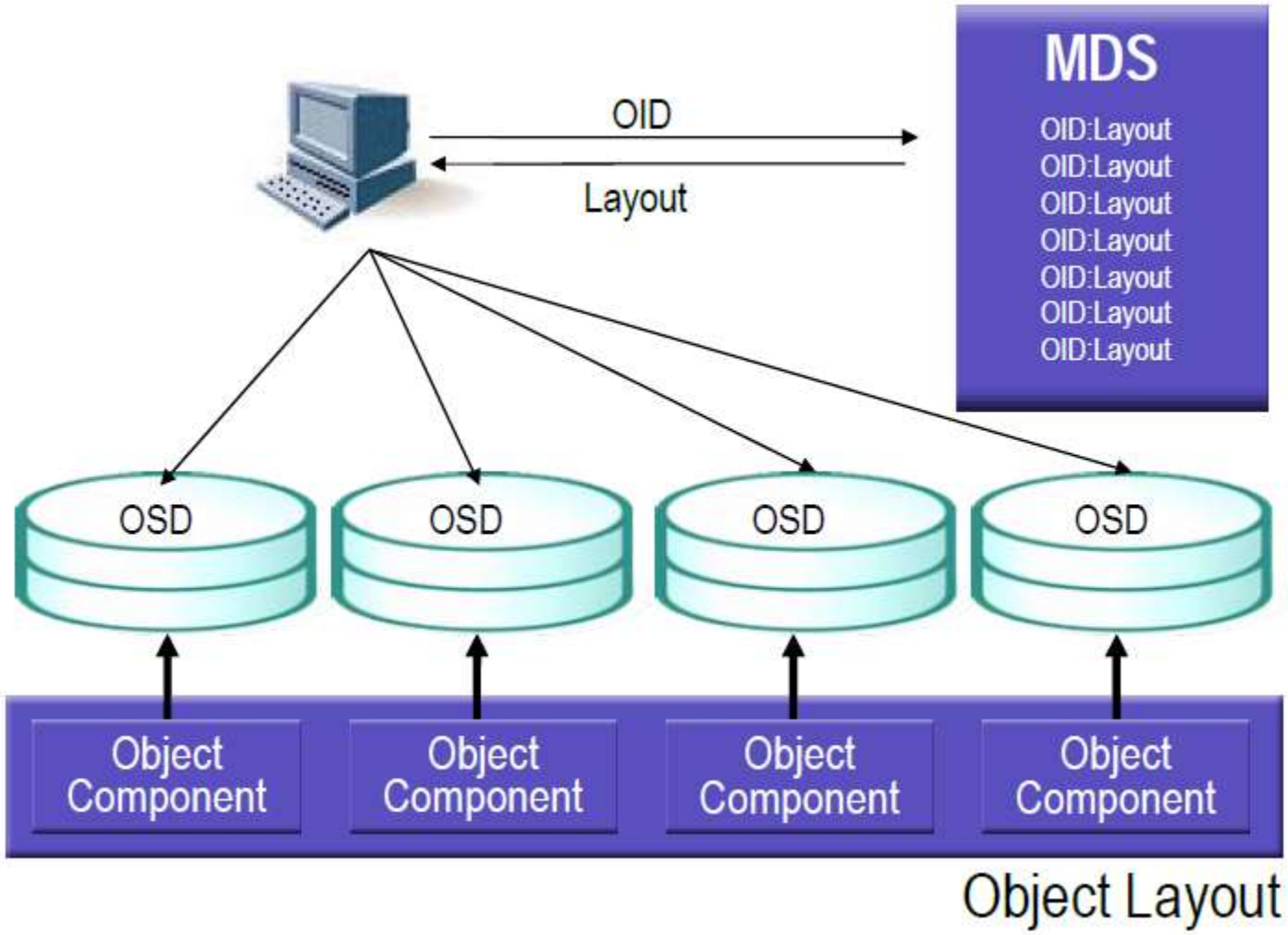
# Scaling Object-based FS (3)

- Allocation of blocks to Objects handled within OSDs
  - Partitioning improves scalability
  - Compartmentalized managements improves reliability through isolated failure domains
- The File Server piece is called the MDS
  - Meta-Data Server
  - Can be clustered for scalability

# Why Objects helps Scaling

- 90% of File System cycles are in the read/write path
  - Block allocation is expensive
  - Data transfer is expensive
  - OSD offloads both of these from the file server
  - Security model allows direct access from clients
- High level interfaces allow optimization
  - The more function behind an API, the less often you have to use the API to get your work done
- Higher level interfaces provide more semantics
  - User authentication and access control
  - Namespace and indexing

# Object Decomposition



# Object-based File Systems

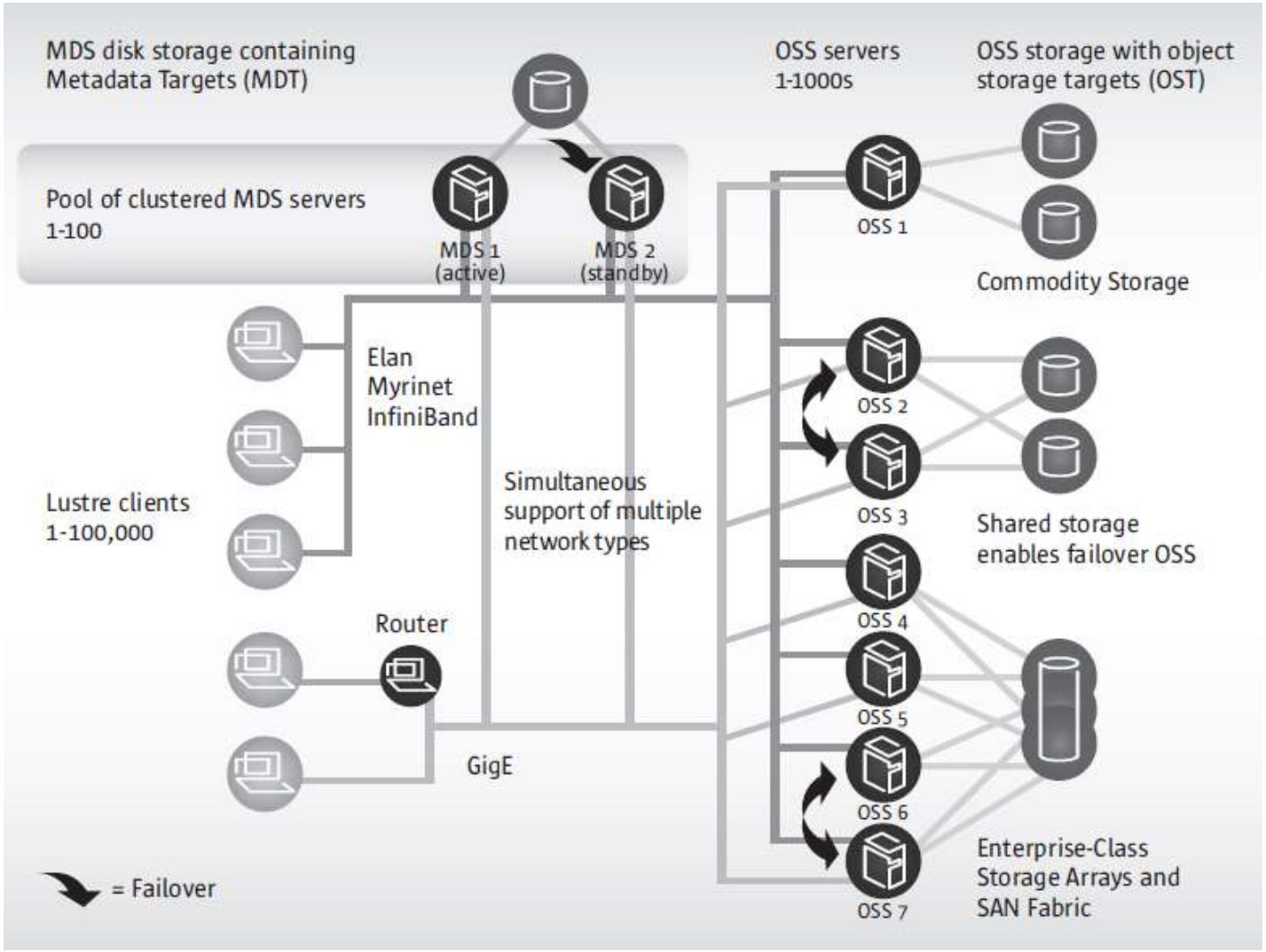
- Lustre
  - Custom OSS/OST model
  - Single metadata server
- PanFS
  - ANSI T10 OSD model
  - Multiple metadata servers
- Ceph
  - Custom OSD model
  - CRUSH metadata distribution
- pNFS
  - Out-of-band metadata service for NFSv4.1
  - T10 Objects, Files, Blocks as data services

- These systems scale
  - 1000's of disks (i.e., PB's)
  - 1000's of clients
  - 100's GB/sec
  - All in one file system

# Lustre (1)

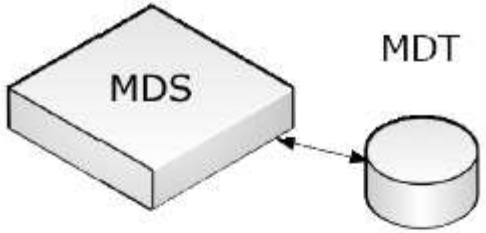
- Supercomputing focus emphasizing
  - High I/O throughput
  - Scalability in the Pbytes of data and billions of files
- OSDs called OSTs (Object Storage Targets)
- Only RAID-0 supported across Objects
  - Redundancy inside OSTs
- Runs over many transports
  - IP over ethernet
  - Infiniband
- OSD and MDS are Linux based & Client Software supports Linux
  - Other platforms under consideration
- Used in Telecom/Supercomputing Center/Aerospace/National Lab

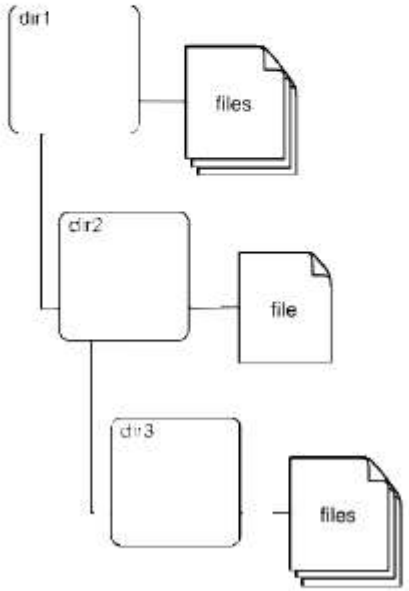
# Lustre (2) Architecture



# Lustre (3) Architecture-MDS

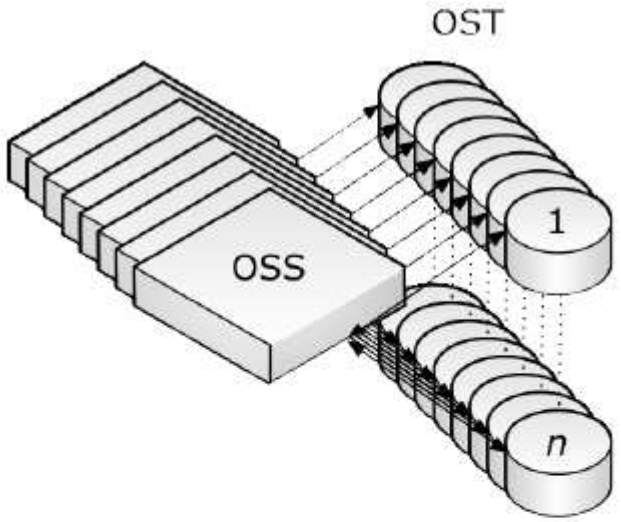
- Metadata Server (MDS)
  - Node(s) that manage namespace, file creation and layout, and locking. Directory operations
    - File open/close
    - File status
    - File creation
    - Map of file object location
  - Relatively expensive serial atomic transactions to maintain consistency
- •**Metadata Target (MDT)**
  - Block device that stores metadata

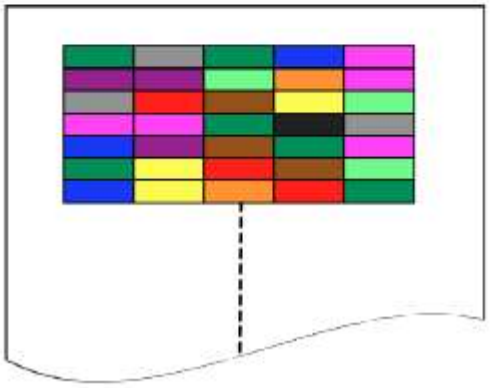


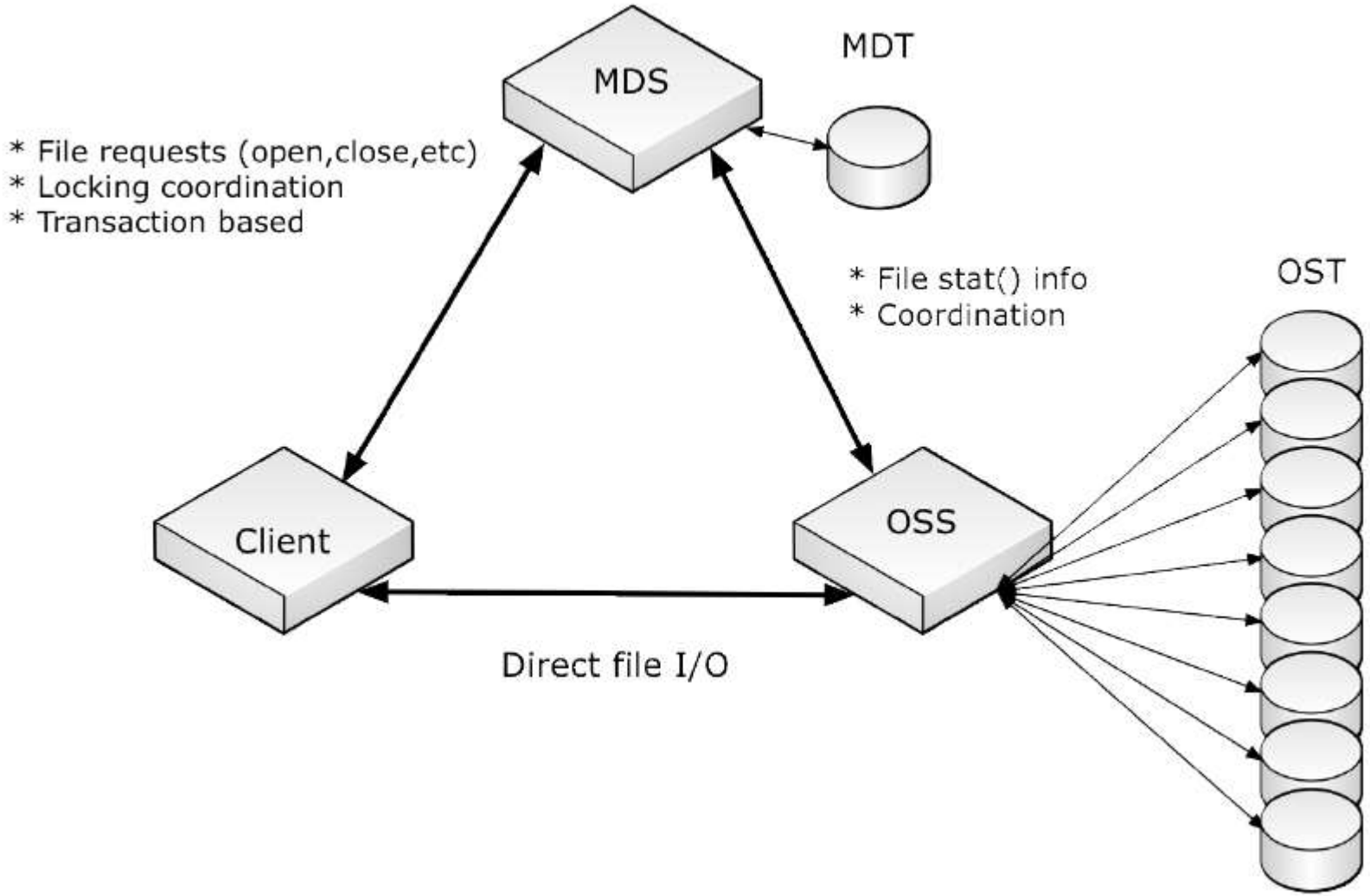
# Lustre (3) Architecture-OSS

- **Object Storage Server (OSS)**
  - Multiple nodes that manage network requests for file objects on disk.
- **Object Storage Target (OST)**
  - Block device that stores file objects



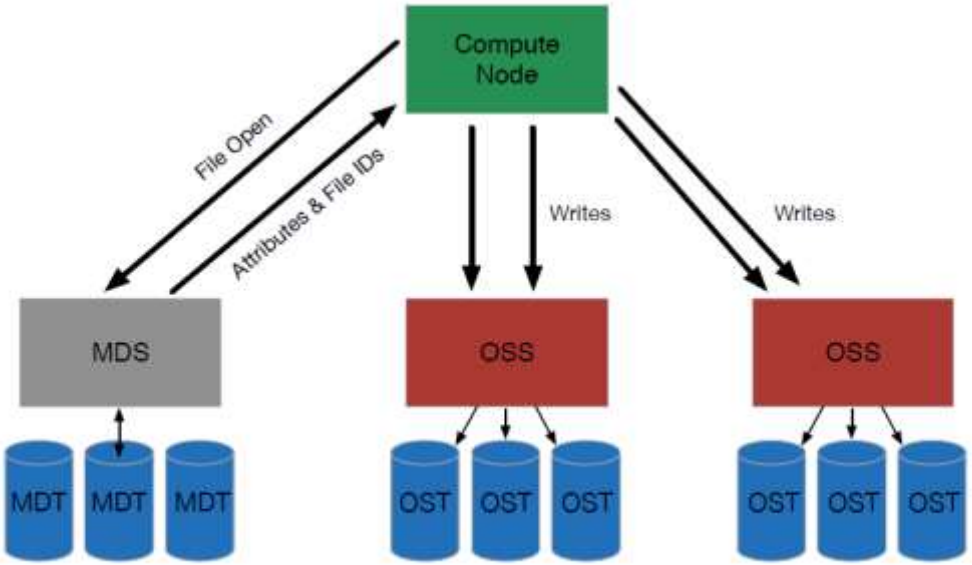



# Lustre (4) Simplest Lustre File System



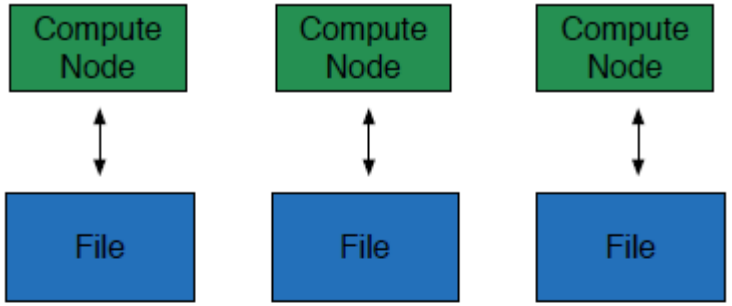
# Lustre (5) File Operation

- When a compute node needs to create or access a file, it requests the associated storage locations from the MDS and the associated MDT.
- I/O operations then occur directly with the OSSs and OSTs associated with the file bypassing the MDS.
- For read operations, file data flows from the OSTs to the compute node.

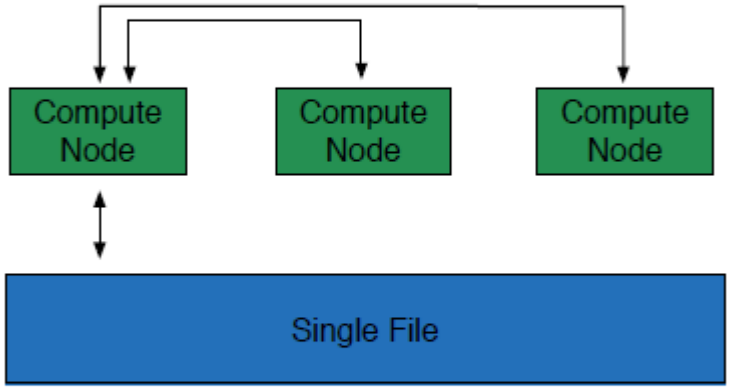


# Lustre (6) File I/Os

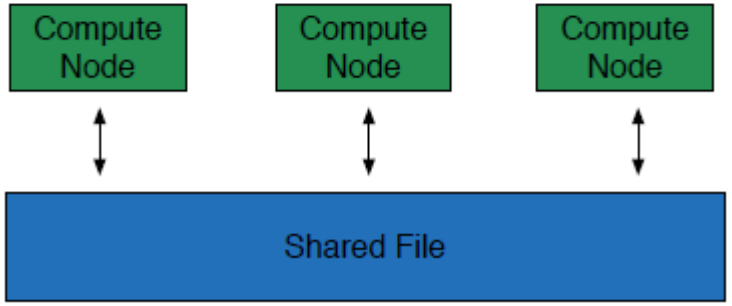
- Single stream




- Single stream through a master

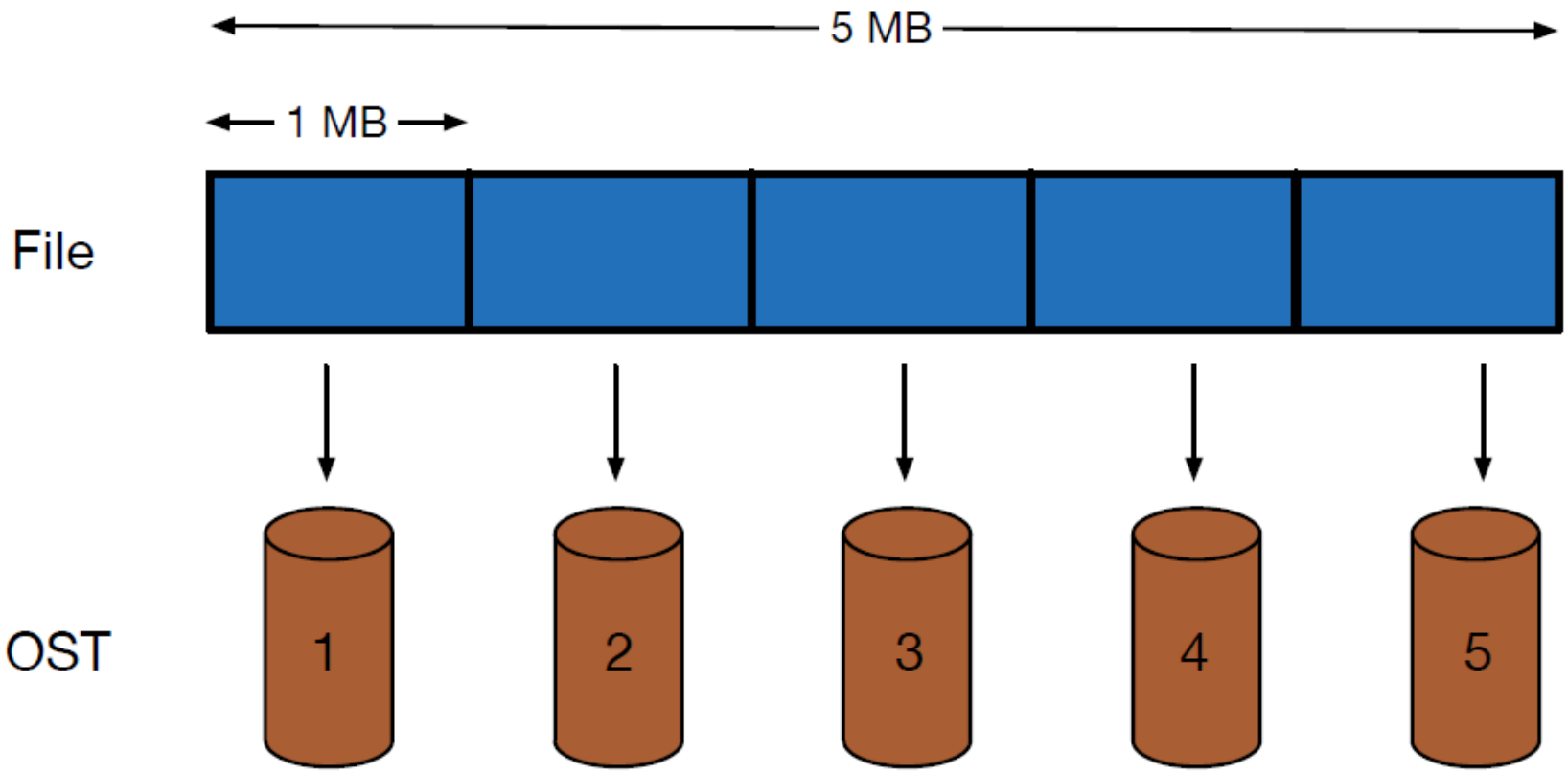



- Parallel



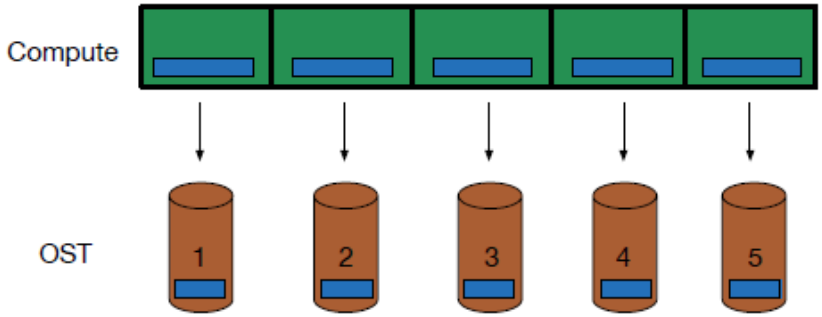
# Lustre (7) File Striping

- A file is split into segments and consecutive segments are stored on different physical storage devices (OSTs).

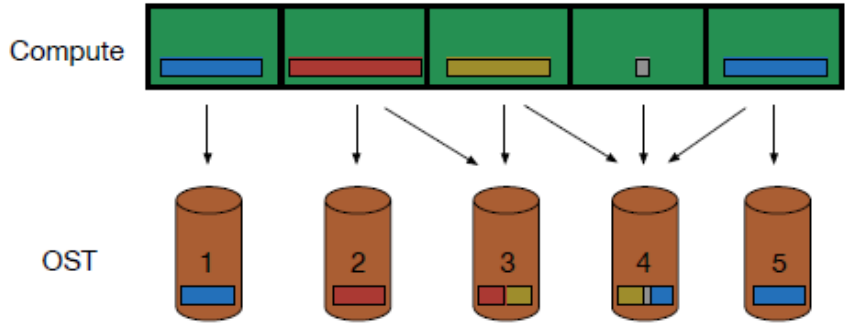


# Lustre (8) Aligned and Unaligned Stripes

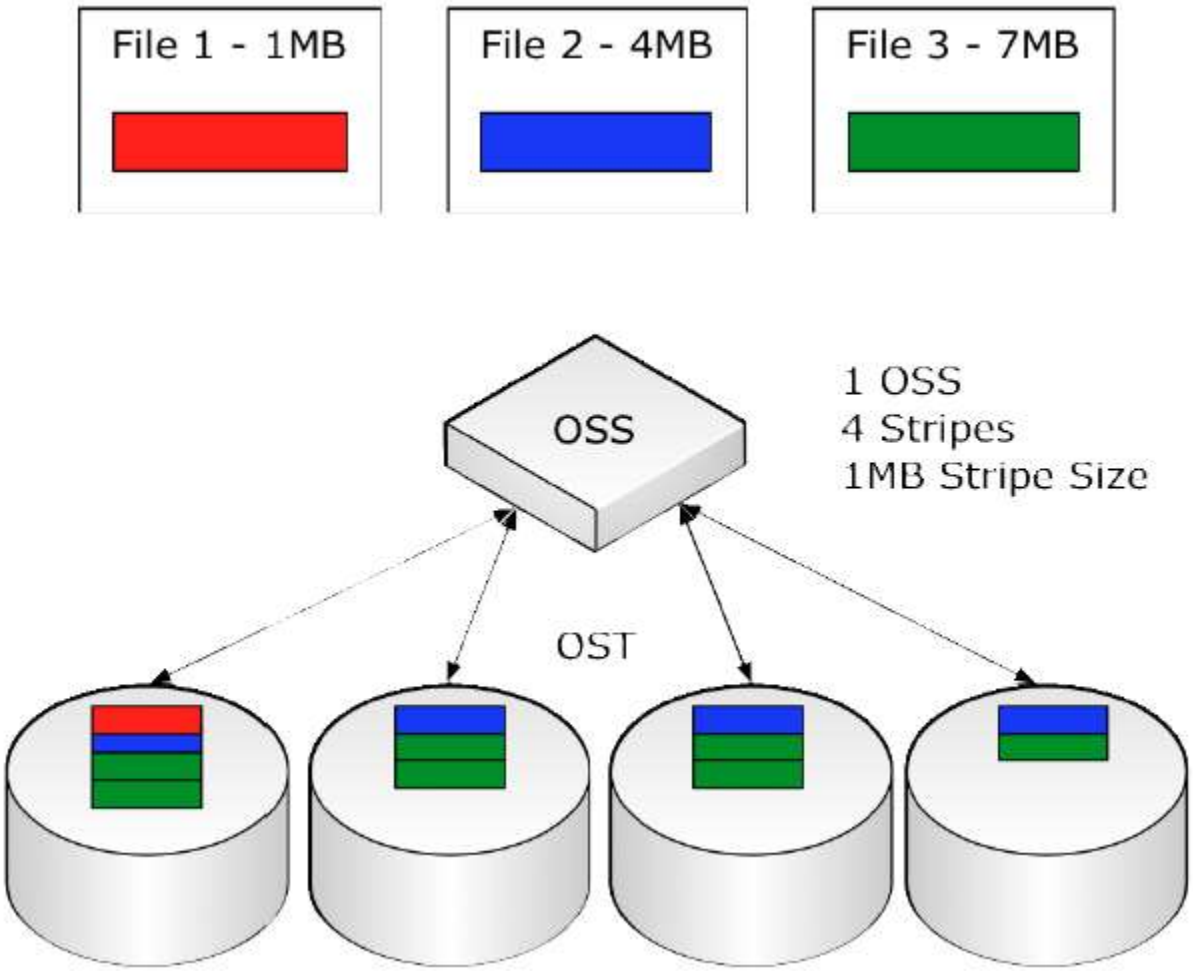
- Aligned stripes is where each segment fits fully onto a single OST. Processes accessing the file do so at corresponding stripe boundaries.




- Unaligned stripes means some file segments are split across OSTs.



# Lustre (9) Striping Example



# Lustre (10) Advantages/Disadvantages

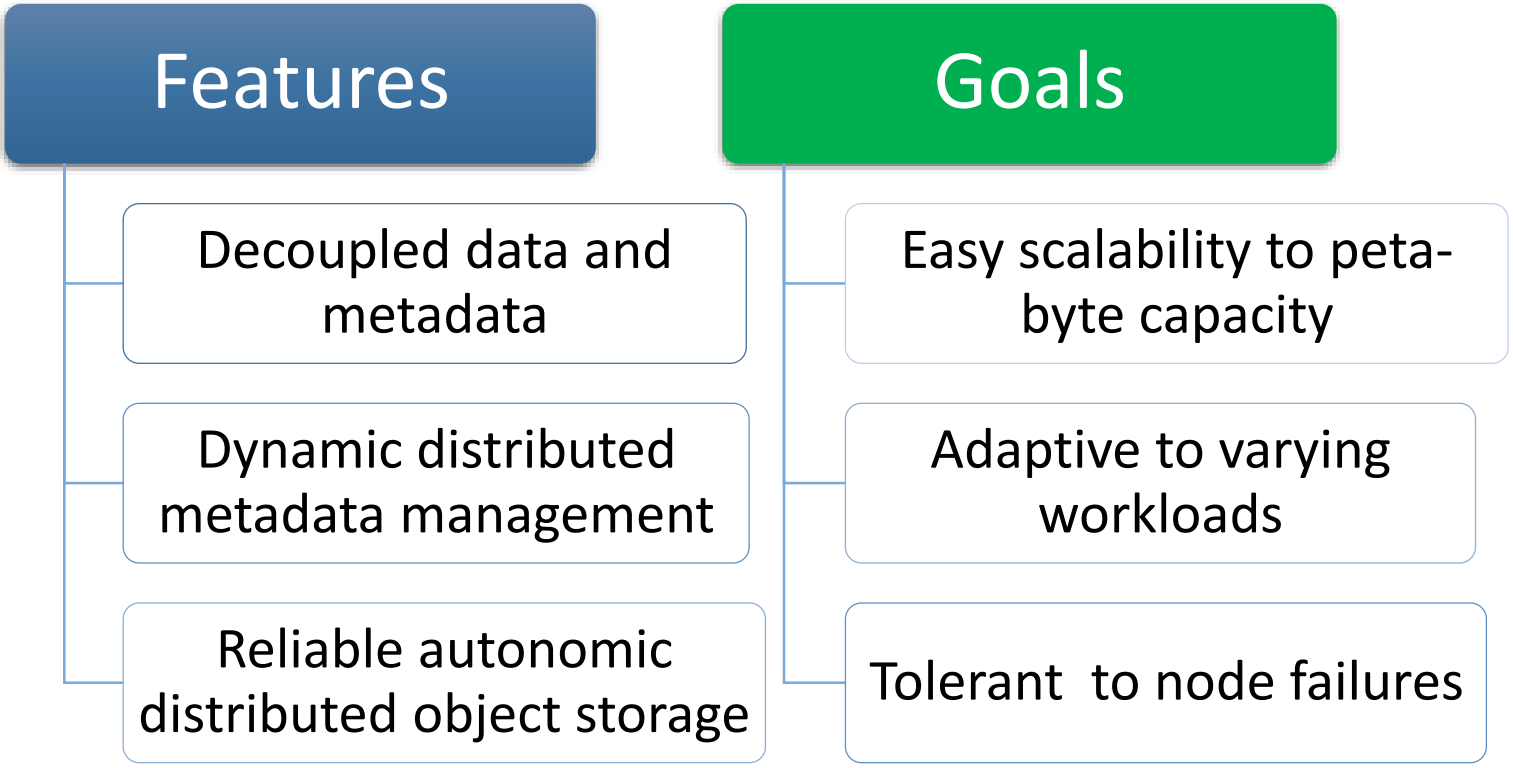
- Striping will not benefit *ALL* applications

| Advantages | Disadvantages |
| --- | --- |
| **Bandwidth** – file objects are striped across OSTs, so bandwidth is the aggregate I/O rate | **User Overhead** – Time and thought required to understand your I/O patterns and create stripe layout for directories and files |
| **File Size** – file objects striped across OST can have a total size larger than available space on any single OST | **System Overhead** – Additional stripes means more OST lookups to determine the size of the file (more time) |

# Ceph (1)

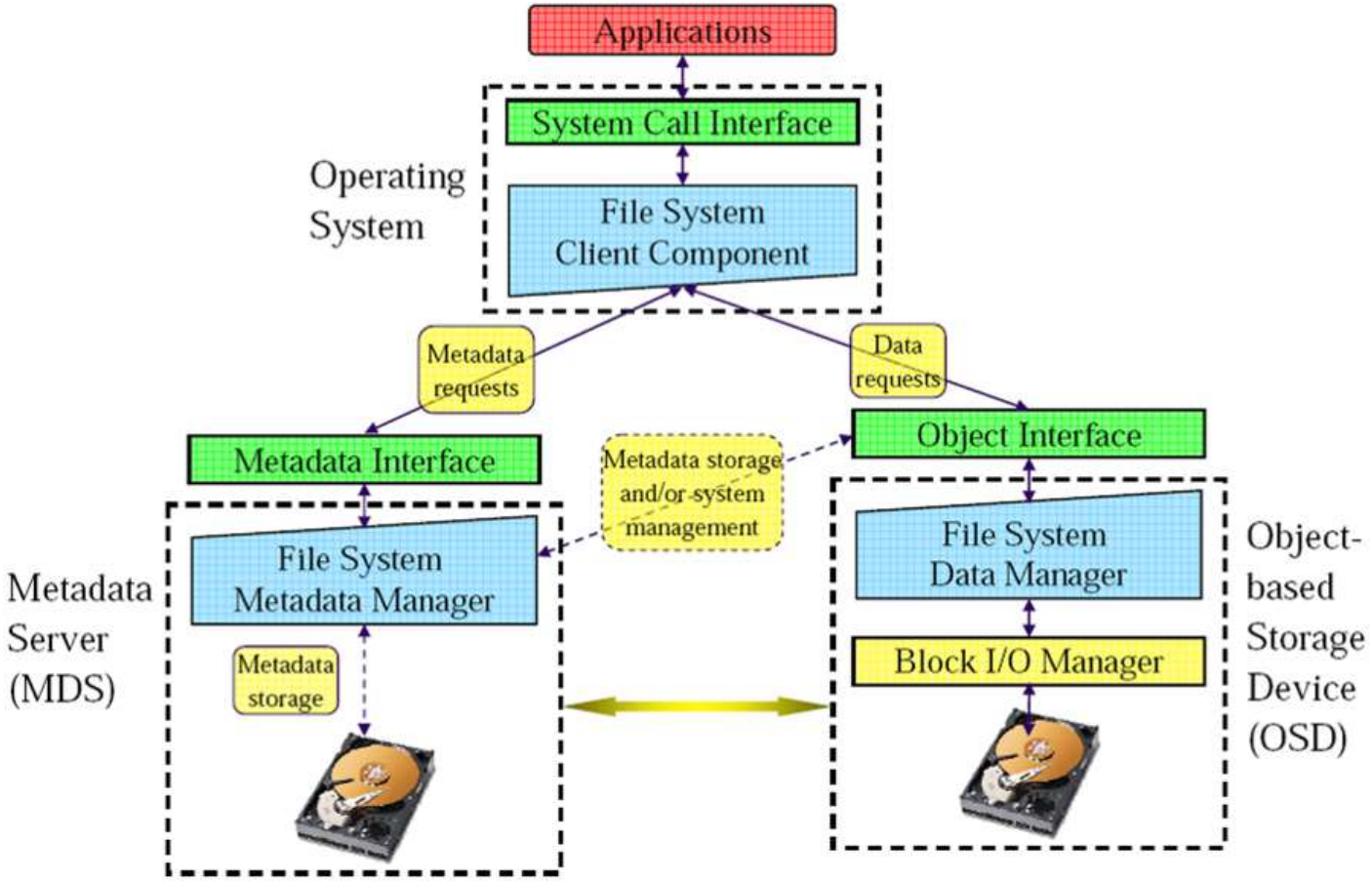
- What is Ceph?

Ceph is a distributed file system that provides excellent performance, scalability and reliability.

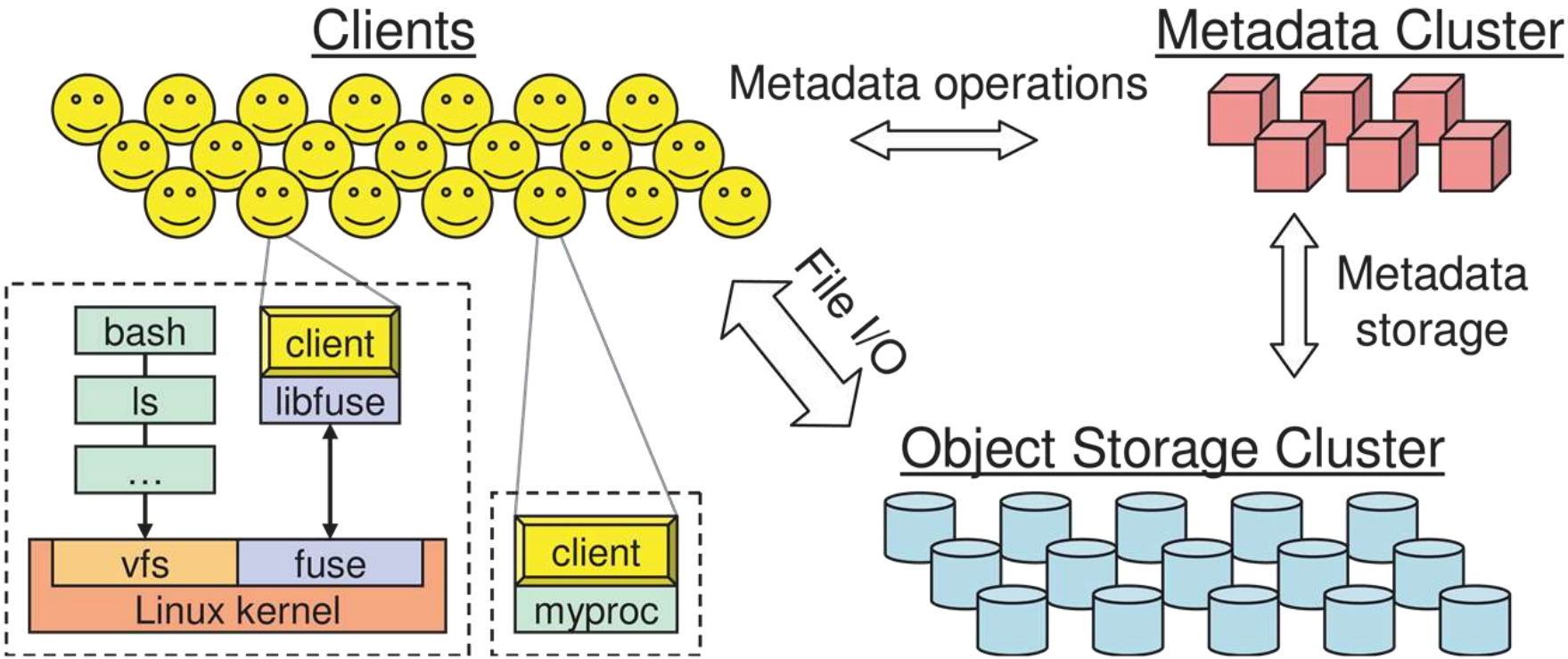


# Ceph (2) – Architecture

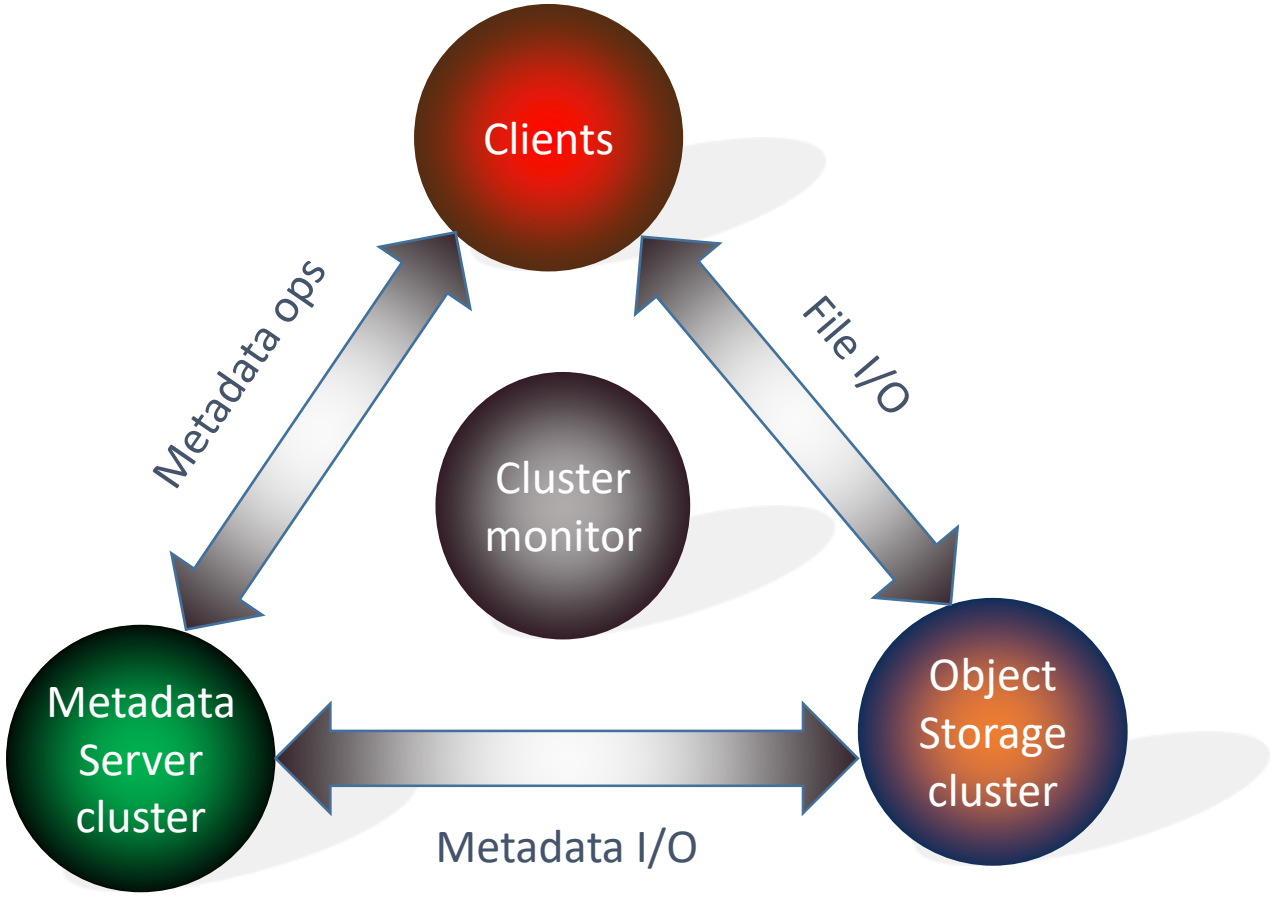
- Decoupled Data and Metadata



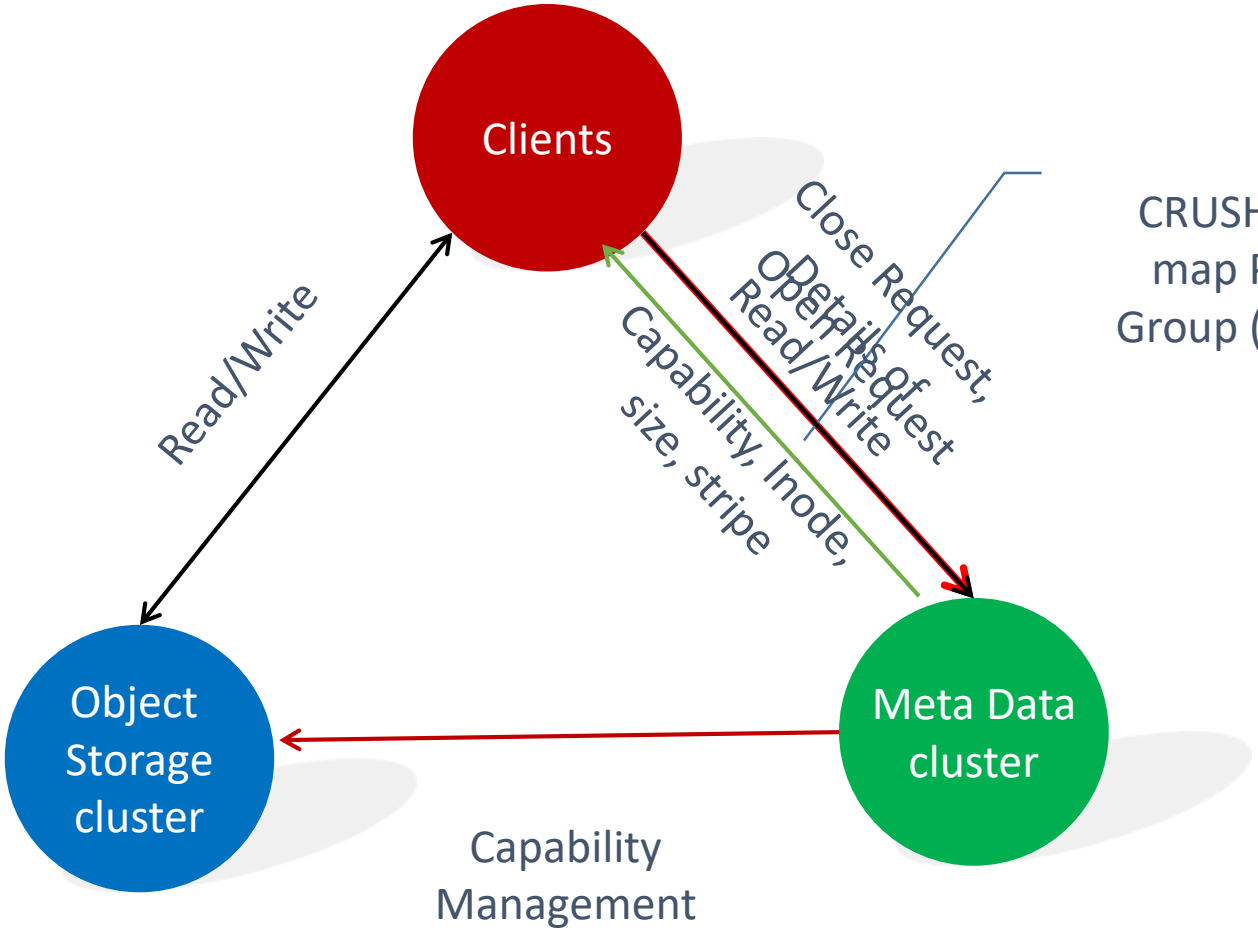
# Ceph (3) – Architecture



# Ceph (4) – Components

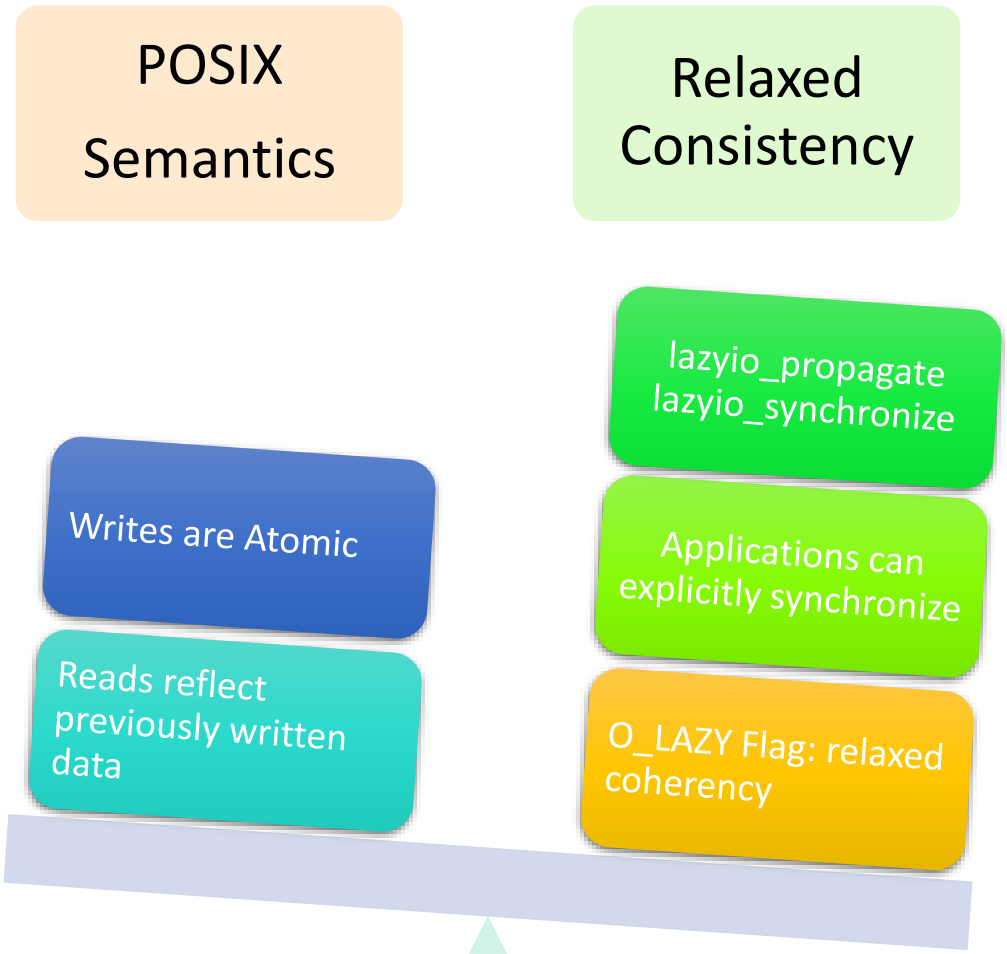


# Ceph (5) - Components

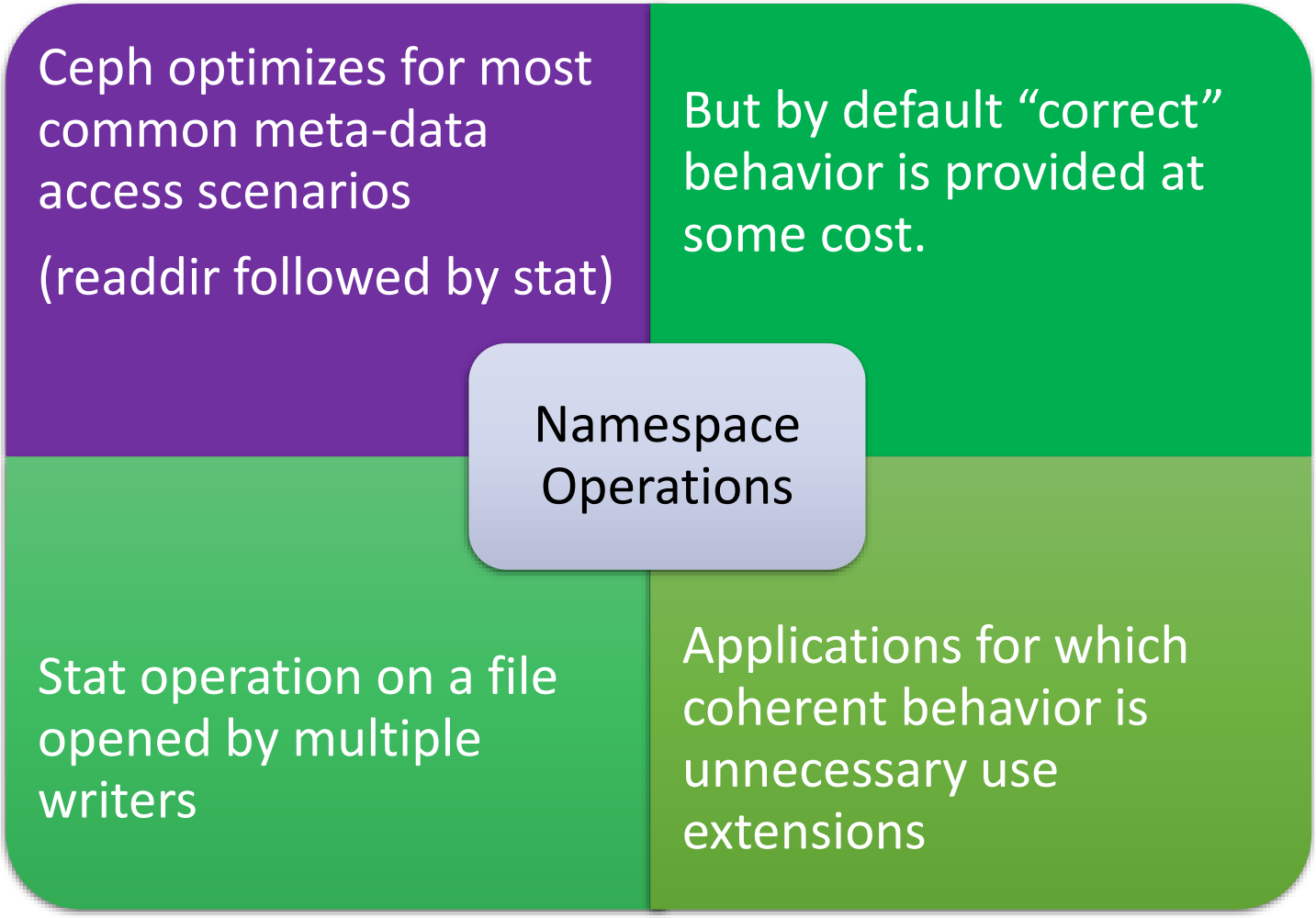


# Ceph (6) – Components

- Client Synchronization
  - Synchronous I/O. performance killer
  - Solution: HPC extensions to POSIX
    - Default: Consistency / correctness
    - Optionally relax
  - Extensions for both data and metadata

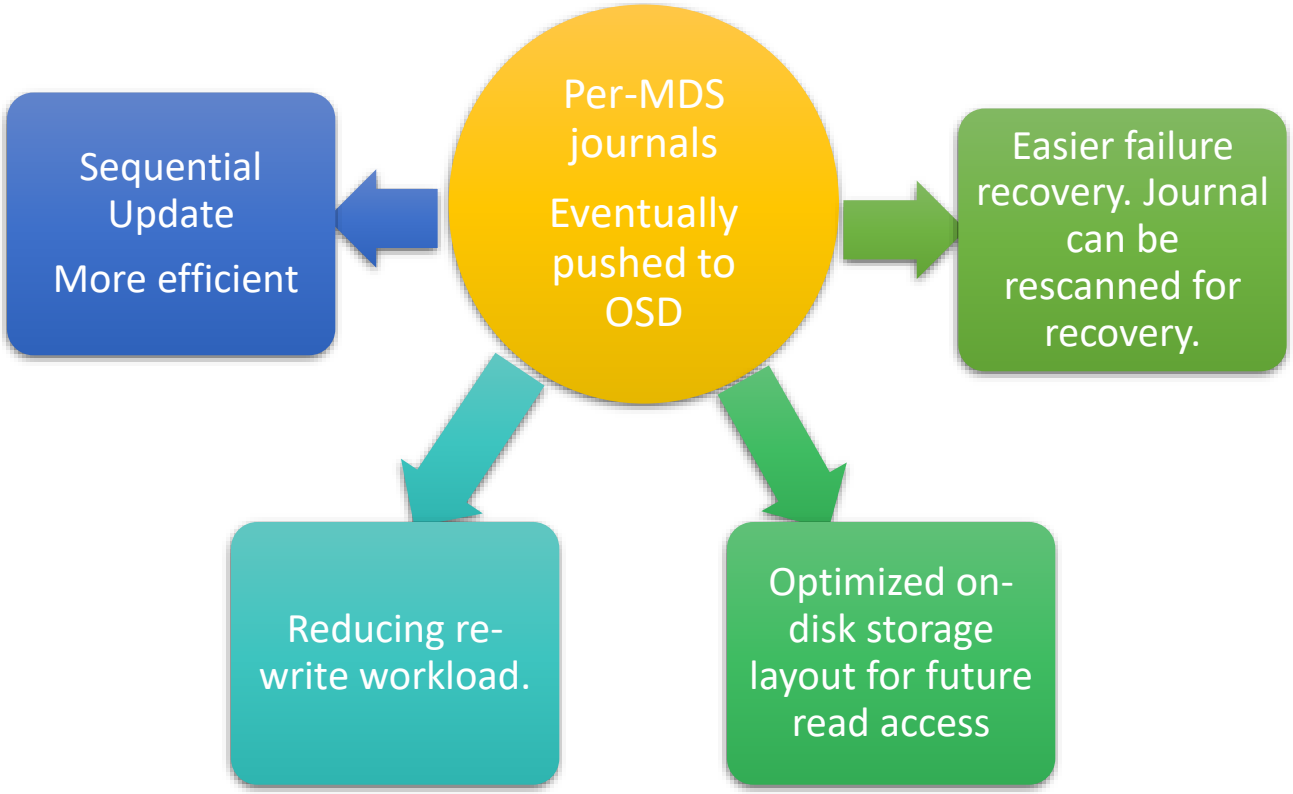


# Ceph (7) – Namespace Operations



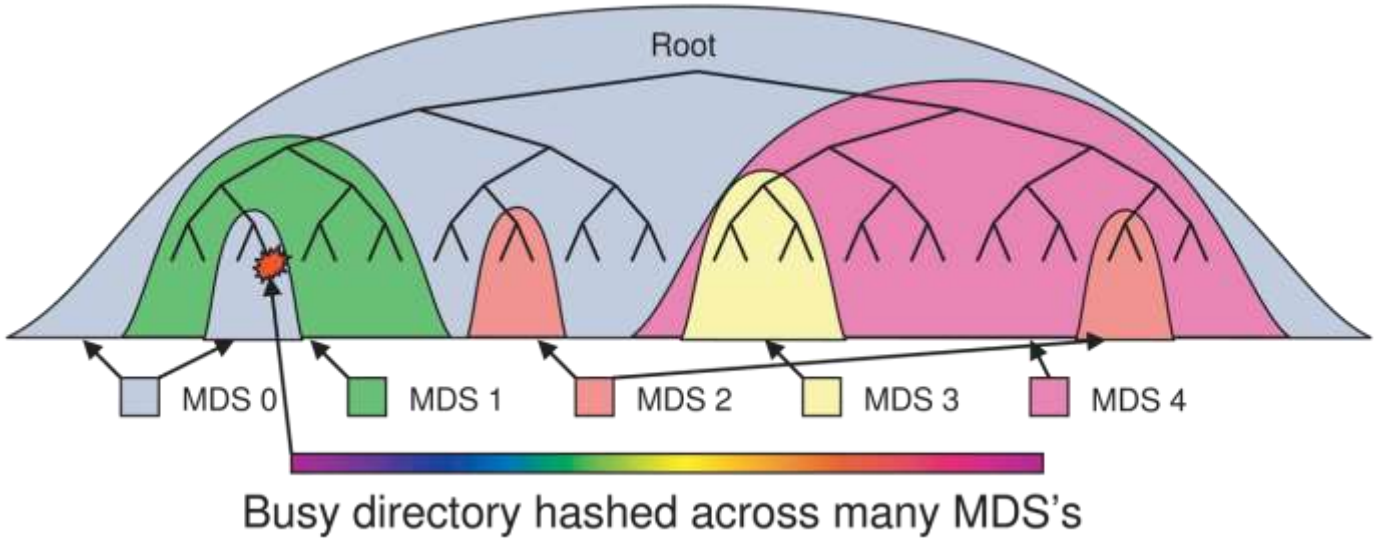
# Ceph (8) – Metadata

- Metadata Storage
  - Advantages



# Ceph (9) – Metadata

- Dynamic Sub-tree Partitioning

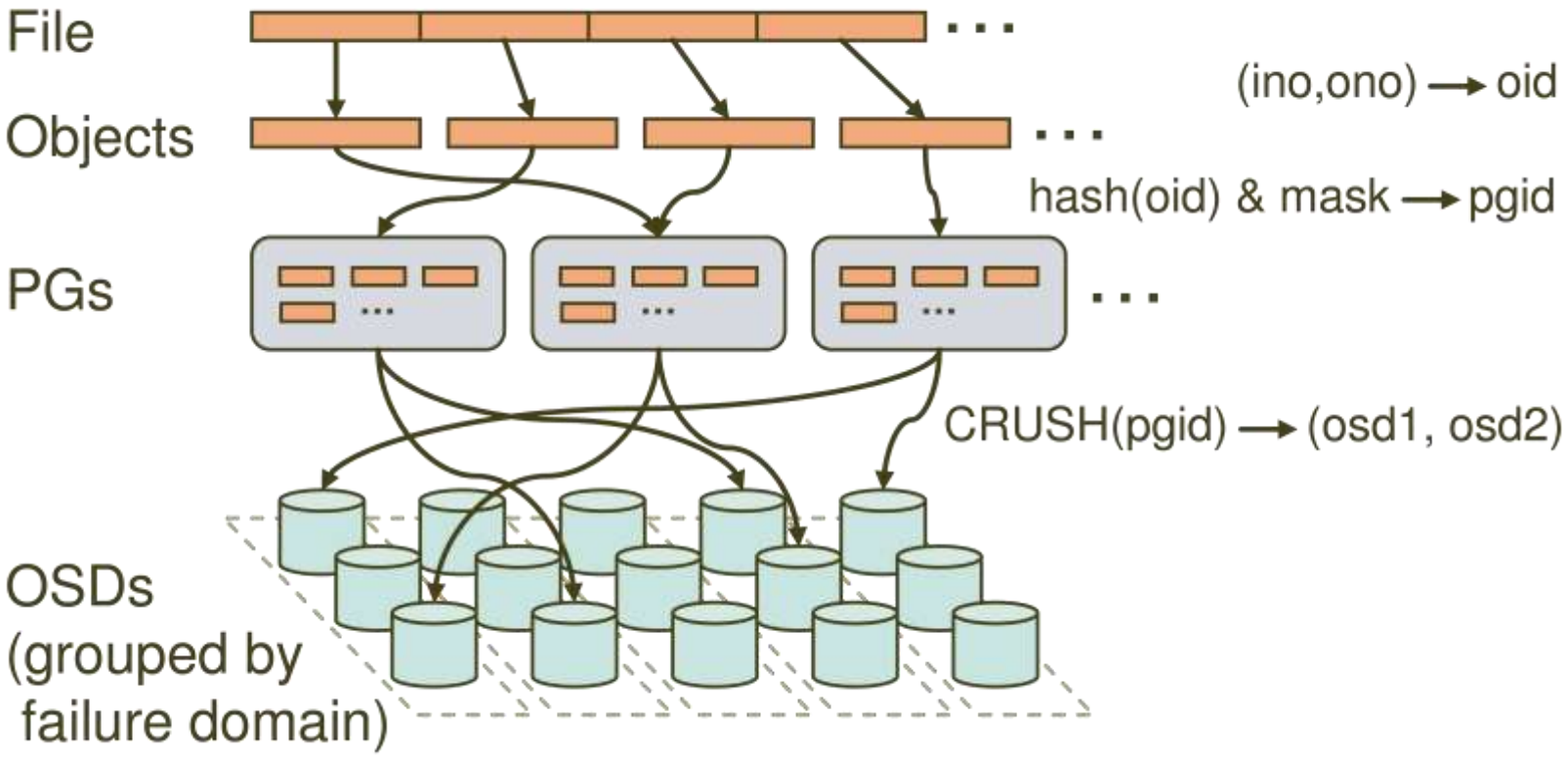



- Adaptively distribute cached metadata hierarchically across a set of nodes.
- Migration preserves locality.
- MDS measures popularity of metadata.

# Ceph (10) – Metadata

- Traffic Control for metadata access
  - Challenge
    - Partitioning can balance workload but can't deal with hot spots or flash crowds
  - Ceph Solution
    - ✓ Heavily read directories are selectively replicated across multiple nodes to distribute load
    - ✓ Directories that are extra large or experiencing heavy write workload have their contents hashed by file name across the cluster

# Ceph (11) – Distributed Object Storage

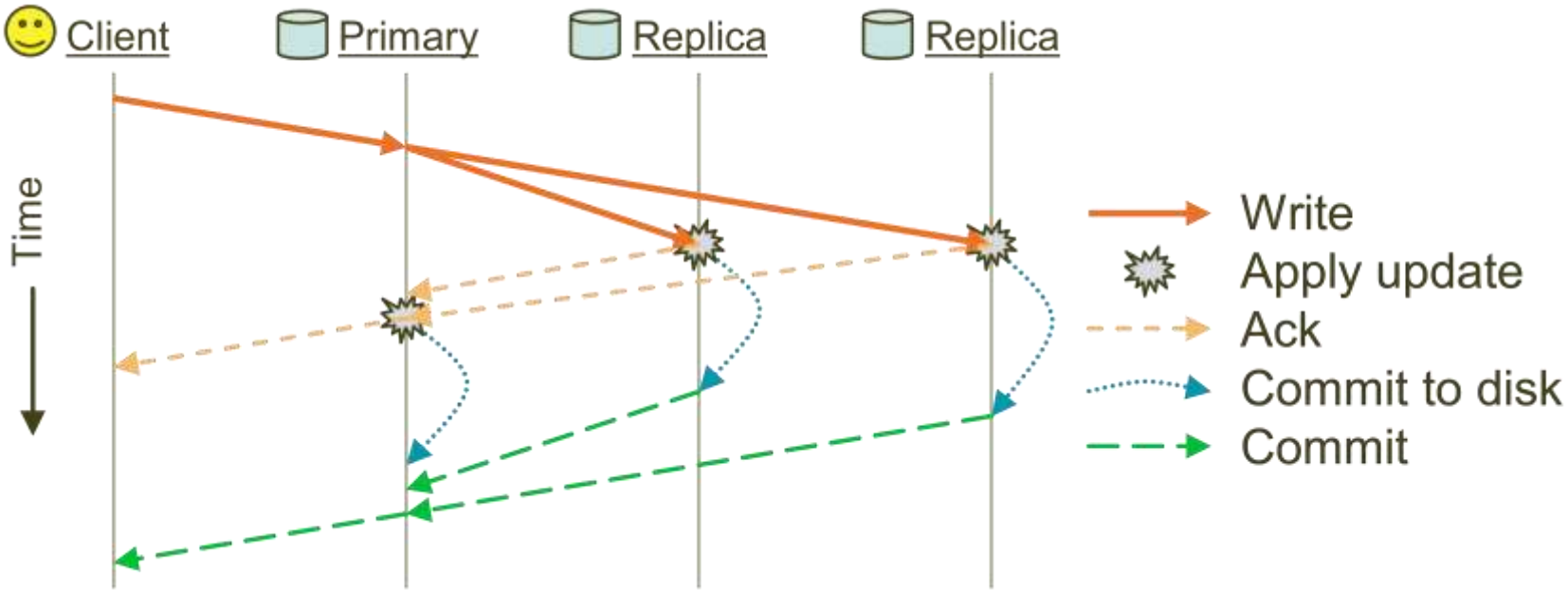


# Ceph (11) – CRUSH

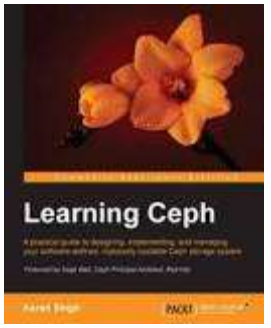
- CRUSH(x) → ($osd_{n1}$, $osd_{n2}$, $osd_{n3}$)
  - Inputs
    - x is the placement group
    - Hierarchical cluster map
    - Placement rules
  - Outputs a list of OSDs
- Advantages
  - Anyone can calculate object location
  - Cluster map infrequently updated

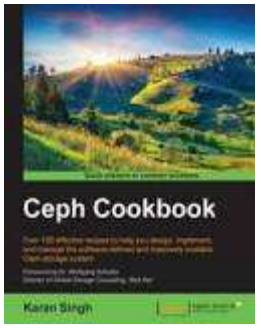
# Ceph (12) – Replication

- Objects are replicated on OSDs within same PG
  - Client is oblivious to replication



# Ceph (13) – Conclusion




- Strengths:
  - Easy scalability to peta-byte capacity
  - High performance for varying work loads
  - Strong reliability
- Weaknesses:
  - MDS and OSD Implemented in user-space
  - The primary replicas may become bottleneck to heavy write operation
  - N-way replication lacks storage efficiency
- References
  - Ceph: A Scalable, High Performance Distributed File System. In Proc. of OSDI'06

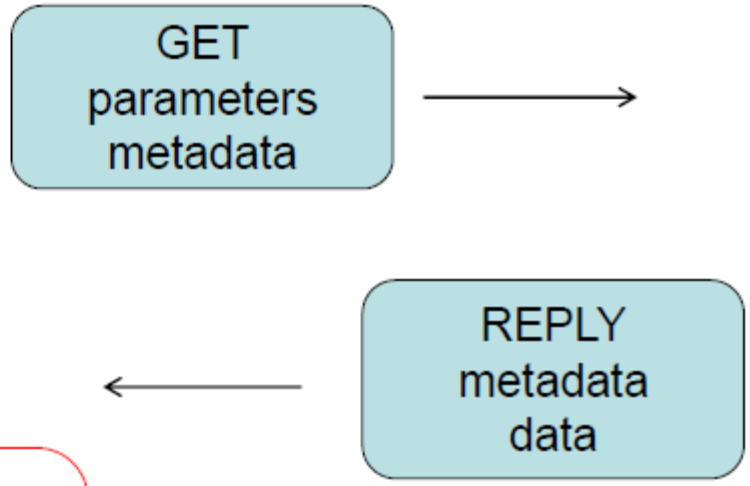
# 4 Object-based Storage in Cloud

# Web Object Features

- RESTful API (i.e., web-based)
- Security/Authentication tied to Billing
- Metadata capabilities
- Highly available
- Loosely consistent
- Data Storage
  - Blobs
  - Tables
  - Queues
- Other related APIs (compute, search, etc.)
  - Storage API is relatively simple in comparison

# Simple HTTP example

```
% telnet www.google.com 80
GET /index.html HTTP/1.0
(blank line)
HTTP/1.0 200 OK
Date: Wed, 13 Feb 2013 07:24:07 GMT
Content-Type: text/html; charset=ISO-8859-1
<html>
<head><title>Google</title></head>
<body><img src= /images/srpr/logo3w.png>
<form><input type=text name=q>
<input type=submit value="Google Search" name="search">
<input type=submit value="I'm Feeling Lucky" name="lucky">
</form></body></html>
```



# HTTP and objects

- Request specifies method and object:
  - Operation: GET, POST, PUT, HEAD, COPY
  - Object ID (/index.html)
- Parameters use MIME format borrowed from email
  - Content-type: utf8;
  - Set-Cookie: tracking=1234567;
- Add a data payload
  - Optional
  - Separated from parameters with a blank line (like email)
- Response has identical structure
  - Status line, key-value parameters, optional data payload

**This is a method call on an object**

**These are parameters**

**This is data**

# OpenStack REST API for Storage

- GET v1/account HTTP/1.1
  - Login to your account
- HEAD v1/account HTTP/1.1
  - List account metadata
- PUT v1/account/container HTTP/1.1
  - Create container
- PUT v1/account/container/object HTTP/1.1
  - Create object
- GET v1/account/container/object HTTP/1.1
  - Read object
- HEAD v1/account/container/object HTTP/1.1
  - Read object metadata

# Create an object

```
PUT /v1/<account>/<container>/<object> HTTP/1.1
Host: storage.swiftdrive.com
X-Auth-Token: eaaafd18-0fed-4b3a-81b4-663c99ec1cbb
ETag: 8a964ee2a5e88be344f36c22562a6486
Content-Length: 512000
X-Delete-At: 1339429105
Content-Disposition: attachment; filename=platmap.mp4
Content-Type: video/mp4
Content-Encoding: gzip
X-Object-Meta-PIN: 1234
[ ...object content... ]
```

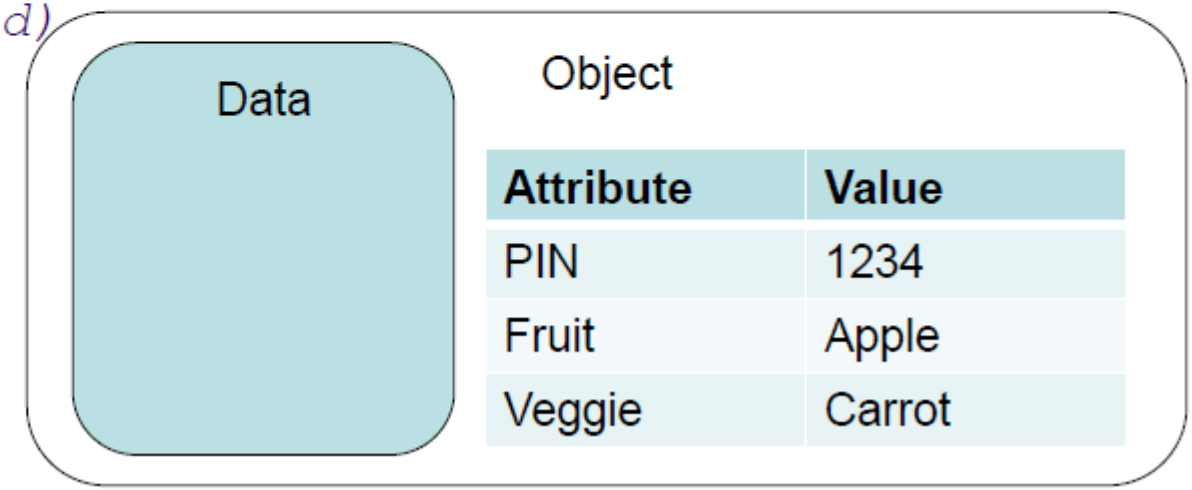
MD5 checksum

Mon Jun 11 08:38:25 PDT 2012

User defined metadata

# Update metadata

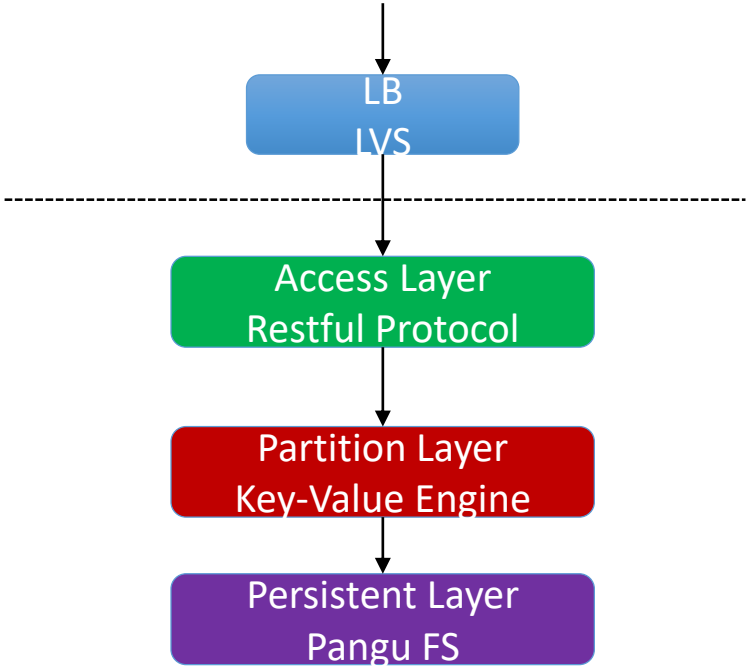
```
POST /v1/<account>/<container>/<object> HTTP/1.1
Host: storage.swiftdrive.com
X-Auth-Token: eaaafd18-0fed-4b3a-81b4-663c99ec1cbb
X-Object-Meta-Fruit: Apple
X-Object-Meta-Veggie: Carrot

(no data payload)
```



# Ali OSS (1)

阿里云 aliyun.com

- Access URL: http://<bucket>.oss-cn-beijing.aliyuncs.com/<object>




Load Balancing

Protocol Manager &
Access Control

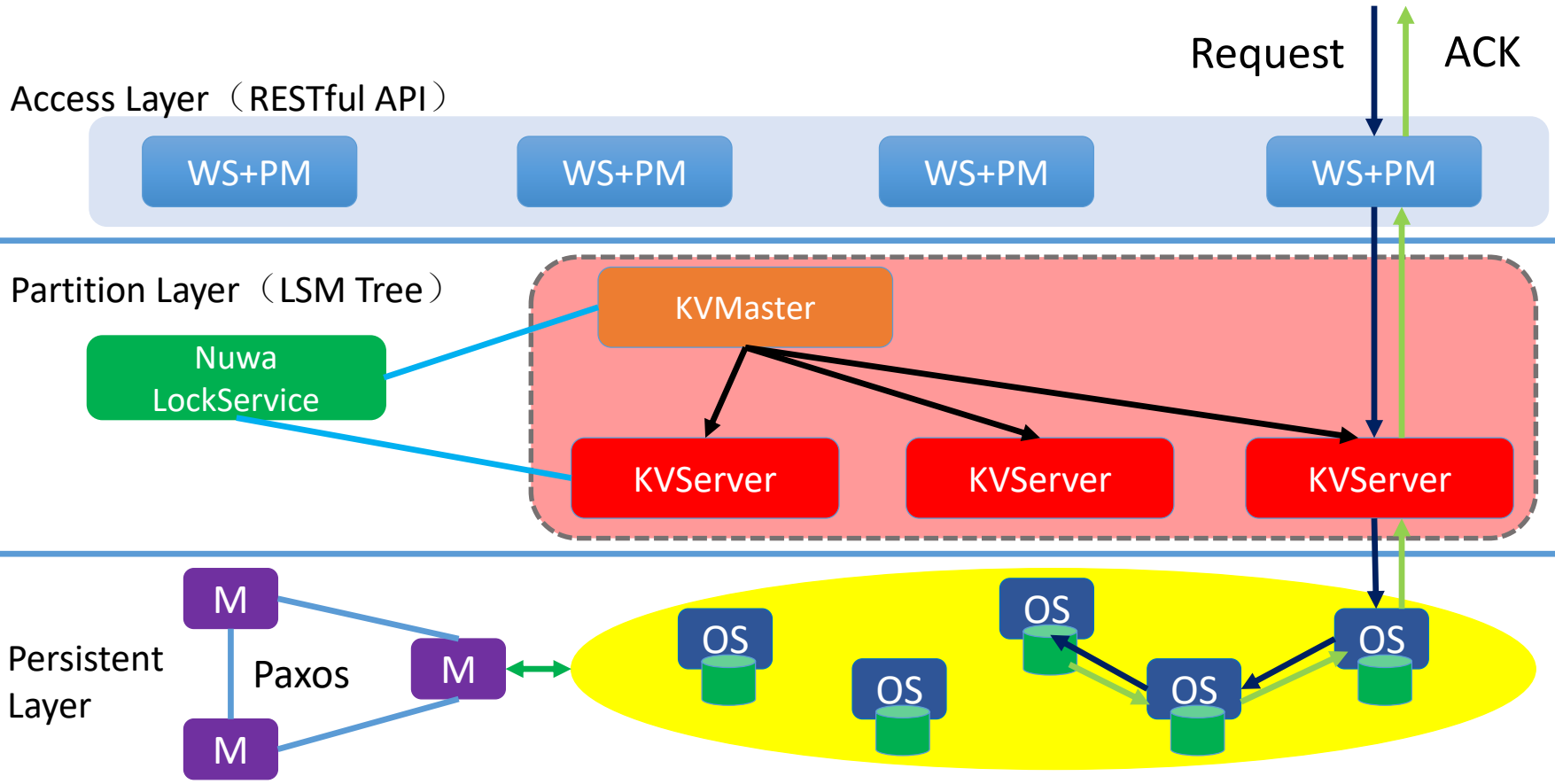
Partition & Index

Persistent, Redundancy
& Fault-Tolerance

# Ali OSS (2) Architecture

阿里云 aliyun.com

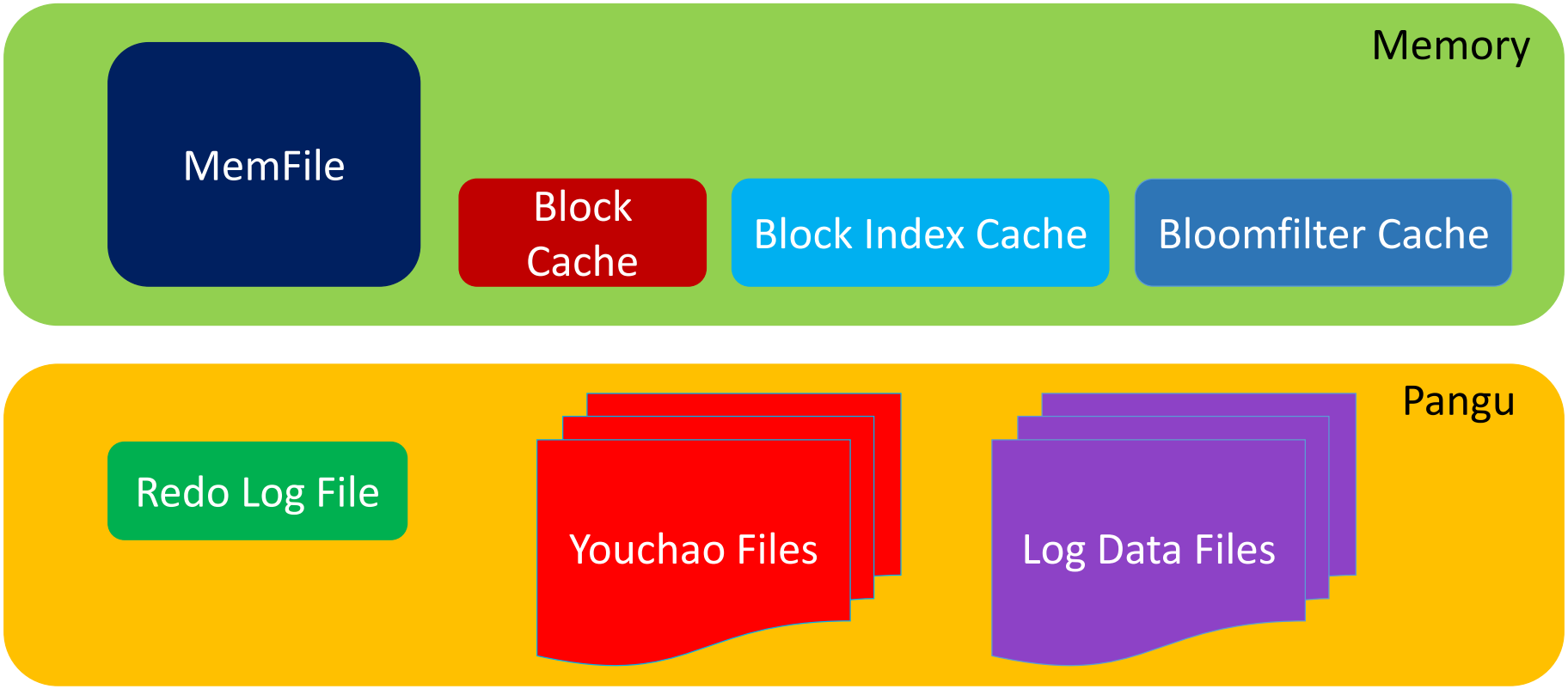
- WS: Web Server PM: Protocol Manager



# Ali OSS (3) Partition Layer



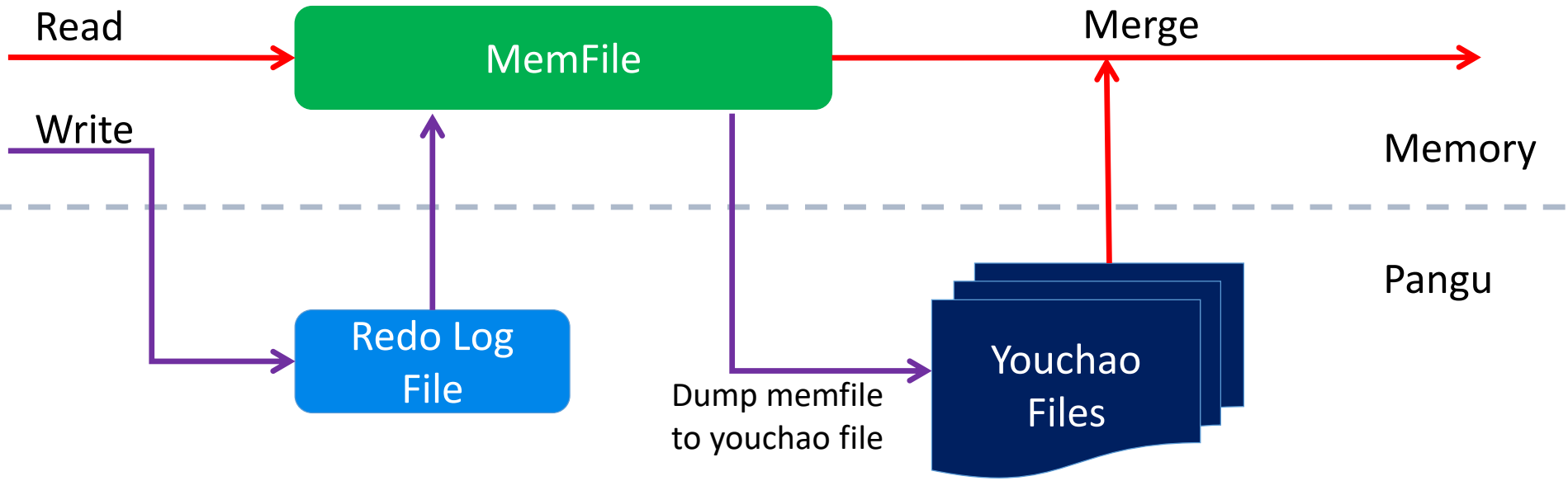
- Append/Dump/Merge



# Ali OSS (4) Partition Layer

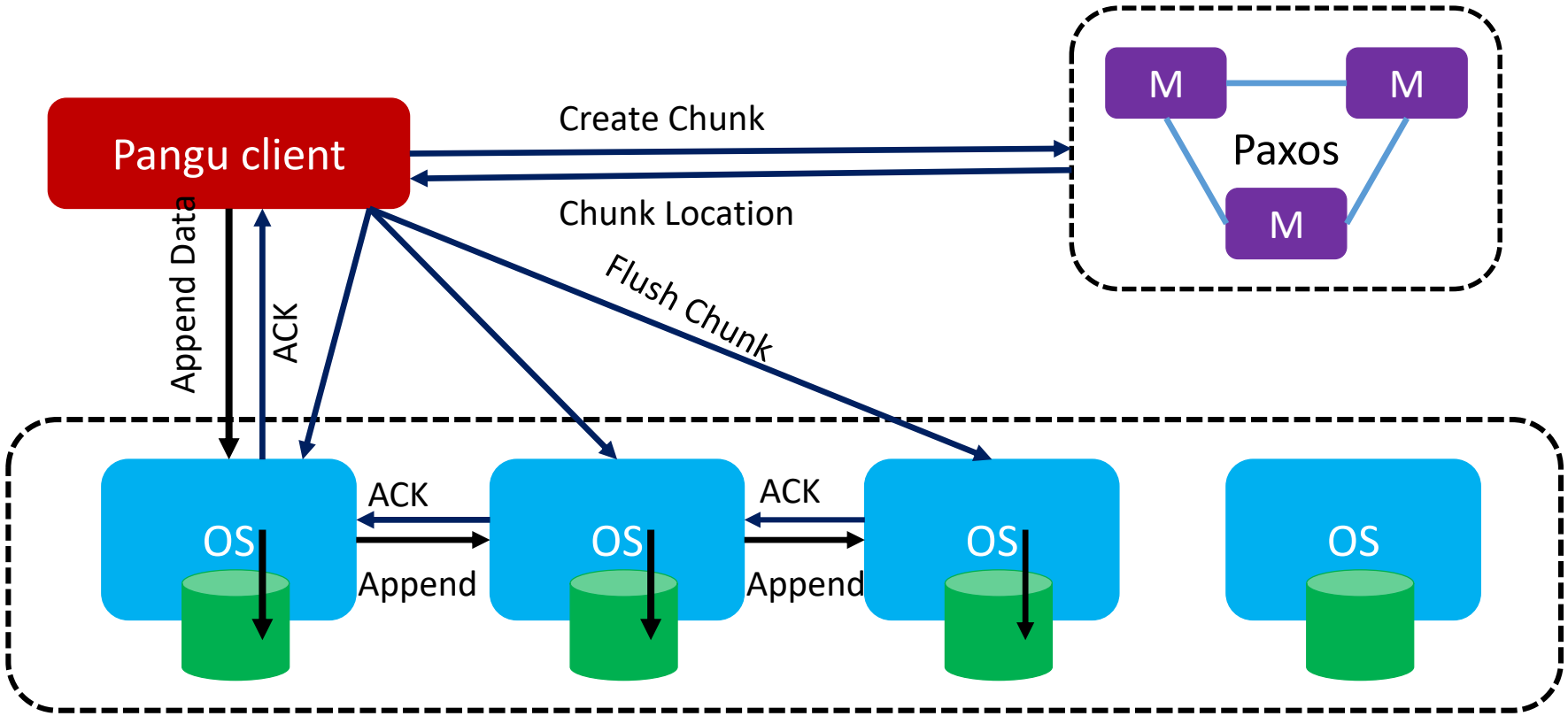


- Read/Write Process



# Ali OSS (5) Persistent Layer

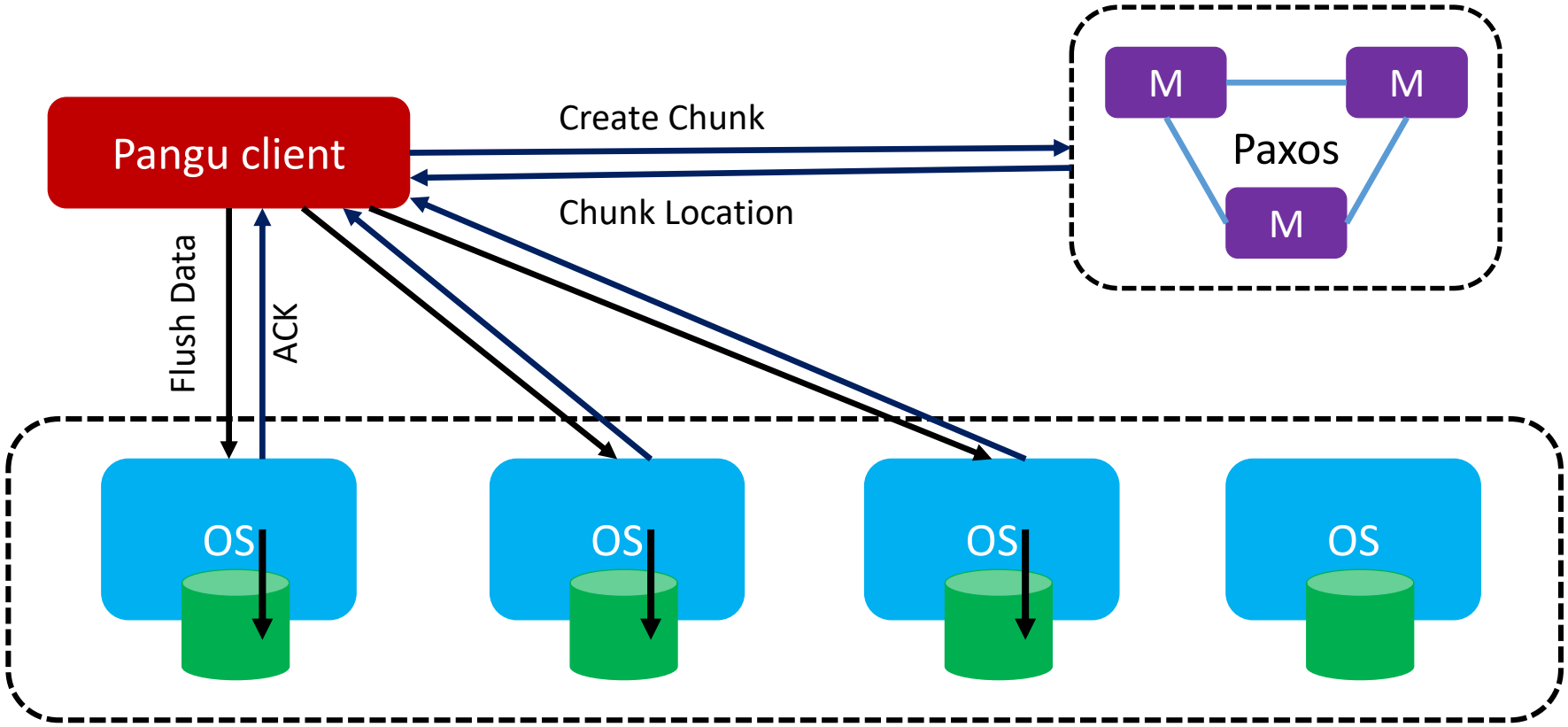
- Write Pangu Normal File



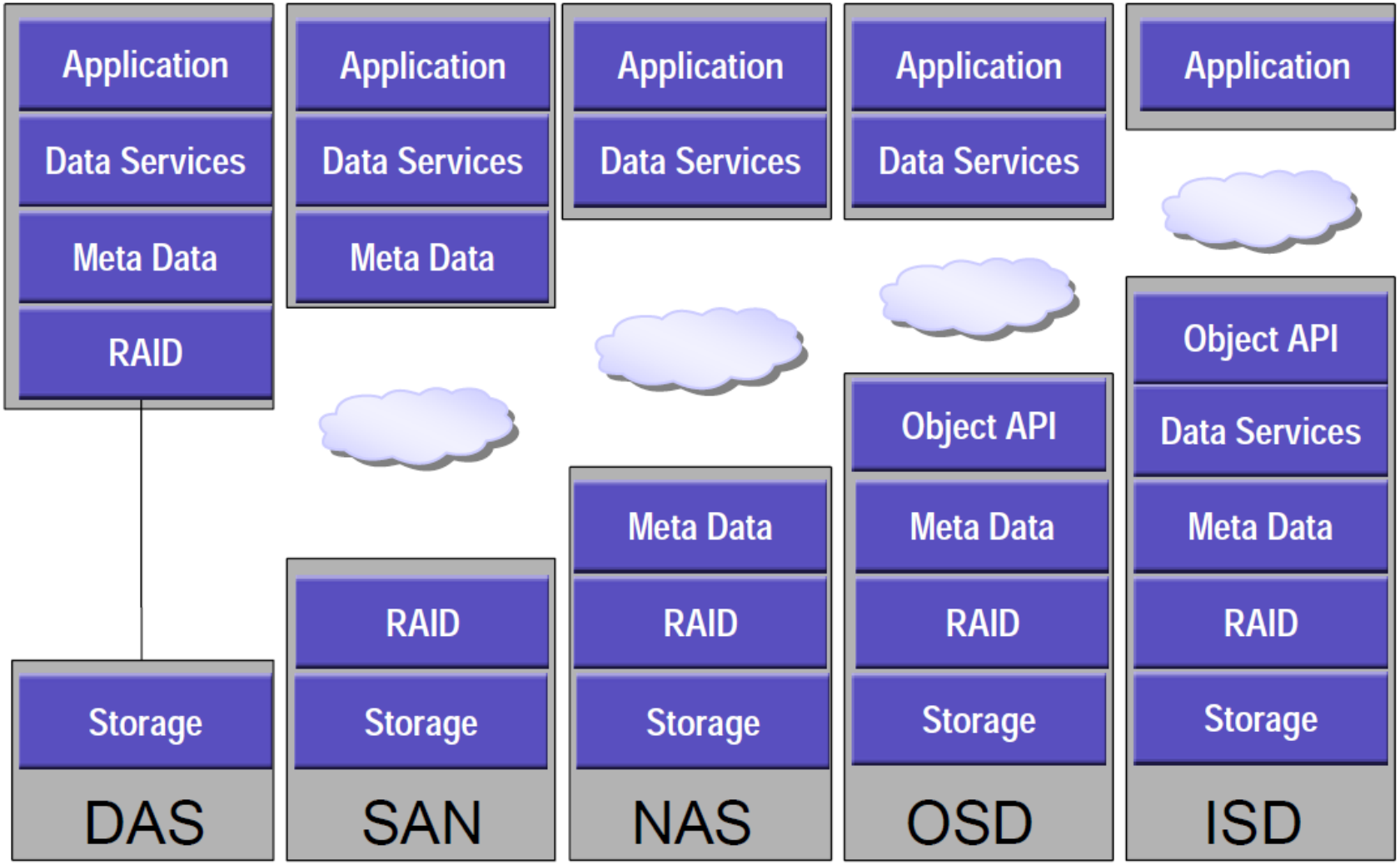
# Ali OSS (6) Persistent Layer

阿里云 aliyun.com

- Write Pangu Log File



# The Evolution of Data Storage



# 5 Google File System (GFS)

# Why build GFS?

- Node failures happen frequently
- Files are huge – multi-GB
- Most files are modified by appending at the end
  - Random writes (and overwrites) are practically non-existent
- High sustained bandwidth is more important than low latency
  - Place more priority on processing data in bulk

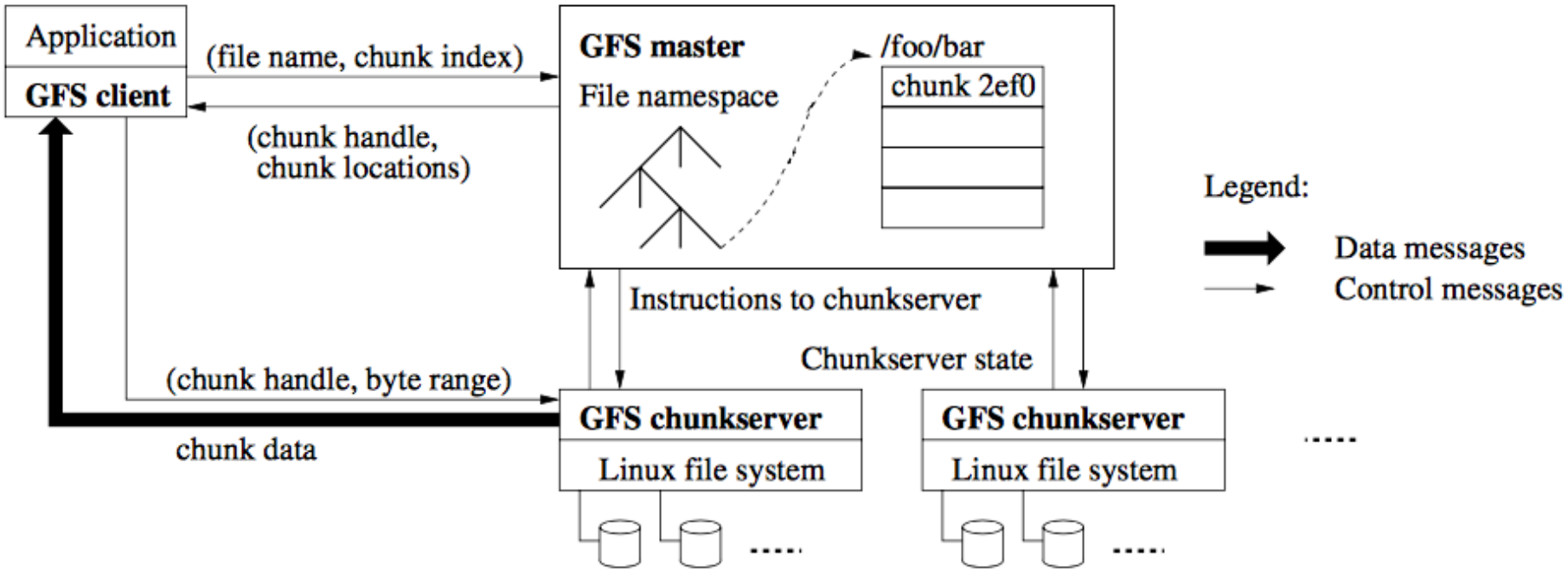
# Typical workloads on GFS

- Two kinds of reads: large streaming reads & small random reads
  - Large streaming reads usually read 1MB or more
  - Oftentimes, applications read through contiguous regions in the file
  - Small random reads are usually only a few KBs at some arbitrary offset
- Also many large, sequential writes that append data to files
  - Similar operation sizes to reads
  - Once written, files are seldom modified again
  - Small writes at arbitrary offsets do not have to be efficient
- Multiple clients (e.g. ~100) concurrently appending to a single file
  - e.g. producer-consumer queues, many-way merging

# Interface

- Not POSIX-compliant, but supports typical file system operations: `create`, `delete`, `open`, `close`, `read`, and `write`
- `snapshot`: creates a copy of a file or a directory tree at low cost
- `record append`: allow multiple clients to append data to the same file concurrently
  - At least the very first append is guaranteed to be atomic

# GFS Architecture (1)



# GFS Architecture (2)

- **Very important**: data flow is decoupled from control flow
  - Clients interact with the master for metadata operations
  - Clients interact directly with chunkservers for all files operations
  - This means performance can be improved by scheduling expensive data flow based on the network topology
- Neither the clients nor the chunkservers cache file data
  - Working sets are usually too large to be cached, chunkservers can use Linux's buffer cache

# The Master Node (1)

- Responsible for all system-wide activities
  - managing chunk leases, reclaiming storage space, load-balancing
- Maintains all file system metadata
  - Namespaces, ACLs, mappings from files to chunks, and current locations of chunks
  - all kept in memory, namespaces and file-to-chunk mappings are also stored persistently in ***operation log***
- Periodically communicates with each chunkserver in HeartBeat messages
  - This let's master determines chunk locations and assesses state of the overall system
  - **Important**: The chunkserver has the final word over what chunks it does or does not have on its own disks – **not** the master

# The Master Node (2)

- For the namespace metadata, master does not use any per-directory data structures – no inodes! (No symlinks or hard links, either.)
  - Every file and directory is represented as a node in a lookup table, mapping pathnames to metadata. Stored efficiently using prefix compression (< 64 bytes per namespace entry)
- Each node in the namespace tree has a corresponding read-write lock to manage concurrency
  - Because all metadata is stored in memory, the master can efficiently scan the entire state of the system periodically in the background
  - Master's memory capacity does not limit the size of the system

# The Operation Log

- Only persistent record of metadata
- Also serves as a logical timeline that defines the serialized order of concurrent operations
- Master recovers its state by replaying the operation log
  - To minimize startup time, the master checkpoints the log periodically
    - The checkpoint is represented in a B-tree like form, can be directly mapped into memory, but stored on disk
    - Checkpoints are created without delaying incoming requests to master, can be created in ~1 minute for a cluster with a few million files

# Why a Single Master? (1)

- The master now has global knowledge of the whole system, which drastically simplifies the design
- But the master is (hopefully) never the bottleneck
  - Clients never read and write file data through the master; client only requests from master which chunkservers to talk to
  - Master can also provide additional information about subsequent chunks to further reduce latency
  - Further reads of the same chunk don’t involve the master, either

# Why a Single Master? (2)

- Master state is also replicated for reliability on multiple machines, using the operation log and checkpoints
  - If master fails, GFS can start a new master process at any of these replicas and modify DNS alias accordingly
  - "Shadow" masters also provide read-only access to the file system, even when primary master is down
    - They read a replica of the operation log and apply the same sequence of changes
    - Not mirrors of master – they lag primary master by fractions of a second
    - This means we can still read up-to-date file contents while master is in recovery!

# Chunks and Chunkservers

- Files are divided into fixed-size ***chunks***, which has an immutable, globally unique 64-bit ***chunk handle***
  - By default, each chunk is replicated three times across multiple chunkservers (user can modify amount of replication)
- Chunkservers store the chunks on local disks as Linux files
  - Metadata per chunk is < 64 bytes (stored in master)
    - Current replica locations
    - Reference count (useful for copy-on-write)
    - Version number (for detecting stale replicas)

# Chunk Size

- 64 MB, a key design parameter (Much larger than most file systems.)
- Disadvantages:
  - Wasted space due to internal fragmentation
  - Small files consist of a few chunks, which then get lots of traffic from concurrent clients
    - This can be mitigated by increasing the replication factor
- Advantages:
  - Reduces clients' need to interact with master (reads/writes on the same chunk only require one request)
  - Since client is likely to perform many operations on a given chunk, keeping a persistent TCP connection to the chunkserver reduces network overhead
  - Reduces the size of the metadata stored in master → metadata can be entirely kept in memory

# Consistency Model

- Terminology:
  - ***consistent***: all clients will always see the same data, regardless of which replicas they read from
  - ***defined***: same as ***consistent*** and, furthermore, clients will see what the modification is in its entirety
- Guarantees:

| | Write | Record Append |
|---|---|---|
| Serial success | *defined* | *defined* interspersed with *inconsistent* |
| Concurrent successes | *consistent* but *undefined* | |
| Failure | *inconsistent* | |

# Data Modification in GFS

- After a sequence of modifications, if successful, then modified file region is guaranteed to be ***defined*** and contain data written by last modification
- GFS applies modification to a chunk in the same order on all its replicas
- A chunk is lost irreversibly **if and only if** all its replicas are lost before the master node can react, typically within minutes
  - even in this case, data is lost, not corrupted

# Record Appends

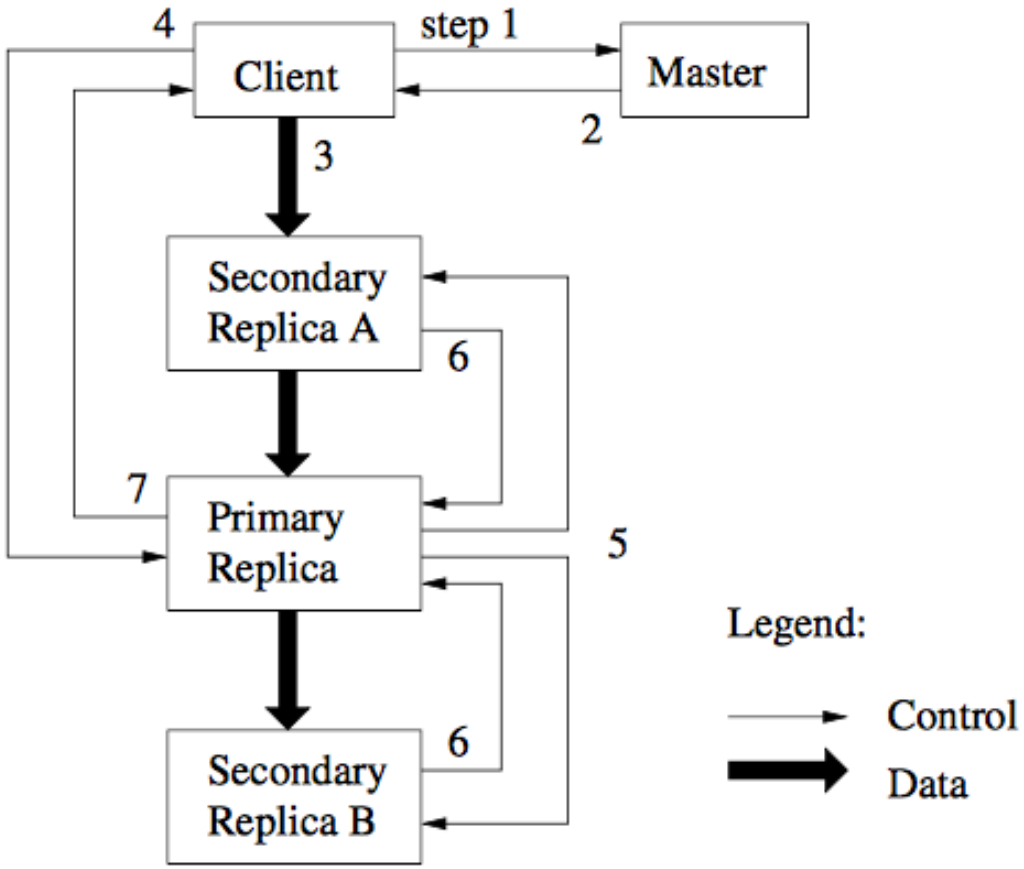
- A modification operation that guarantees that data (the “record”) will be appended atomically at least once – but at the offset of GFS’s choosing
  - The offset chosen by GFS is returned to the client so that the application is aware
- GFS may insert padding or record duplicates in between different record append operations
- Preferred that applications use this instead of write
  - Applications should also write self-validating records (e.g. checksumming) with unique IDs to handle padding/duplicates

# GFS Write Control and Data Flow (1)

- If the master receives a modification operation for a particular chunk:
  - Master finds the chunkservers that have the chunk and grants a ***chunk lease*** to one of them
    - This server is called the ***primary***, the other servers are called ***secondaries***
    - The primary determines the serialization order for all of the chunk's modifications, and the secondaries follow that order
  - After the lease expires (~60 seconds), master may grant primary status to a different server for that chunk
    - The master can, at times, revoke a lease (e.g. to disable modifications when file is being renamed)
    - As long as chunk is being modified, the primary can request an extension indefinitely
  - If master loses contact with primary, that's okay: just grant a new lease after the old one expires

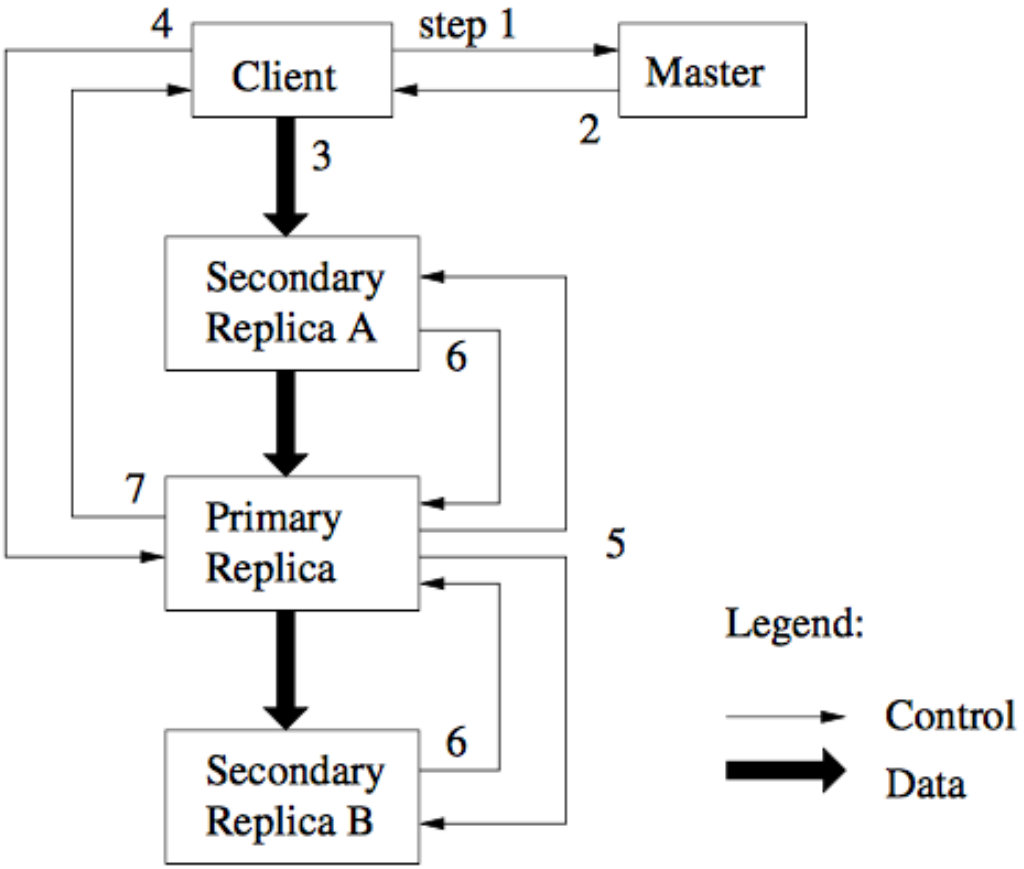
# GFS Write Control and Data Flow (2)

- 1. Client asks master for all chunkservers (including all secondaries)
- 2. Master grants a new lease on chunk, increases the chunk version number, tells all replicas to do the same. Replies to client. Client no longer has to talk to master
- 3. Client pushes data to all servers, not necessarily to primary first
- 4. Once data is acked, client sends write request to primary. Primary decides serialization order for all incoming modifications and applies them to the chunk



# GFS Write Control and Data Flow (3)

- 5. After finishing the modification, primary forwards write request and serialization order to secondaries, so they can apply modifications in same order. (If primary fails, this step is never reached.)
- 6. All secondaries reply back to the primary once they finish the modifications
- 7. Primary replies back to the client, either with success or error
  - If write succeeds at primary but fails at any of the secondaries, then we have inconsistent state → error returned to client
  - Client can retry steps (3) through (7)

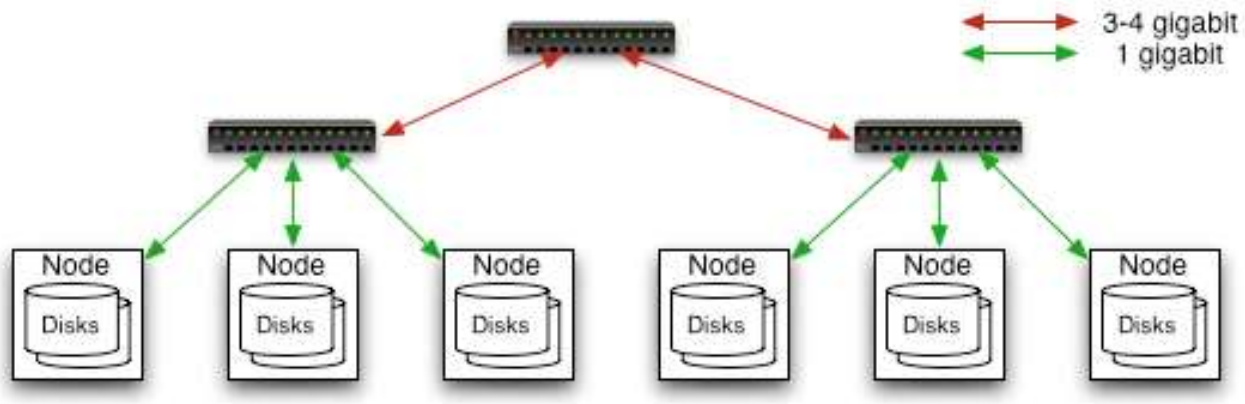


# 6 Hadoop File System (HDFS)

# Hadoop History


- Dec 2004 – Google GFS paper published
- July 2005 – Nutch uses MapReduce
- Feb 2006 – Starts as a Lucene subproject
- Apr 2007 – Yahoo! on 1000-node cluster
- Jan 2008 – An Apache Top Level Project
- May 2009 – Hadoop sorts Petabyte in 17 hours
- Aug 2010 – World's Largest Hadoop cluster at Facebook
  - 2900 nodes, 30+ PetaByte

# Hadoop Commodity Hardware




- **Typically in 2 level architecture**
  - Nodes are commodity PCs
  - 20-40 nodes/rack
  - Uplink from rack is 4 gigabit
  - Rack-internal is 1 gigabit

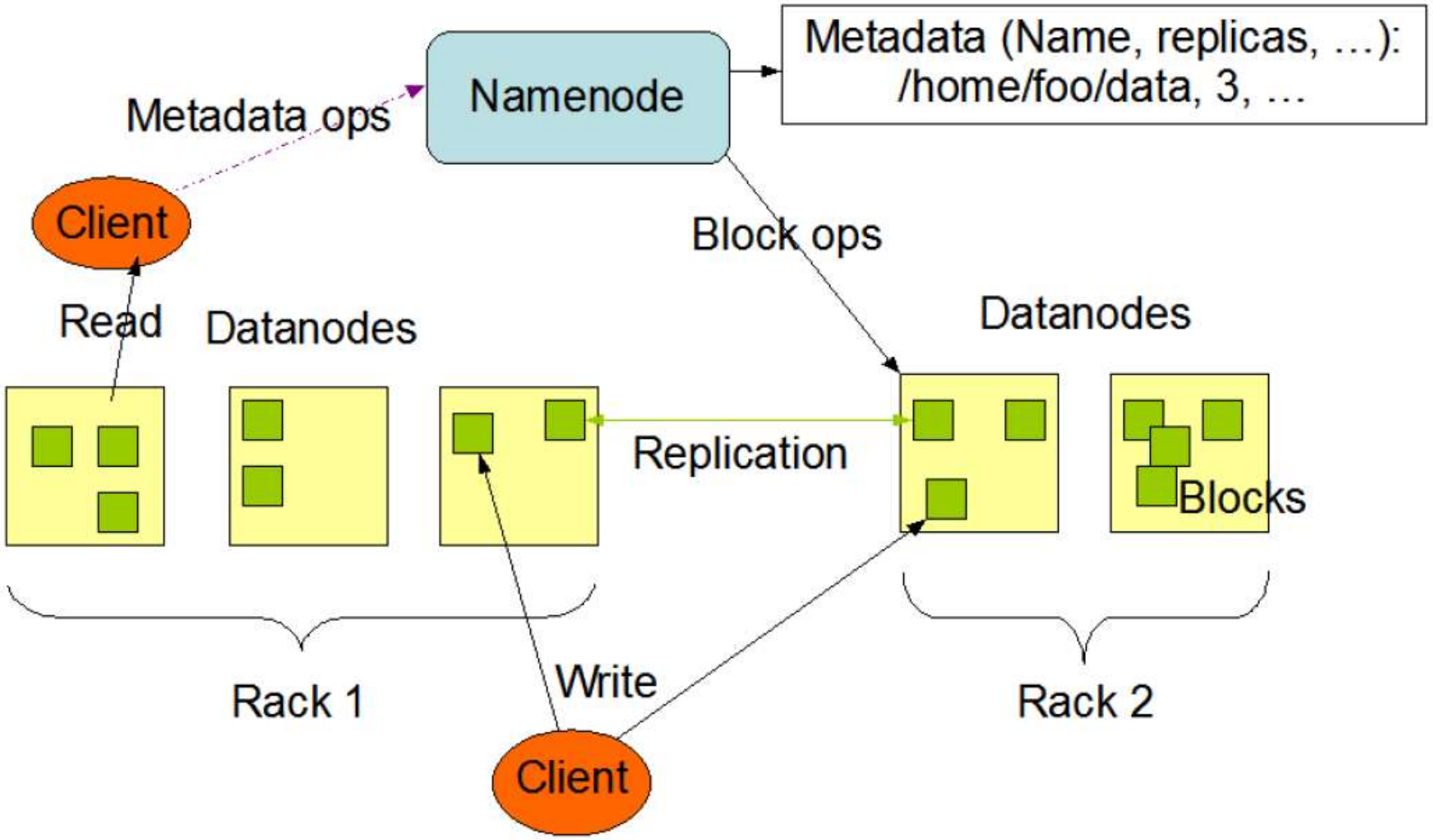
# Goals of Hadoop Distributed File System (HDFS)


- **Very Large Distributed File System**
  - 10K nodes, 1 billion files, 100 PB
- **Assumes Commodity Hardware**
  - Files are replicated to handle hardware failure
  - Detect failures and recovers from them
- **Optimized for Batch Processing**
  - Data locations exposed so that computations can move to where data resides
  - Provides very high aggregate bandwidth
- **User Space, runs on heterogeneous OS**

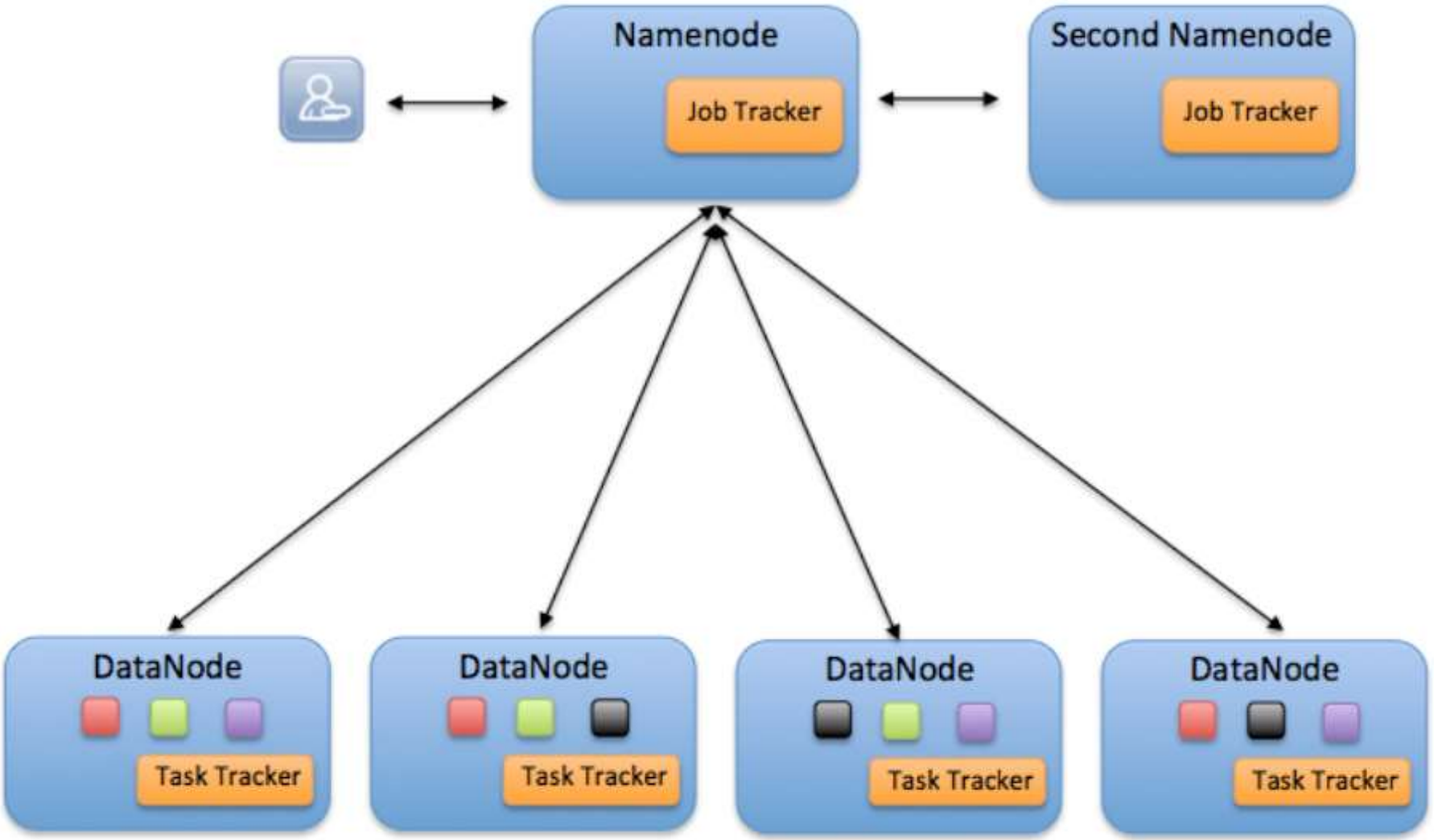
# Basic of HDFS


- **Single Namespace for entire cluster**
- **Data Coherency**
  - Write-once-read-many access model
  - Client can only append to existing files
- **Files are broken up into blocks**
  - Typically 128 - 256 MB block size
  - Each block replicated on multiple DataNodes
- **Intelligent Client**
  - Client can find location of blocks
  - Client accesses data directly from DataNode

# HDFS Architecture (1)



# HDFS Architecture (2)



# Namenode → Metadata


- **Meta-data in Memory**
  - The entire metadata is in main memory
  - No demand paging of meta-data
- **Types of Metadata**
  - List of files
  - List of Blocks for each file & file attributes
- **A Transaction Log**
  - Records file creations, file deletions, etc.

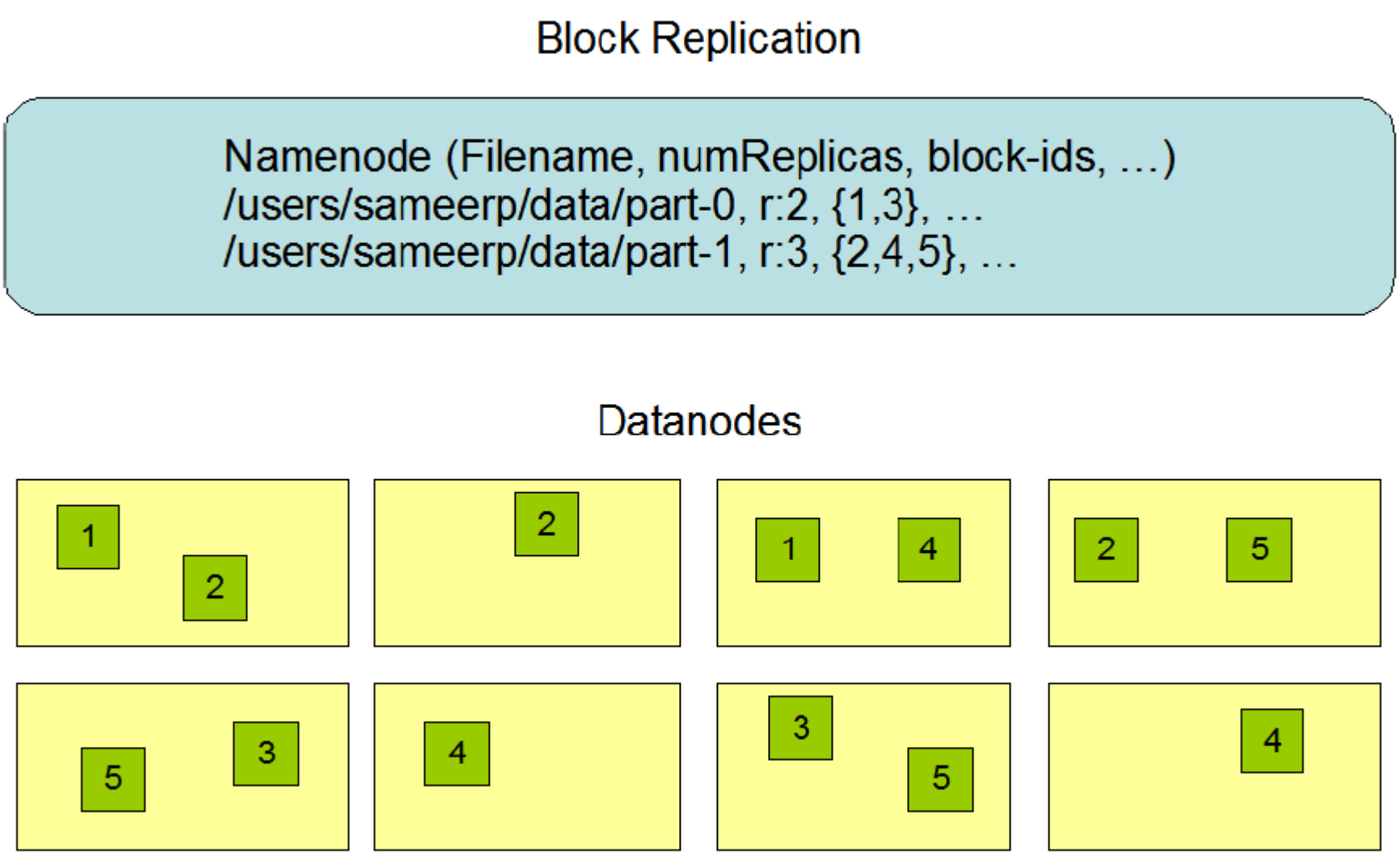
# Datanode


- **A Block Server**
  - Stores data in the local file system (e.g. ext3)
  - Stores meta-data of a block (e.g. CRC32)
  - Serves data and meta-data to Clients
  - Periodic validation of checksums
- **Block Report**
  - Periodically sends a report of all existing blocks to the NameNode (**heartbeats**)
- **Facilitates Pipelining of Data**
  - Forwards data to other specified DataNodes

# Block Placement

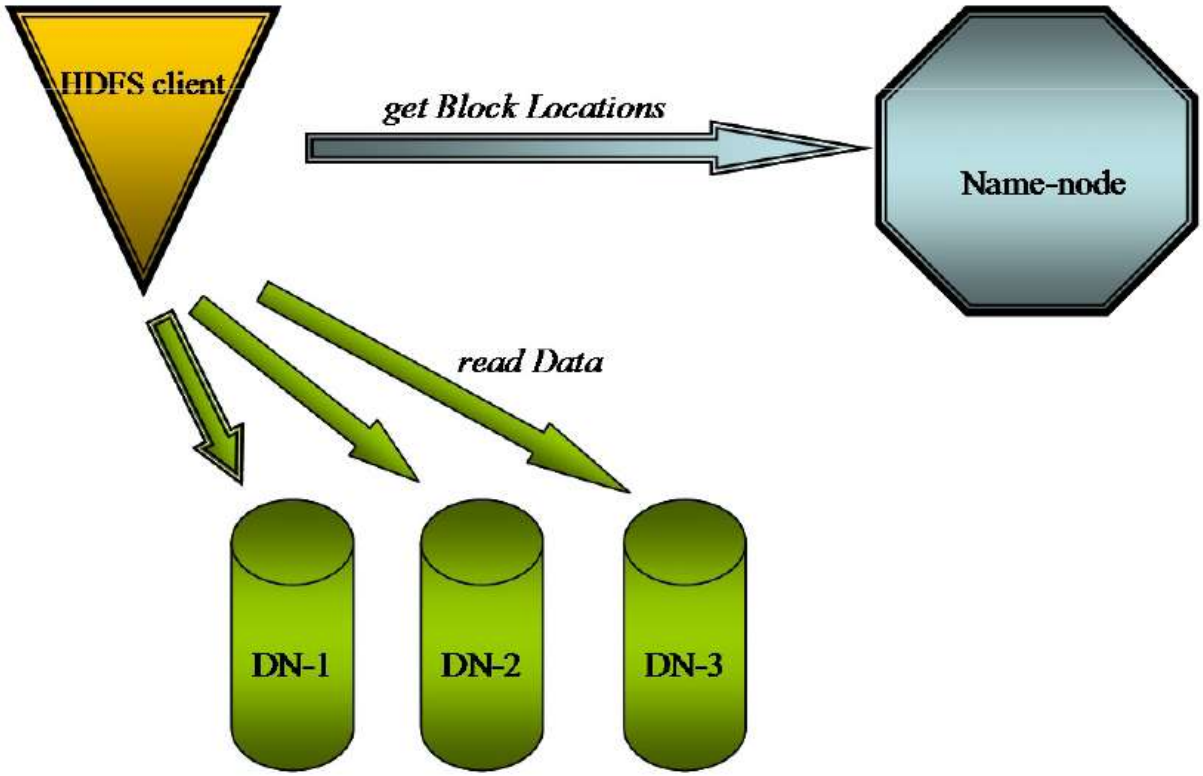

- **Current Strategy**
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
  - Additional replicas are randomly placed
- **Clients read from nearest replica**
- **Pluggable policy for placing block replicas**
  - Co-locate datasets that are often used together

# Block Replication



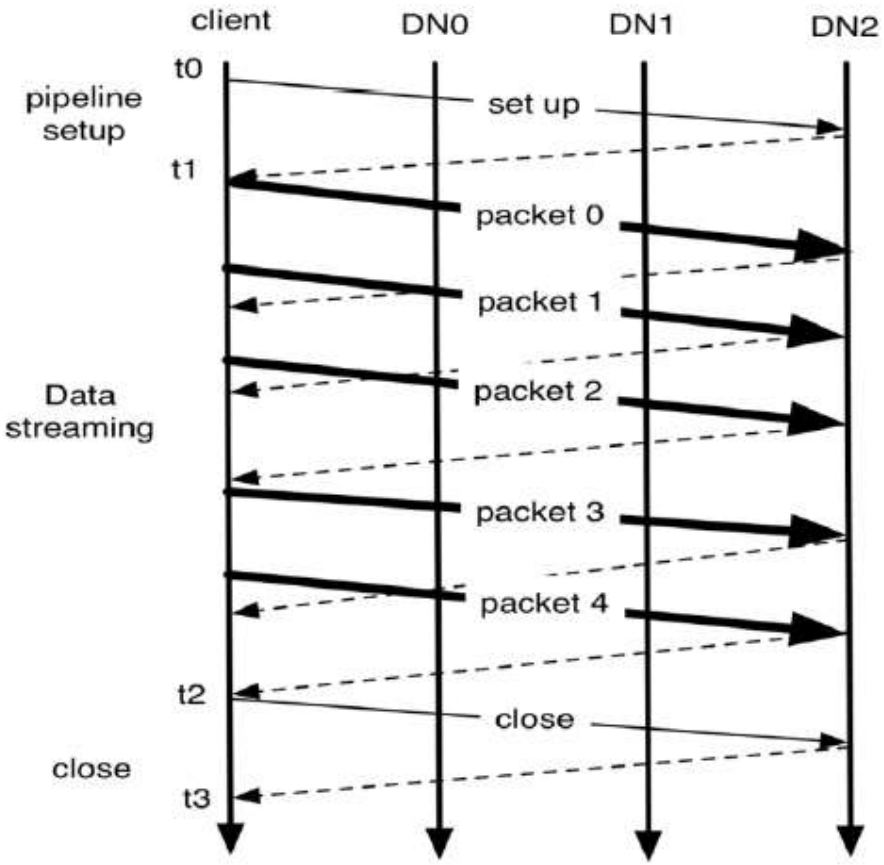
# HDFS Read

- To read a block, the client requests the list of replica locations from the NameNode
- Then pulling data from a replica on one of the DataNodes



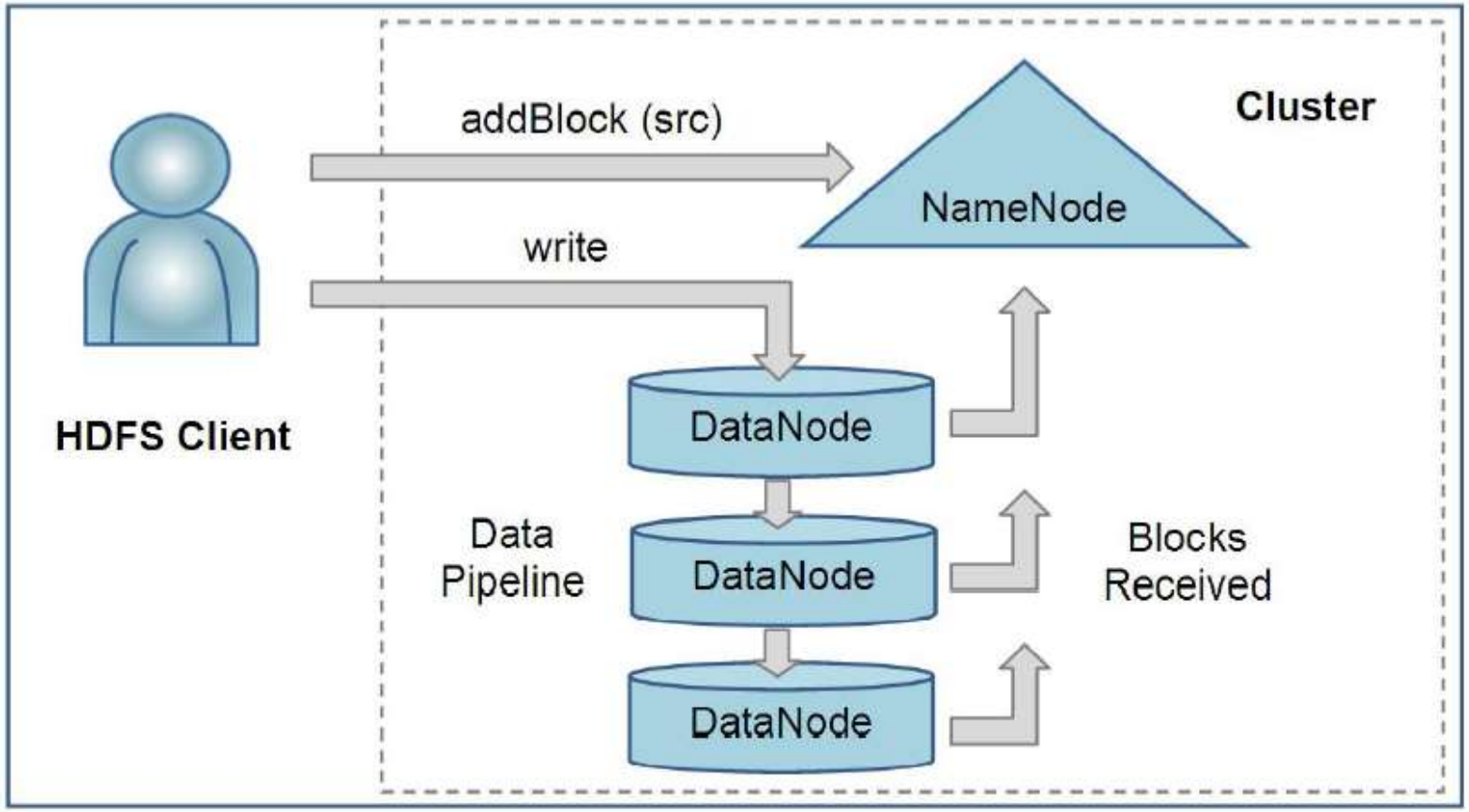
# Data Pipelining

- Client writes block to the first DataNode
  - The first DataNode forwards the data to the next
- DataNode in the Pipeline, and so on
  - When all replicas are written, the Client moves on to write the next block in file
- Not good for latency sensitive applications



# HDFS Write

- To write a block of a file, the client requests a list of candidate DataNodes from the NameNode, and organizes a write pipeline.

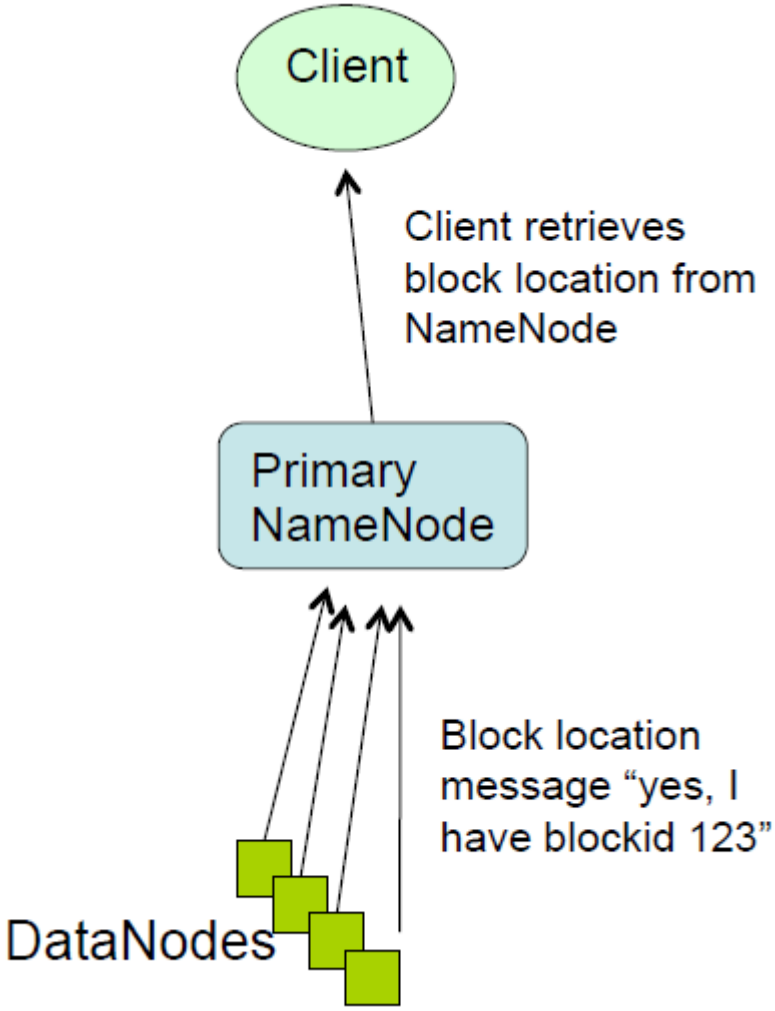


# Namenode failure


- **A Single Point of Failure**
- **Transaction Log stored in multiple directories**
  - A directory on the local file system
  - A directory on a remote file system (NFS/CIFS)
- **This is a problem with 24 x 7 operations**
  - AvatarNode comes to the rescue

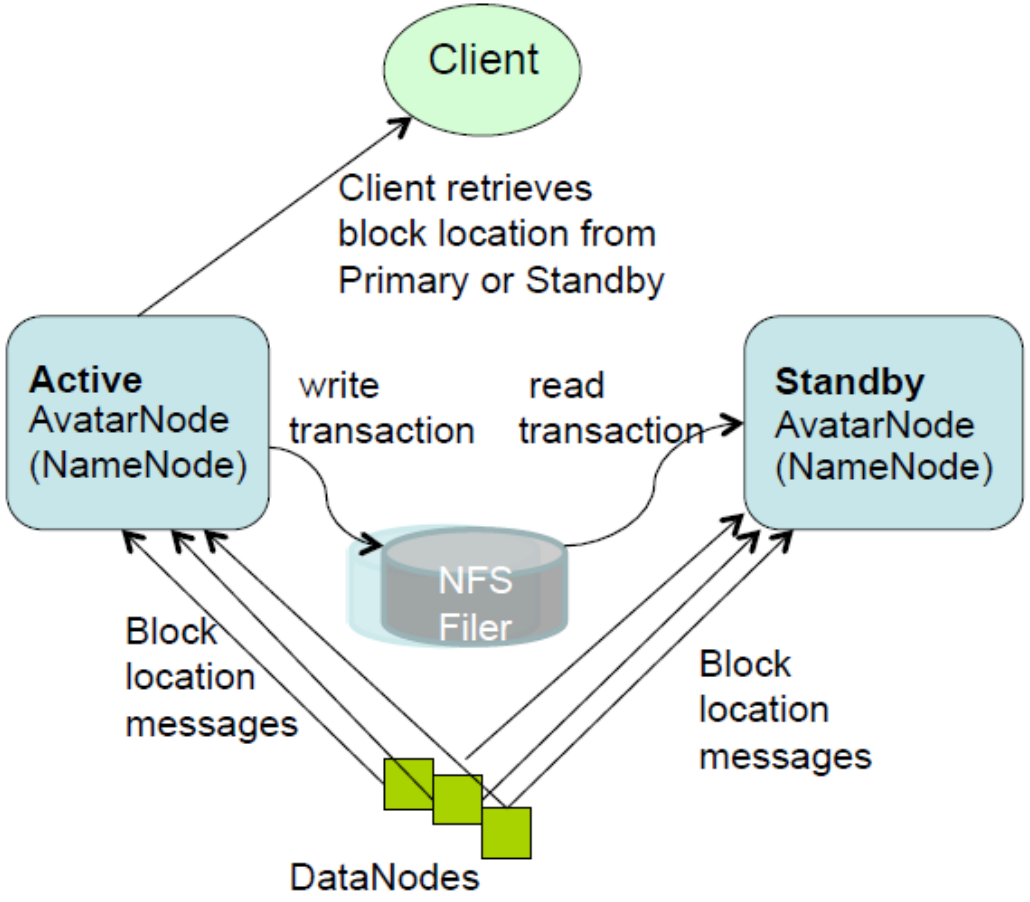
# NameNode High Availability Challenges

- DataNodes send block location information to only one NameNode
- NameNode needs block locations in memory to serve clients
- The in-memory metadata for 100 million files could be 60 GB, huge!



# NameNode High Availability AvatarNode

- **Active-Standby Pair**
  - Coordinated via zookeeper
  - Failover in few seconds
  - Wrapper over NameNode
- **Active AvatarNode**
  - Writes transaction log to filer
- **Standby AvatarNode**
  - Reads transactions from filer
  - Latest metadata in memory

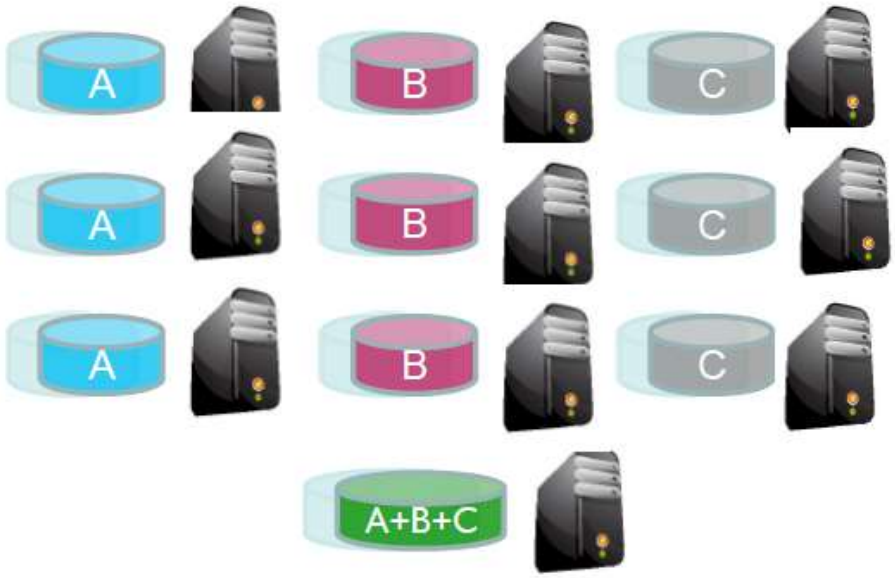


# Rebalancer


- **Goal: % disk full on DataNodes should be similar**
  - Usually run when new DataNodes are added
  - Cluster is online when Rebalancer is active
  - Rebalancer is throttled to avoid network congestion
- **Disadvantages**
  - Does not rebalance based on access patterns or load
  - No support for automatic handling of hotspots of data

# HDFS RAID

- Triplicate every data block
- Background encoding
  - Combine third replica of blocks from a single file to create parity block
  - Remove third replica
- RaidNode
  - Auto fix of failed replicas
- Reed Solomon encoding for old files




A file with three blocks A, B and C

# 7 Microsoft Azure

# Microsoft Azure Storage

Microsoft Azure

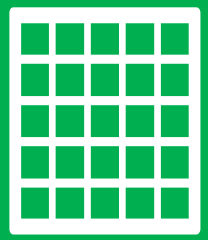
- **Blobs** – File system in the cloud
- **Tables** – Massively scalable structured storage
- **Queues** – Reliable storage and delivery of messages
- **Drives** – Durable NTFS volumes for Windows Azure applications

## Blobs

Simple named files along with metadata for the file.

## Drives

Durable NTFS volumes for Windows Azure applications to use. Based on Blobs.



## Tables

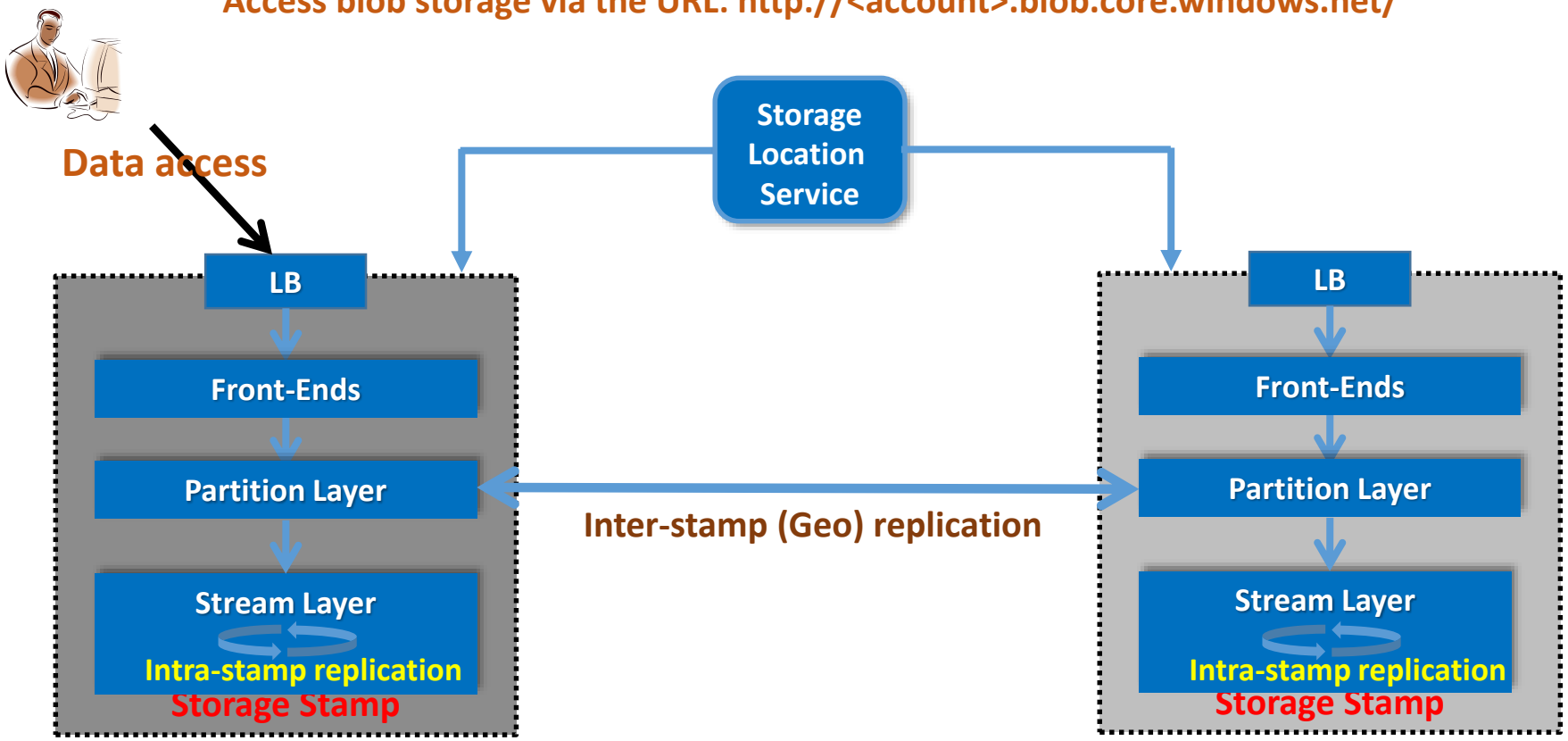
Structured storage. A table is a set of entities; an entity is a set of properties.

## Queues

Reliable storage and delivery of messages for an application.

# Windows Azure Storage Stamps

Microsoft Azure

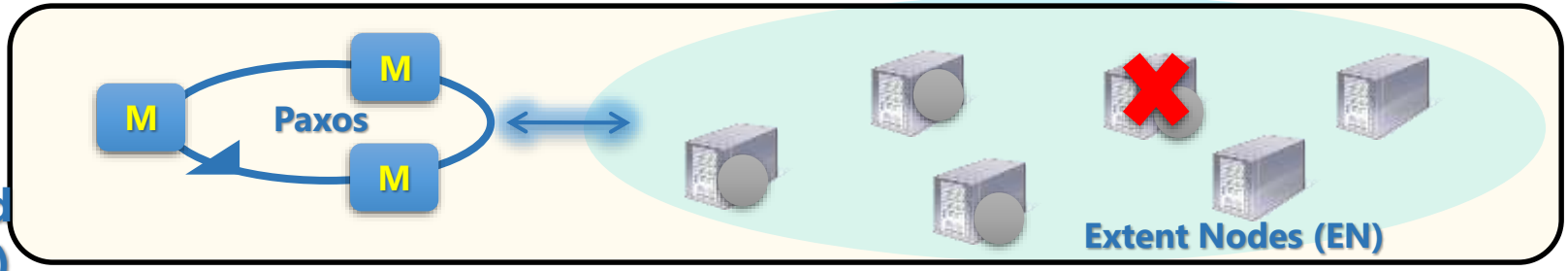


# Storage Stamp Architecture – Stream Layer

Microsoft Azure

- Append-only distributed file system
- All data from the Partition Layer is stored into files (extents) in the Stream layer
- An extent is replicated 3 times across different fault and upgrade domains
  - With random selection for where to place replicas for fast MTTR
- Checksum all stored data
  - Verified on every client read
  - Scrubbed every few days
- Re-replicate on disk/node/rack failure or checksum mismatch

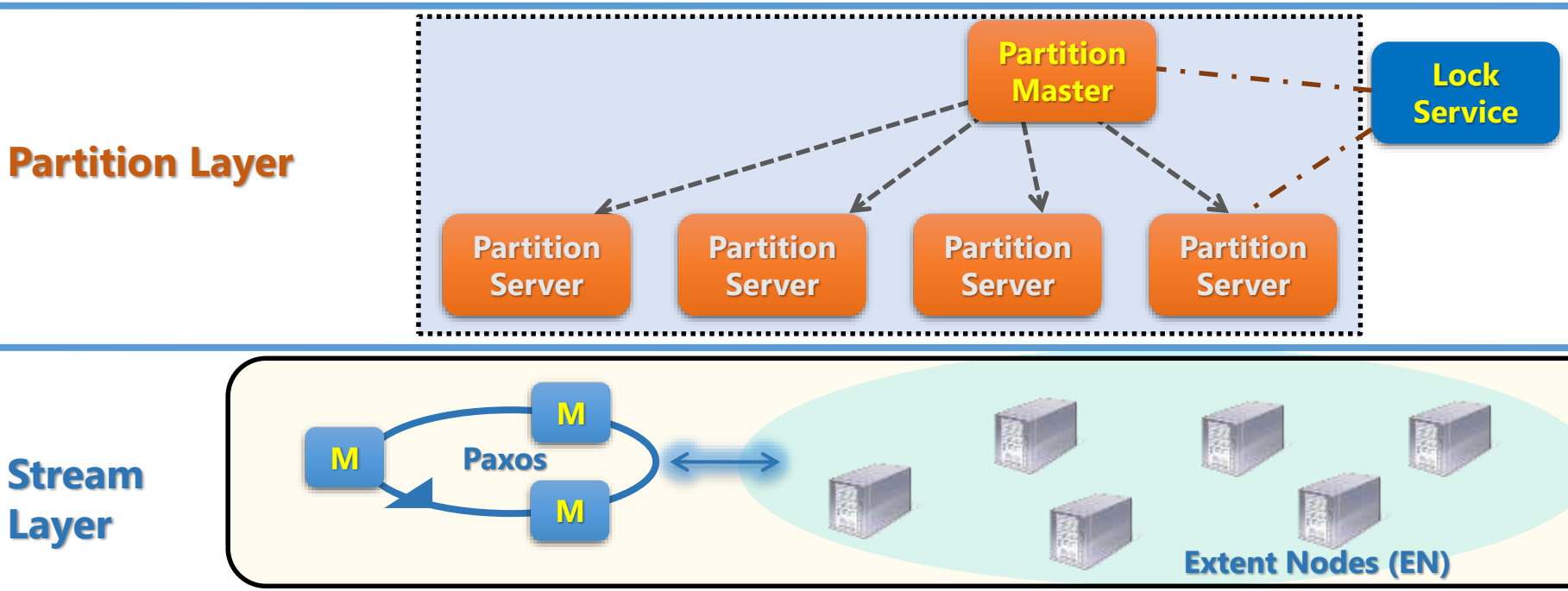
**Stream Layer (Distributed File System)**



# Storage Stamp Architecture – Partition Layer

Microsoft Azure

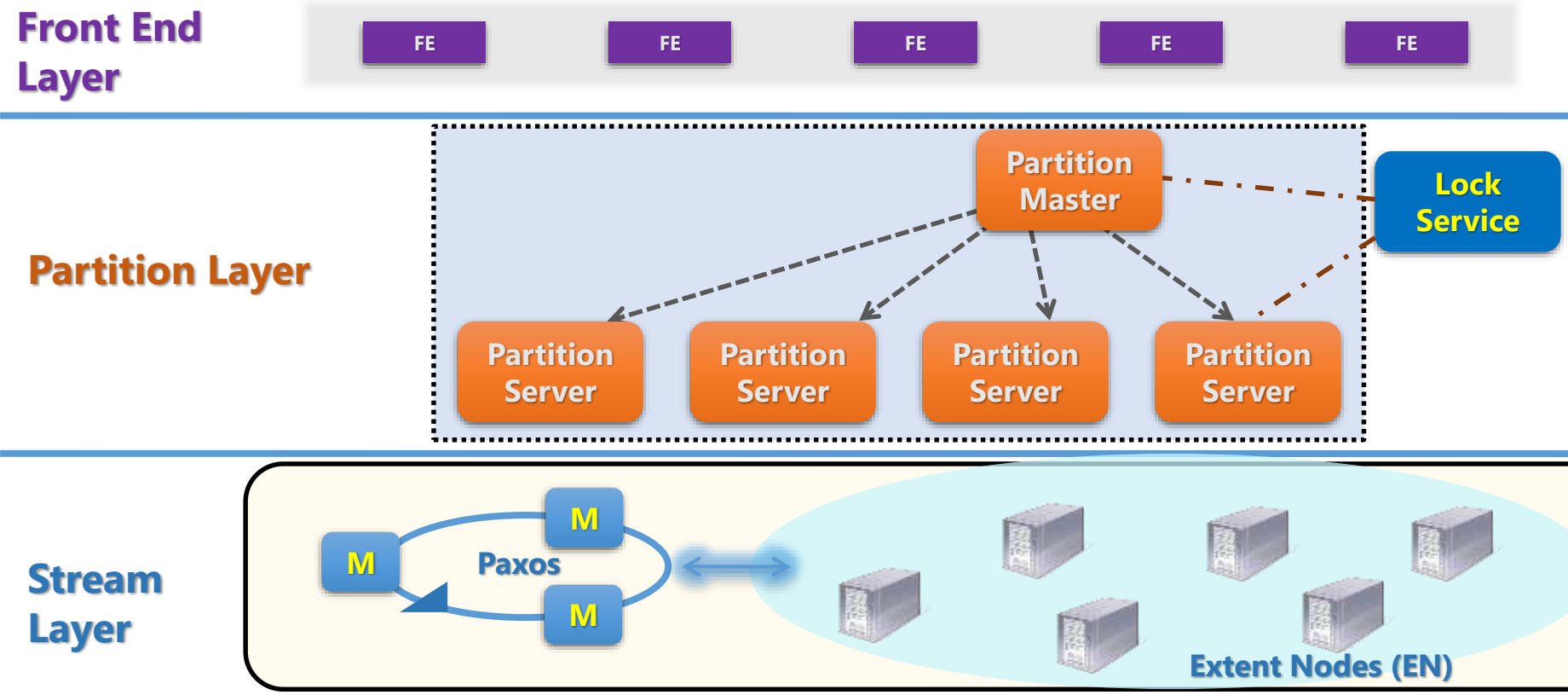
- Provide transaction semantics and strong consistency for Blobs, Tables and Queues
- Stores and reads the objects to/from extents in the Stream layer
- Provides inter-stamp (geo) replication by shipping logs to other stamps
- Scalable object index via partitioning



# Storage Stamp Architecture – Front End Layer

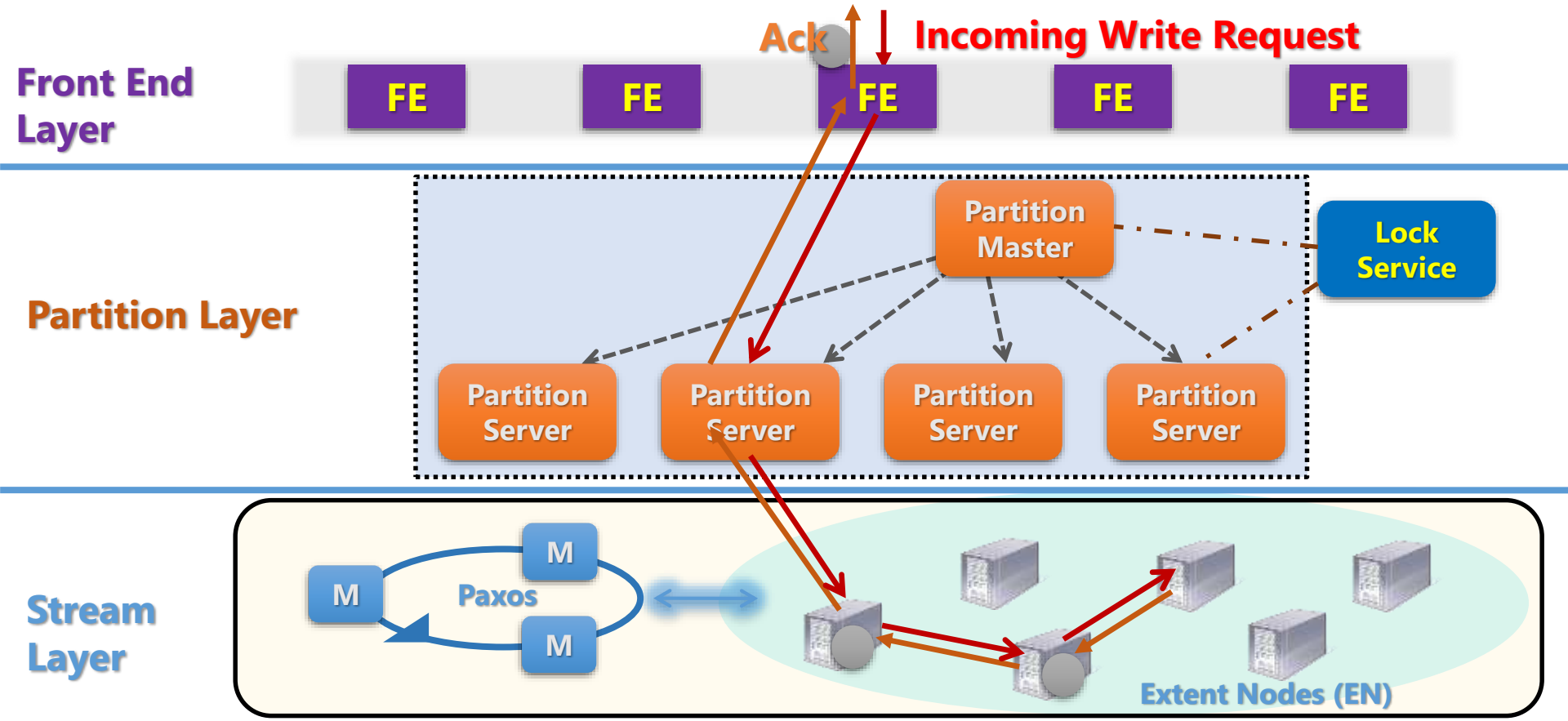
Microsoft Azure

- Stateless Servers
- Authentication + authorization
- Request routing



# Storage Stamp Architecture – Request

Microsoft Azure



# Partition Layer – Scalable Object Index

Microsoft Azure

- 100s of Billions of blobs, entities, messages across all accounts can be stored in a single stamp
  - Need to efficiently enumerate, query, get, and update them
  - Traffic pattern can be highly dynamic
    - Hot objects, peak load, traffic bursts, etc

- Need a scalable index for the objects that can
  - Spread the index across 100s of servers
  - Dynamically load balance
    - Dynamically change what servers are serving each part of the index based on load

# Scalable Object Index via Partitioning

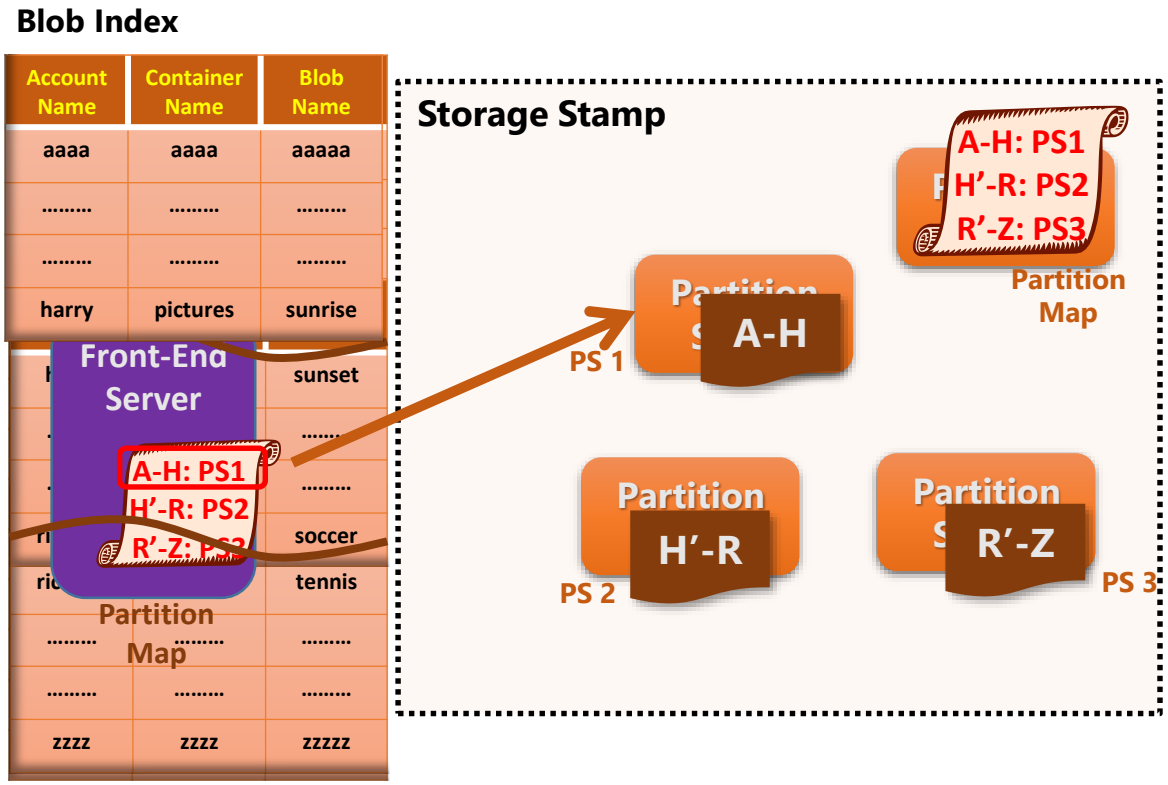
Microsoft Azure

- Partition Layer maintains an internal Object Index Table for each data abstraction
  - Blob Index: contains all blob objects for all accounts in a stamp
  - Table Entity Index: contains all entities for all accounts in a stamp
  - Queue Message Index: contains all messages for all accounts in a stamp
- Scalability is provided for each Object Index
  - Monitor load to each part of the index to determine hot spots
  - Index is dynamically split into thousands of Index RangePartitions based on load
  - Index RangePartitions are automatically load balanced across servers to quickly adapt to changes in load

# Partition Layer – Index Range Partitioning

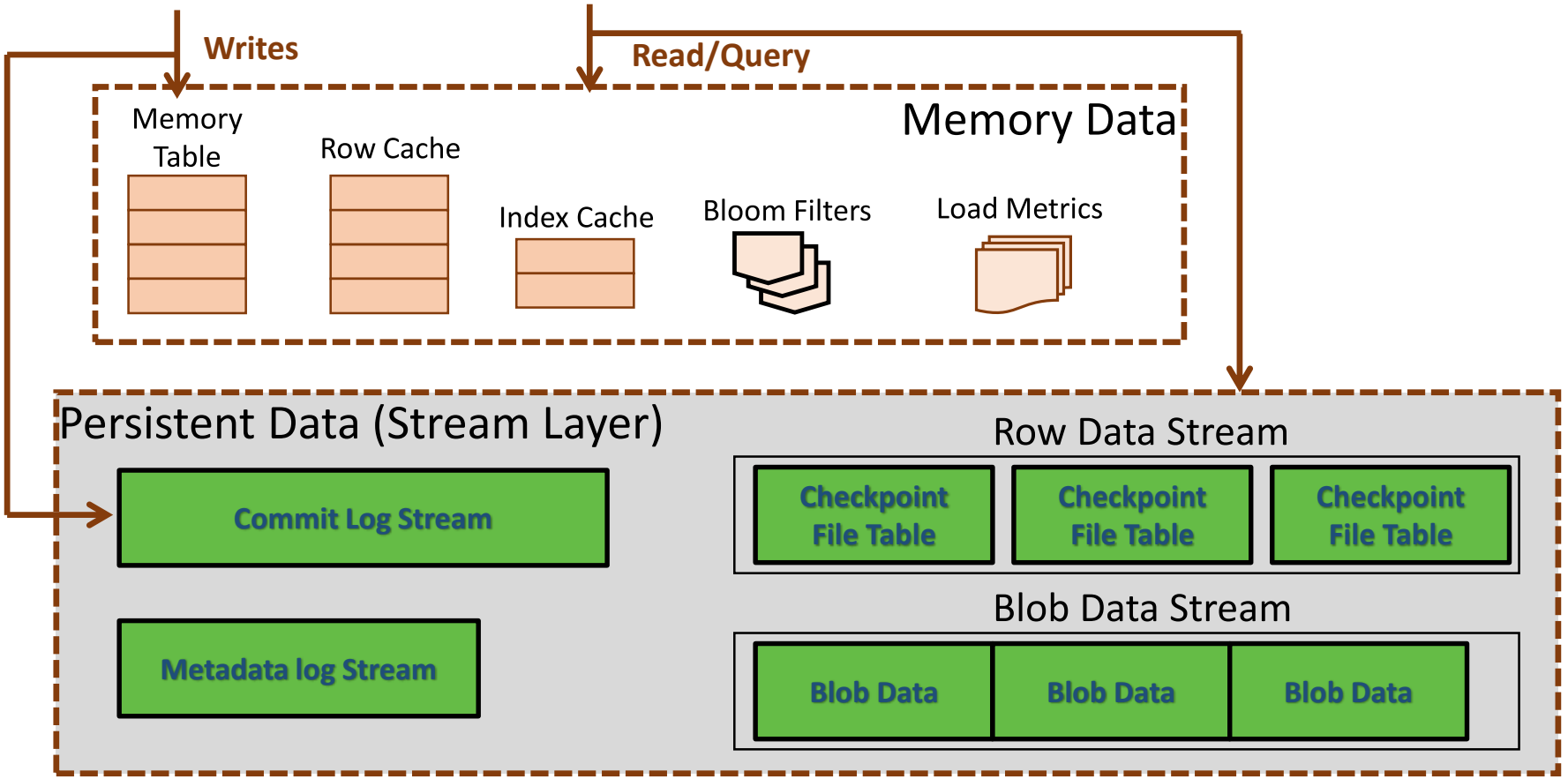
Microsoft Azure

- Split index into RangePartitions based on load
- Split at PartitionKey boundaries
- PartitionMap tracks Index RangePartition assignment to partition servers
- Front-End caches the PartitionMap to route user requests
- Each part of the index is assigned to only one Partition Server at a time



# Each RangePartition – Log Structured Merge Tree

Microsoft Azure



# Stream Layer

Microsoft Azure

- Append-Only Distributed File System
- Streams are very large files
  - Has file system like directory namespace
- Stream Operations
  - Open, Close, Delete Streams
  - Rename Streams
  - Concatenate Streams together
  - Append for writing
  - Random reads

# Stream Layer Concepts

Microsoft Azure

## Block

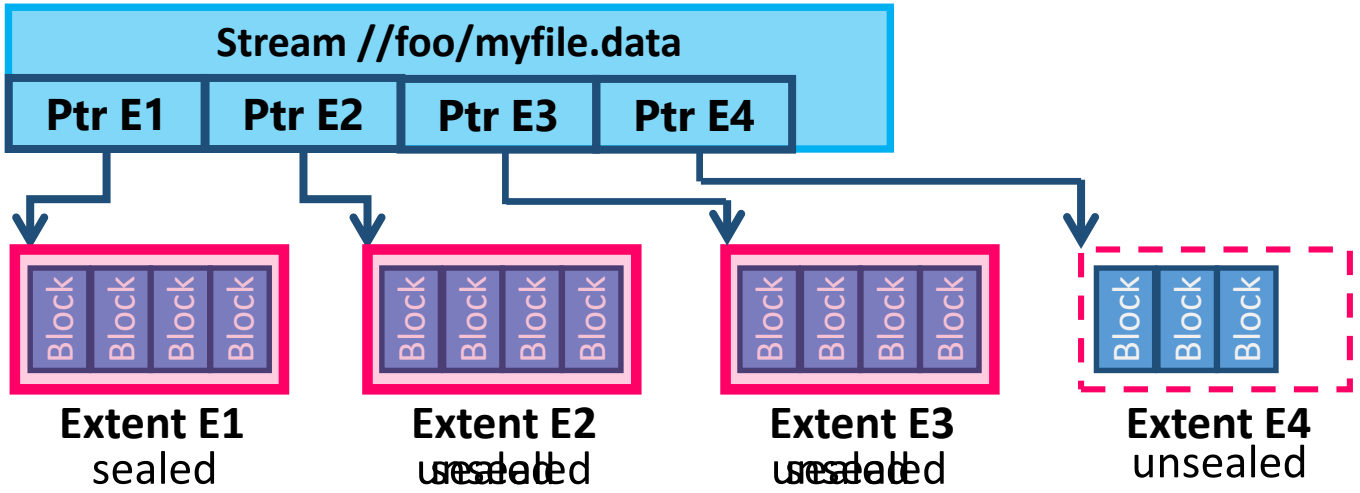
- Min unit of write/read
- Checksum
- Up to N bytes (e.g. 4MB)

## Extent

- Unit of replication
- Sequence of blocks
- Size limit (e.g. 1GB)
- Sealed/unsealed

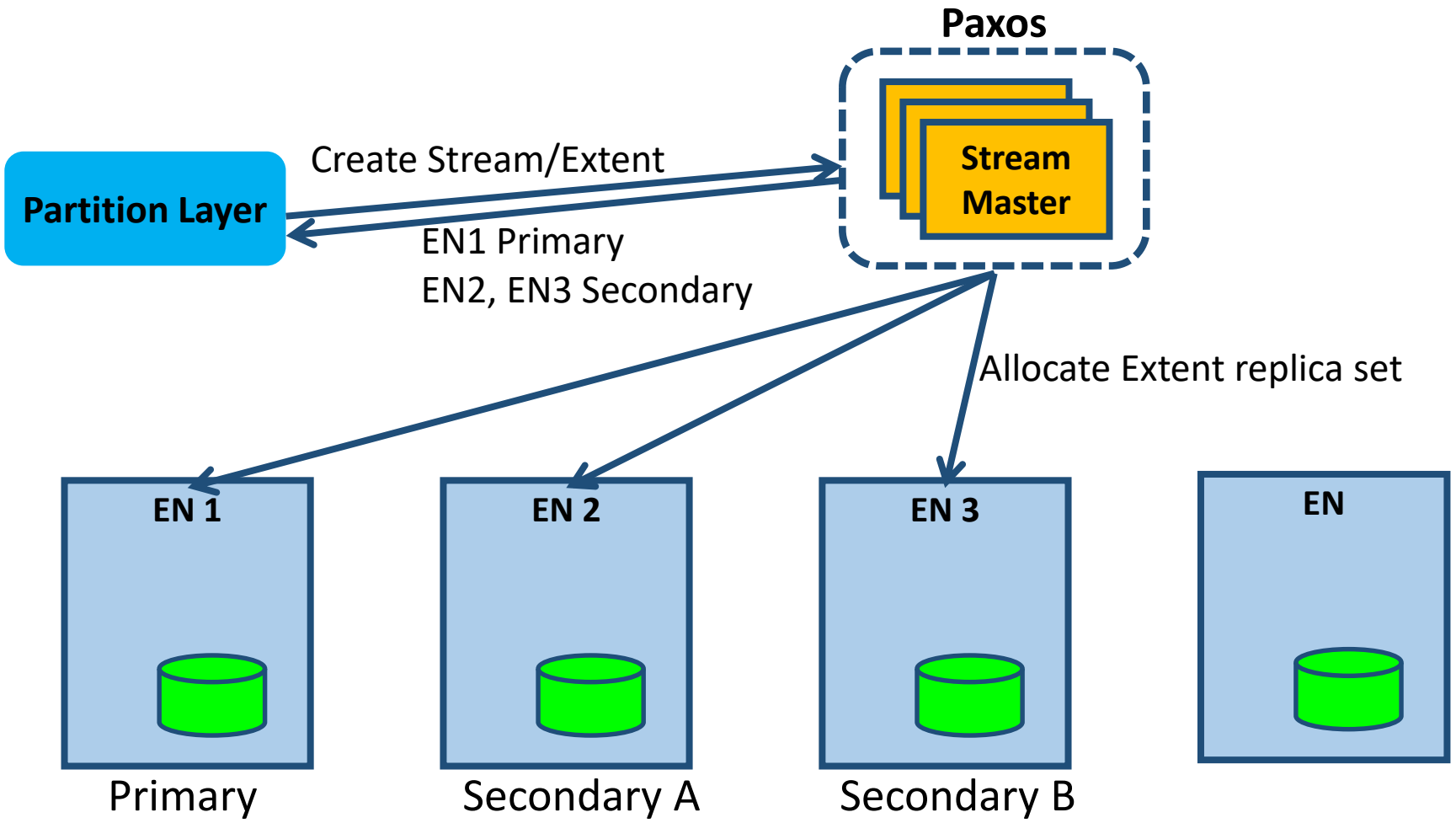
## Stream

- Hierarchical namespace
- Ordered list of pointers to extents
- Append/Concatenate



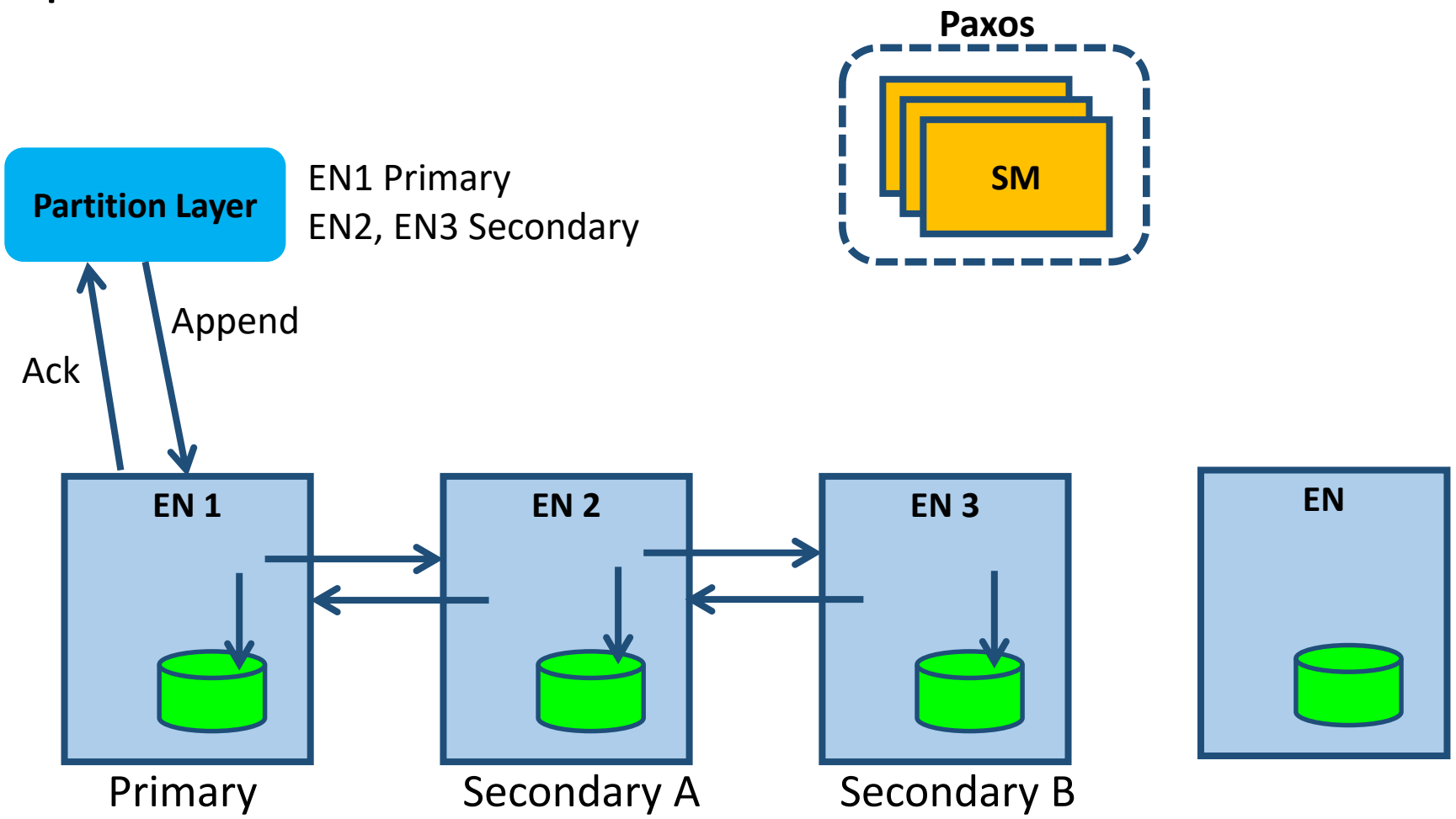
# Creating an Extent

Microsoft Azure



# Replication Flow

Microsoft Azure



# Thank you!