Big Data Processing Technologies

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

Schedule

- lec1: Introduction on big data and cloud computing
- lec2: Introduction on data storage
- lec3: Data reliability (Replication/Archive/EC)
- lec4: Data consistency problem
- lec5: Block storage and file storage
- lec6: Object-based storage
- lec7: Distributed file system
- lec8: Metadata management

Collaborators

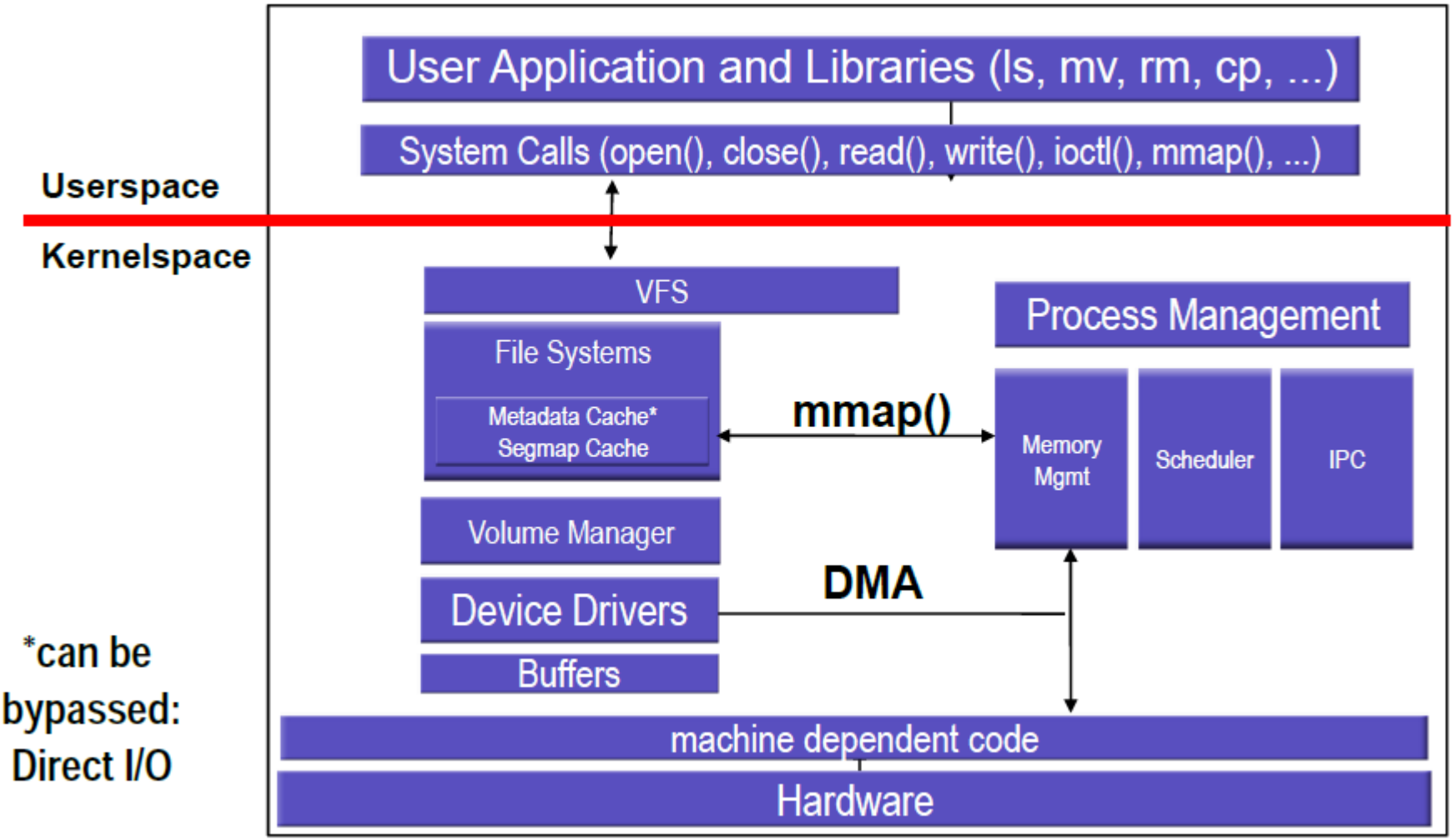
Google 谷歌

阿里云 aliyun.com

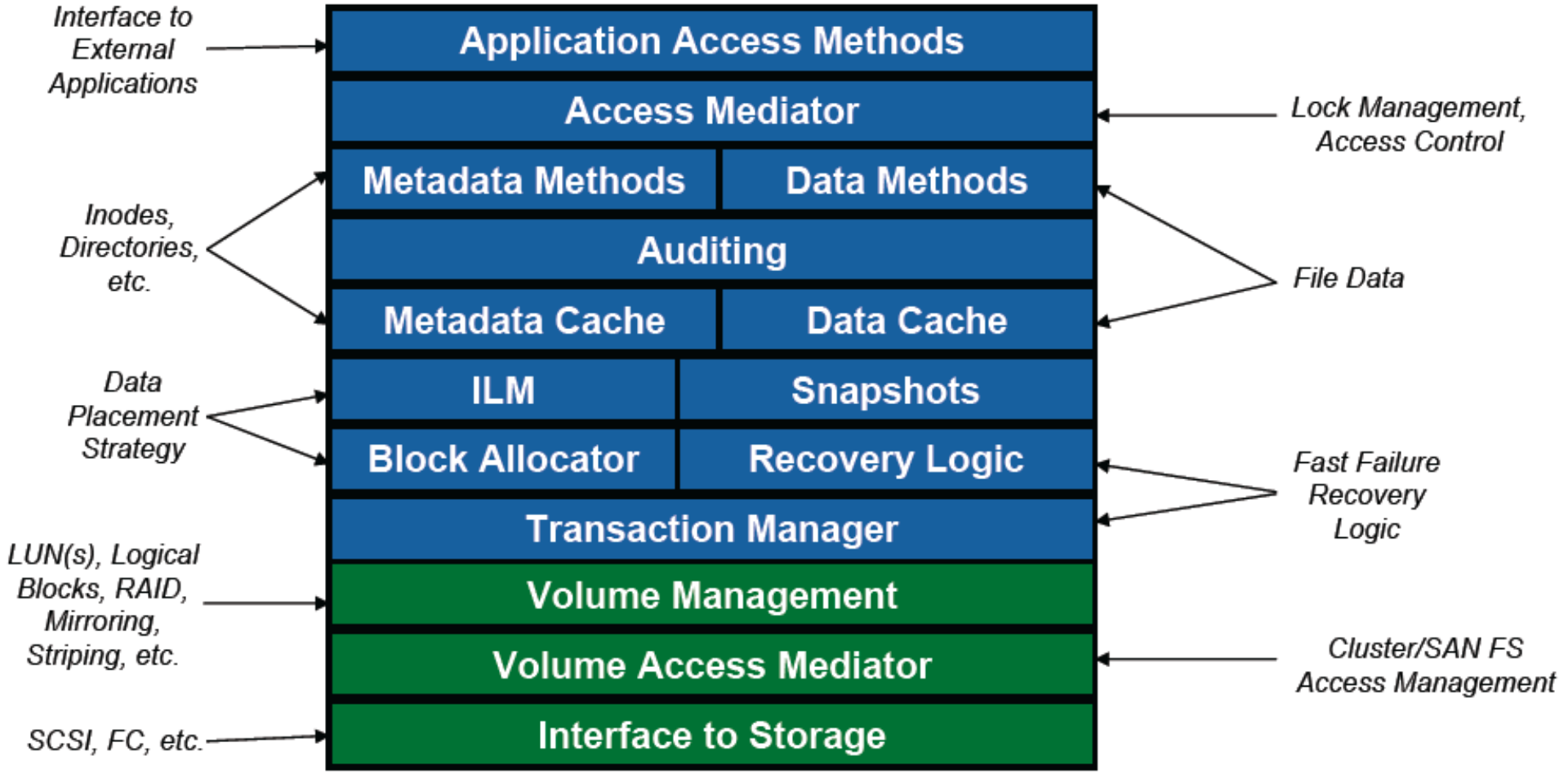
DELL EMC

1 Distributed File System (DFS)

File System & Operating Systems



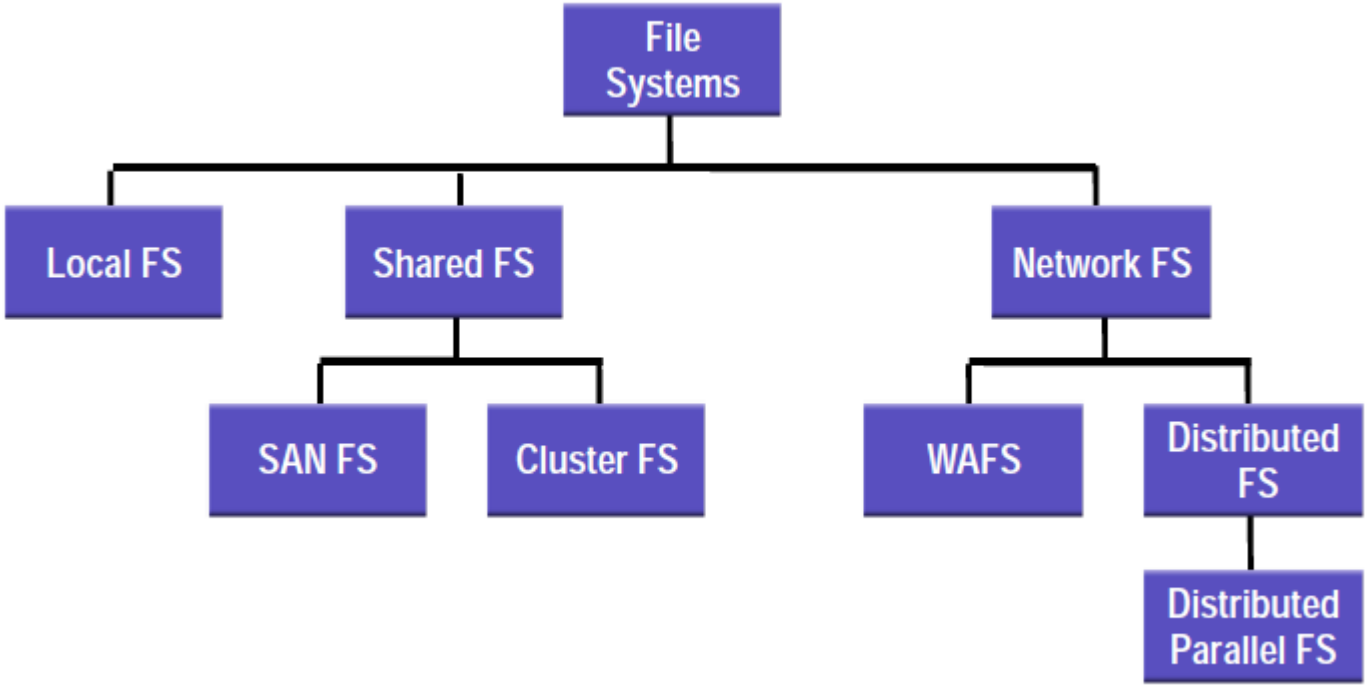
File System Component



The File Systems Evolution

- **File systems evolved over time**
- Starting with local file system over time, additional file systems appeared focusing on specialized requirements such as data sharing, remote file access, distributed file access, parallel file access, HPC, archiving, etc.

The File Systems Taxonomy



File System Types

- **Local File System**
  - Host-based, single operating system
  - Co-located with application server
  - Many types with unique formats, feature mix
- **Shared (SAN and Clustered) File Systems**
  - Host-based file systems
  - Hosts access all data
  - Co-located with application server for performance
- **Distributed File System**
  - Remote, network-access
  - Semantics are limited subset of local file system
  - Cooperating file servers
  - Can include integrated replication
  - Clustered DFS/Wide Area File System

Evaluating File Systems (1)

- **Does it fit the Application Characteristics**
  - Does the application even support the file system?
  - Is it optimized for the type of operations that are important to the application?
- **Performance & Scalability**
  - Does the file system meet the latency and throughput requirements?
  - Can it scale up to the expected workload and deal with growth?
  - Can it support the number of files and total storage needed?
- **Data Management**
  - What Kind of features does it include? Backup, Replication, Snapshots, Information Lifecycle Management (ILM), etc.

Evaluating File Systems (2)

- **Security**
  - Does it conform to the security requirements of your company?
  - Does it integrate with your security services?
  - Does it have Auditing, Access control and at what granularity?
- **Ease of Use**
  - Does it require training the end users or changing applications to perform well?
  - Can it be easily administered in small and large deployments?
  - Does it have centralized monitoring, reporting?
  - How hard is it to recover from a software or hardware failure and how long does it take?
  - How hard is it to upgrade or downgrade the software and is it live?

Application Characteristics

- **Typical applications**
  - (A) OLTP
  - (B) Small Data Set
  - (C) Home Directory
  - (D) Large Scale Streaming
  - (E) High Frequency Metadata Update (small file create/delete)

Workload Profiles	(A)	(B)	(C)	(D)	(E)
1. Latency Sensitive	High	Med	Low	Low	High
2. Throughput	High read/write	High read	Low	High read	High write
3. Concurrent sharing	High	High	Low	High read	Low
4. Caching (re-read rate)	High	High	High	Low	Low

Performance & Scalability

- **Performance**
  - Throughput
  - Read/write access patterns
  - Impact of data protection mechanisms, operations
- **Scalability**
  - Number of files, directories, file systems
  - Performance, recovery time
  - Simultaneous and active users

Data Management (1)

- **Backup**
  - Performance
  - Backup vendors; local agent vs. network-based
  - Data deduplication → backup once
- **Replication**
  - Multiple read-only copies
  - Optimization for performance over network
  - Data deduplication → transfer once
- **Quotas**
  - Granularity： User/Group/Directory tree quotas
  - Extended quota features
  - Ease of set up
  - Local vs. external servers

Data Management (2)

- **Information Lifecycle Management (ILM)**
  - Lots of features, differing definitions
  - Can enforce compliance and auditing rules
  - Cost & performance vs. impact of lost/altered data

Security Considerations (1)

- **Authentication**
  - Support and to what degree
- **Authorization**
  - Granularity by access types
  - Need for client-side software
  - Performance impact of large scale ACL changes
- **Auditing**
  - Controls
  - Audit log full condition
  - Login vs. login attempt vs. data access

Security Considerations (2)

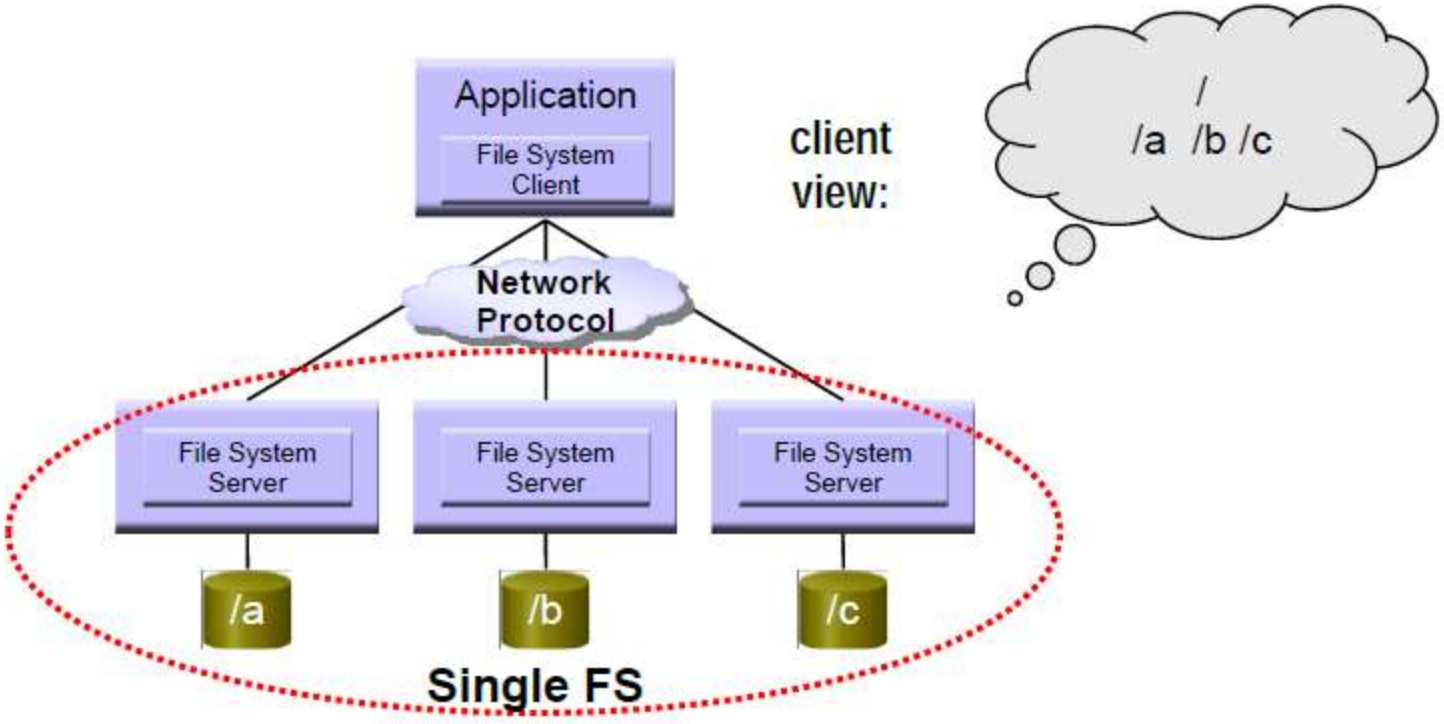
- **Virus scanning**
  - Preferred vendor supported?
  - Performance & scalability
  - External vs. file server-side virus scanning
- **Vulnerabilities**
  - Security & data integrity vulnerabilities vs. performance
  - Compromised file system (one client, one file server)
  - Detection
  - Packet sniffing

Ease of Use

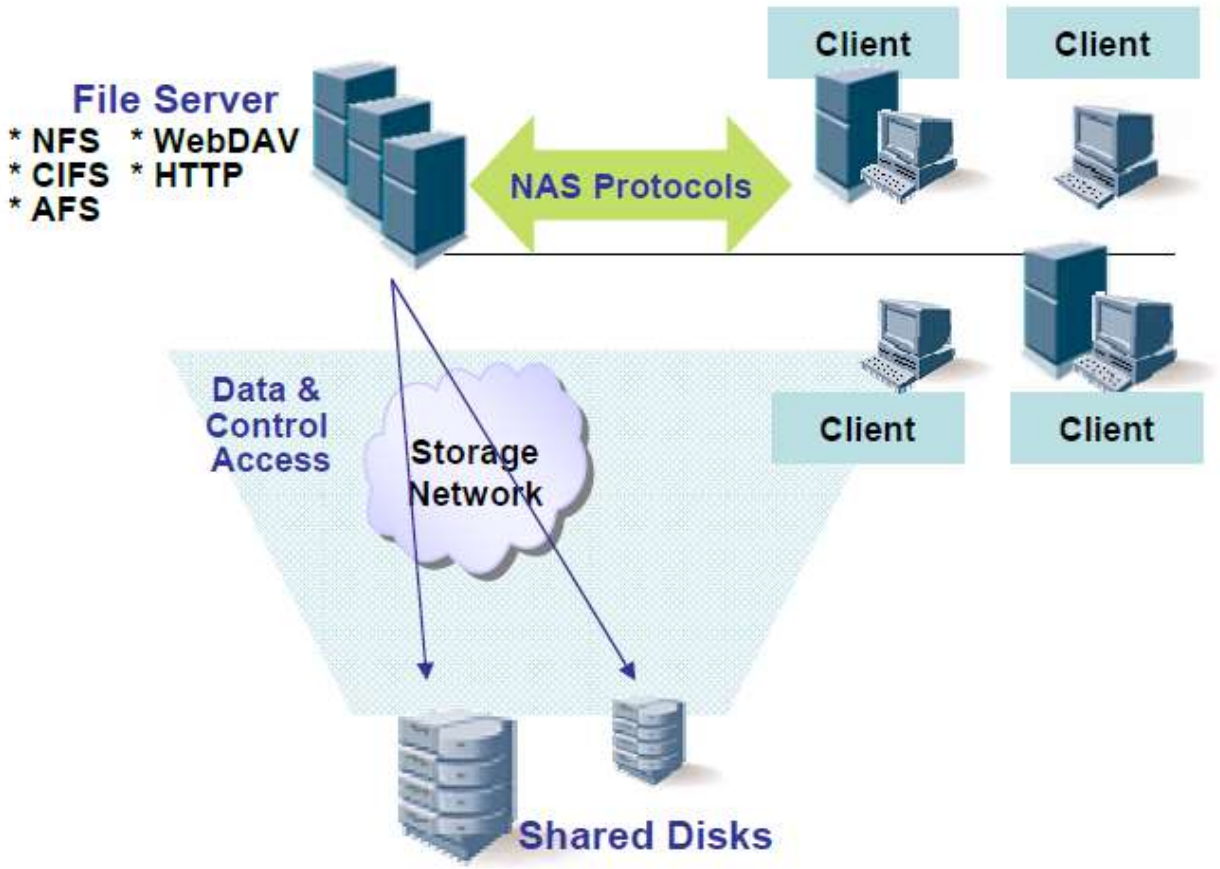
- **End-User**
  - Local file system vs. Distributed File System
- **Deployment & Maintenance**
  - Implementation
  - Scalability of management
  - File system migration
  - Automatic provisioning
  - Centralized monitoring, reporting
  - Hardware failure recovery
  - Performance monitoring

Distributed File System

- **A distributed file system is a network file system whose clients, servers, and storage devices are dispersed among the machines of a distributed system or intranet.**



Distributed File System (NAS & SAN Environment)

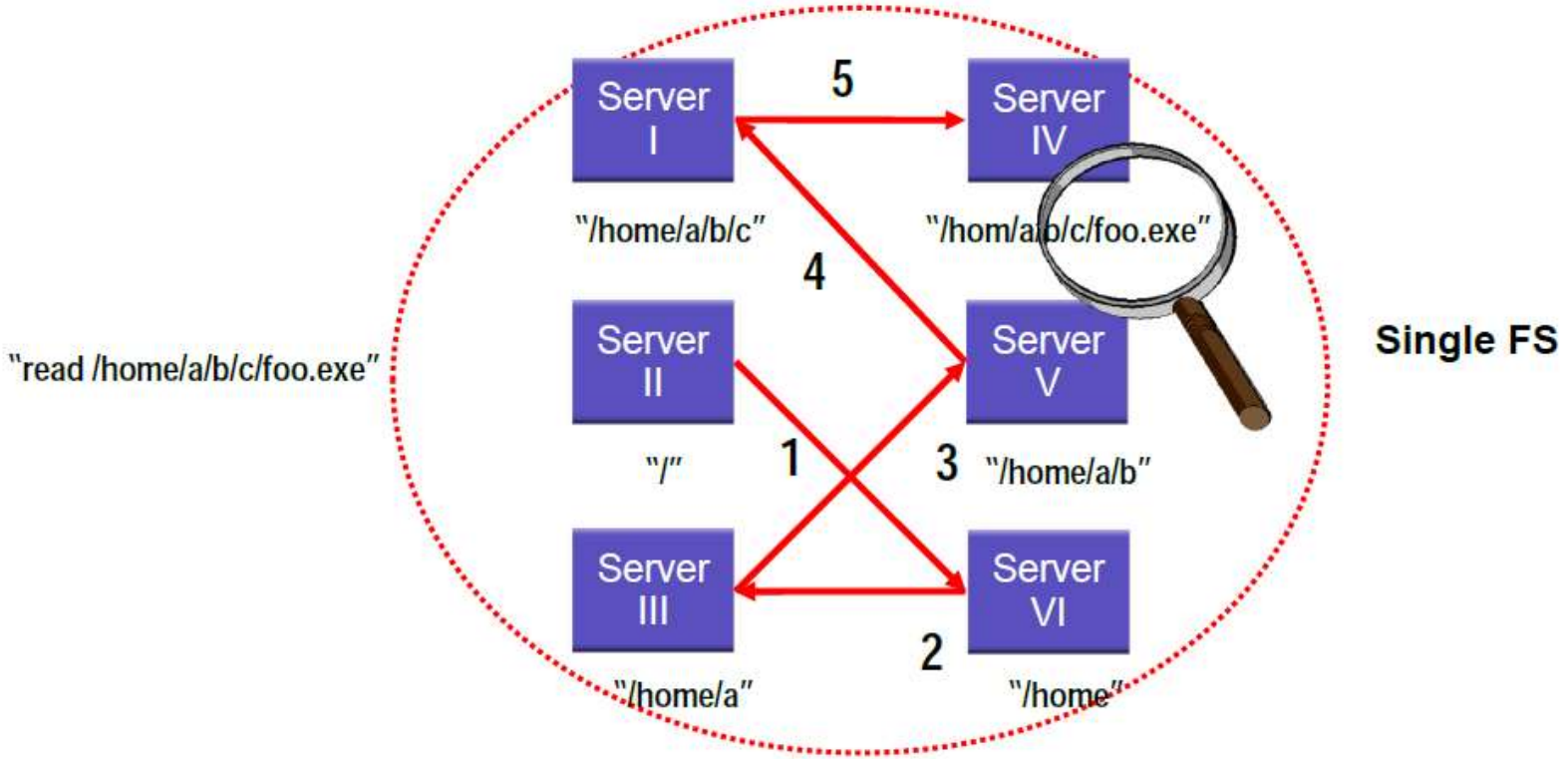


Key Characteristics of DFS

- Often purpose-built file servers
- No real standardization for file sharing across Unix (NFS) and Windows (CIFS)
- Scales independently of application services
- Performance limited to that of a single file server
- Reduces (not eliminate) islands of storage
- Replications sometimes built in
- Global namespace through external service
- Strong network security supported
- Etc.

DFS Logical Data Access Path

- **Using Ethernet as a networking protocol between nodes, a DFS allows a single file system to span across all nodes in the DFS cluster, effectively creating a unified Global Namespace for all files.**



2

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

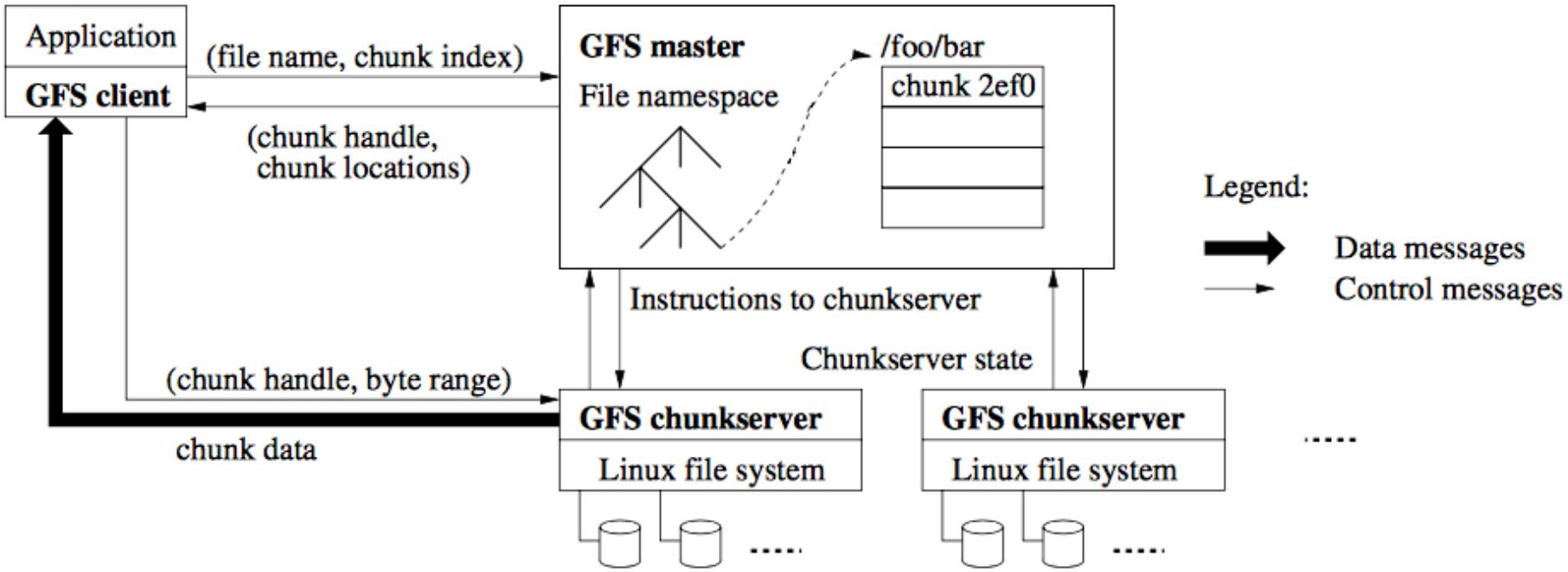
Typical workloads on GFS

- 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

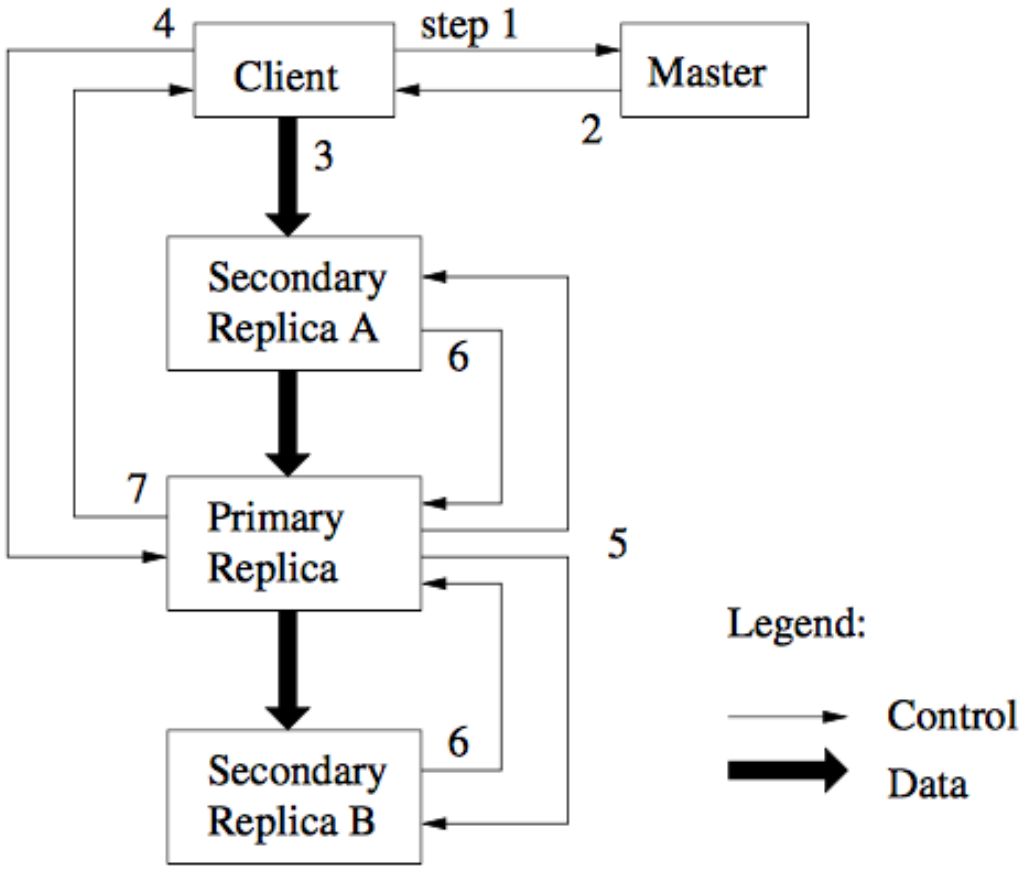
- A modification operation that guarantees that data (the “record”) will be appended atomically at least once – but at the offset of GFS’s choosing
  - The offset chosen by GFS is returned to the client so that the application is aware
- GFS may insert padding or record duplicates in between different 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

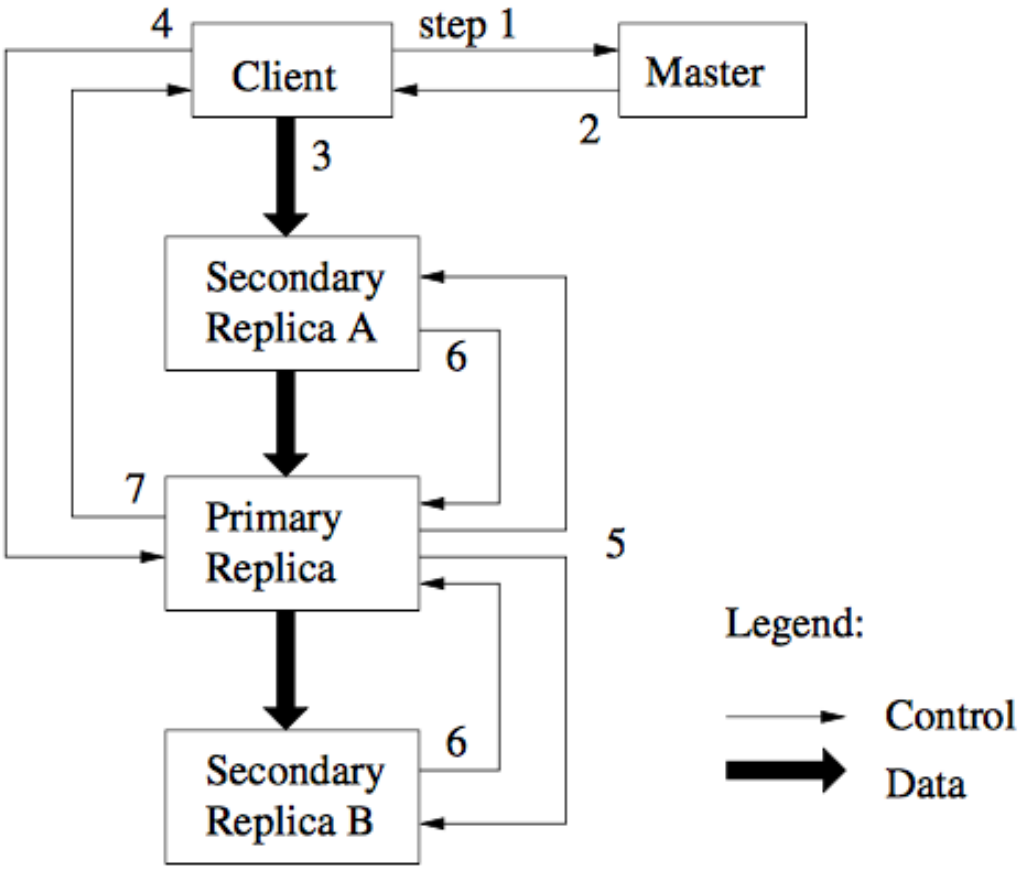
GFS Write Control and Data Flow (2)

- 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)

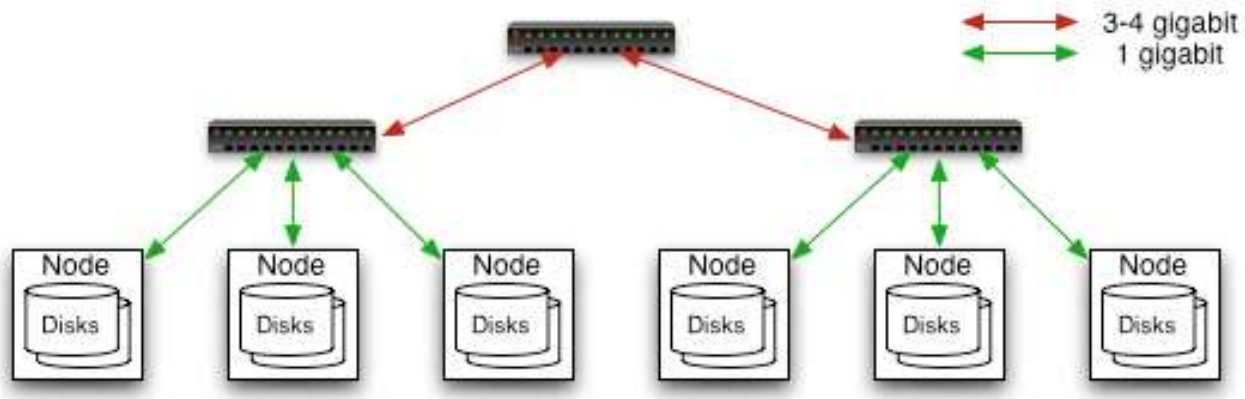


3 Hadoop File System (HDFS)

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

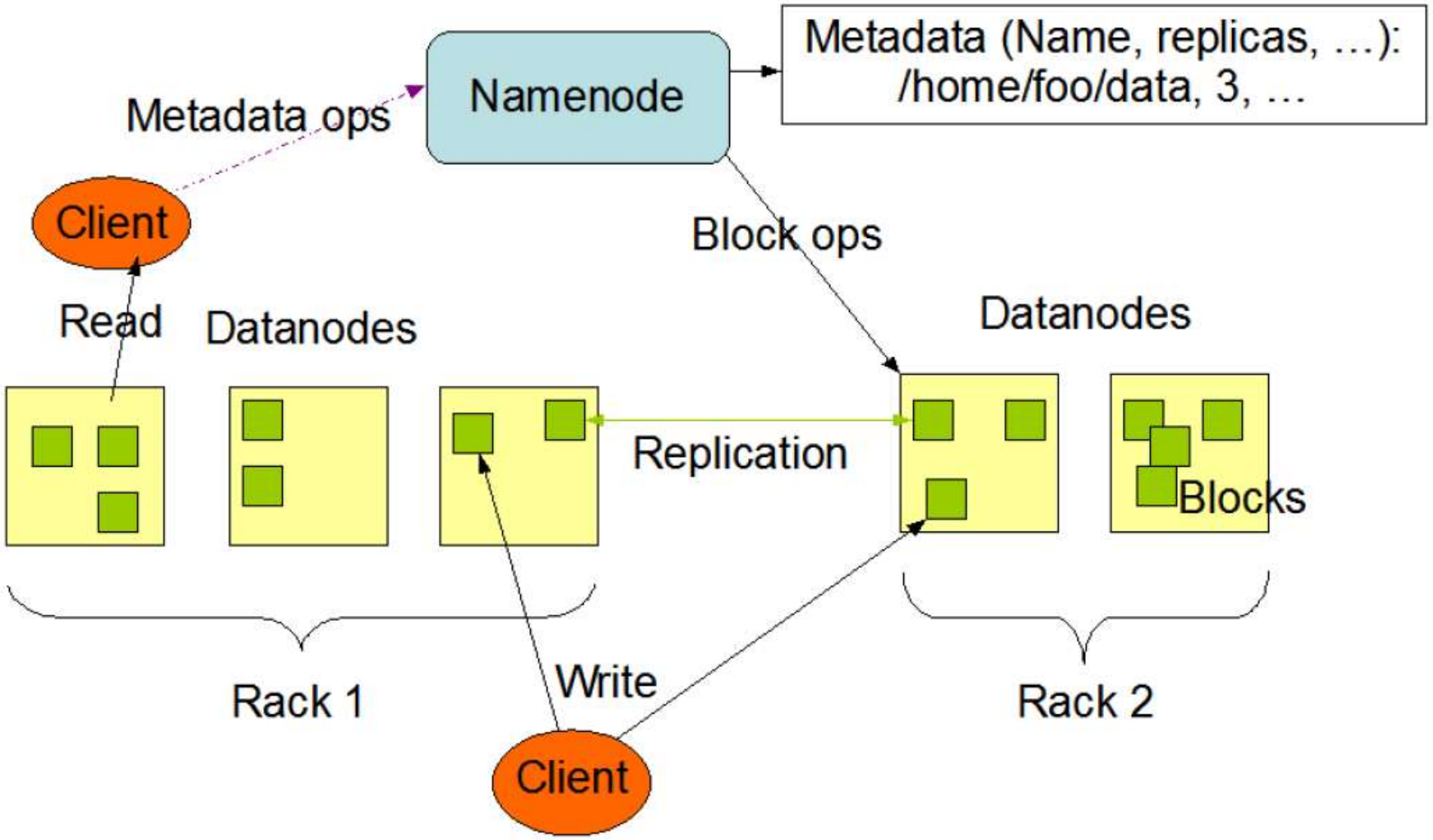
Goals of Hadoop Distributed File System (HDFS)

- **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**

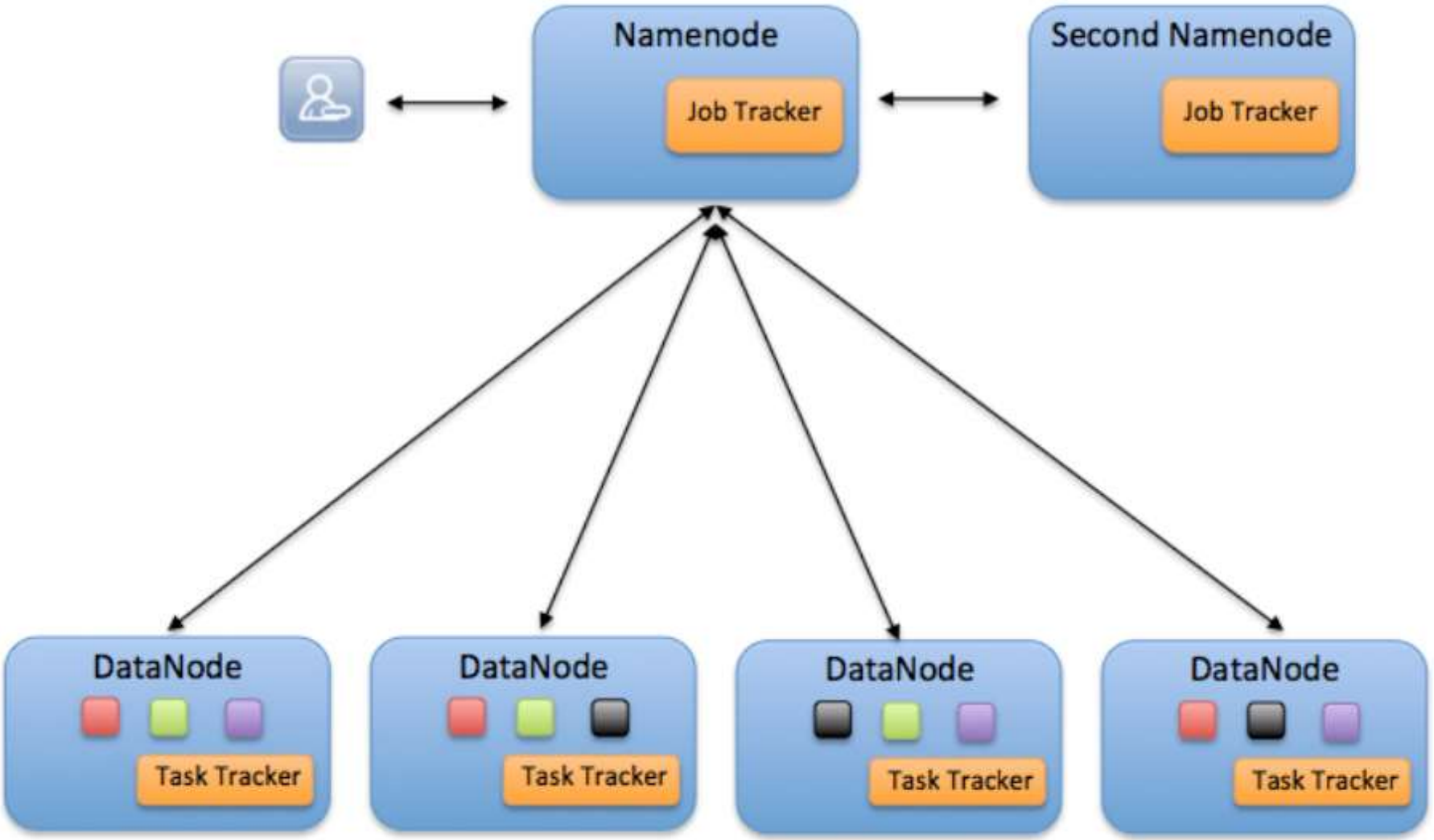
Basic of HDFS

- **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.

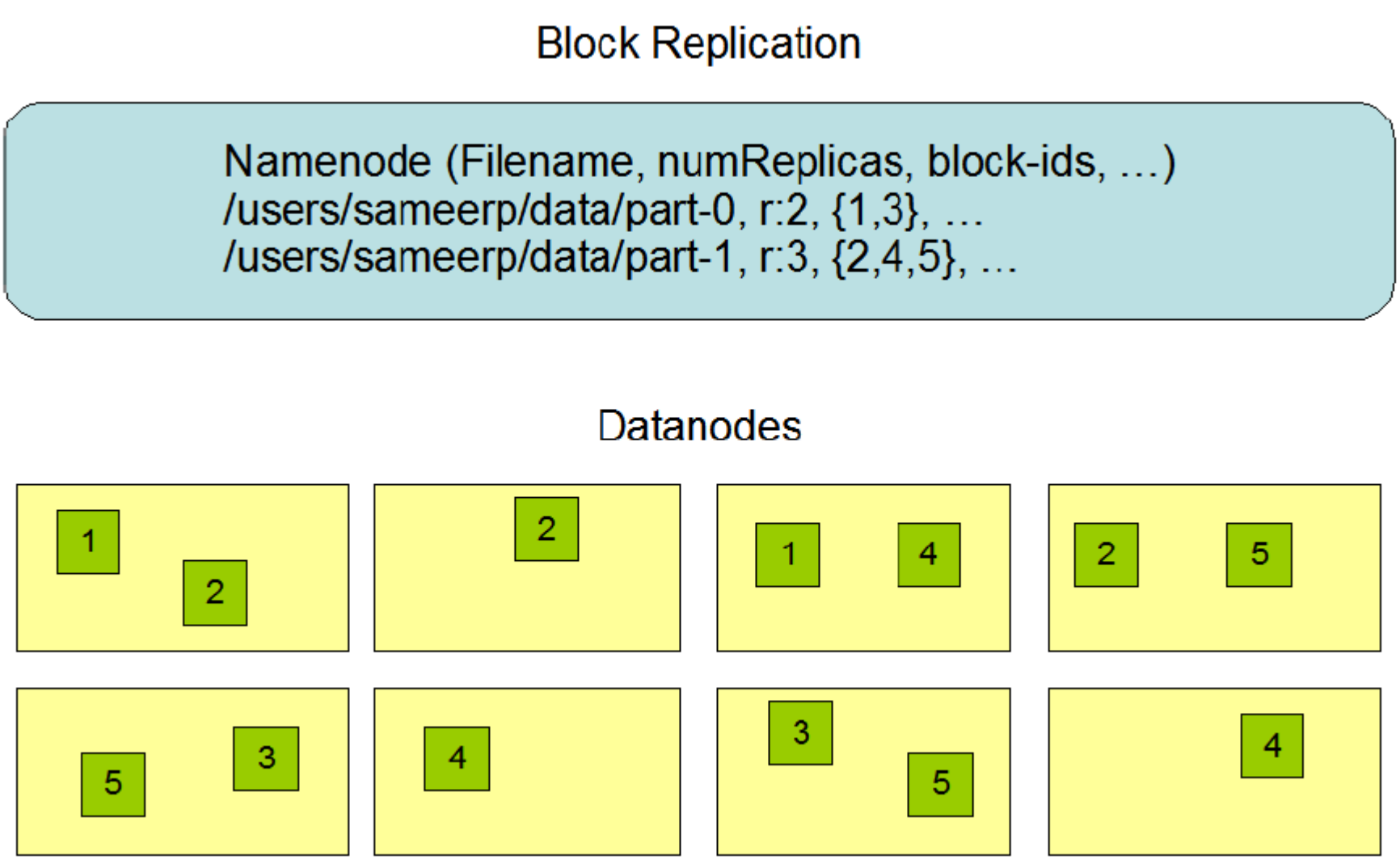
Datanode

- **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

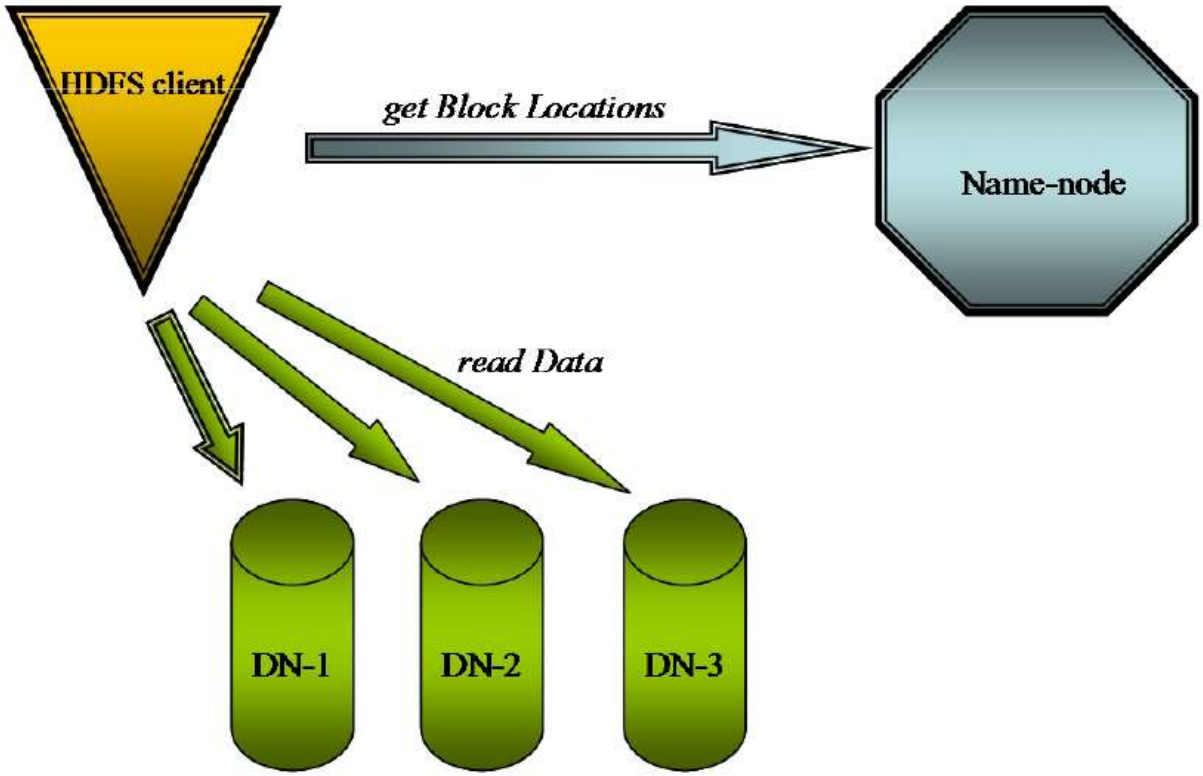
- **Current Strategy**
  - One replica on local node
  - Second replica on a remote rack
  - Third replica on same remote rack
  - Additional replicas are randomly placed
- **Clients read from nearest replica**
- **Pluggable policy for placing block replicas**
  - Co-locate datasets that are often used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

Block Replication



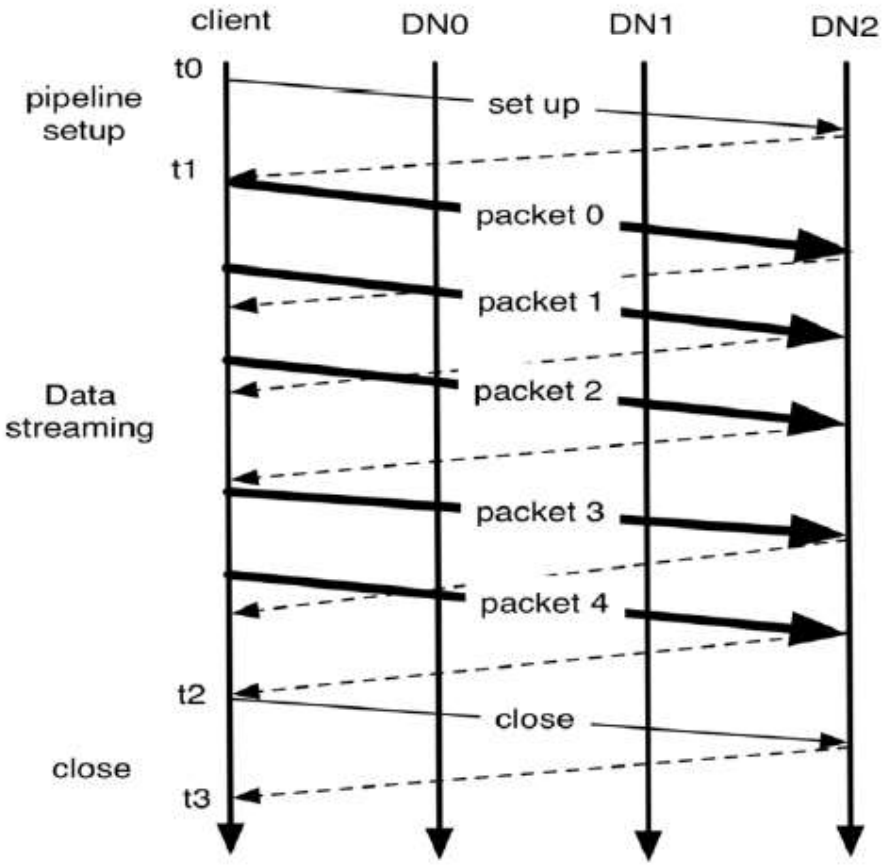
HDFS Read

- 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



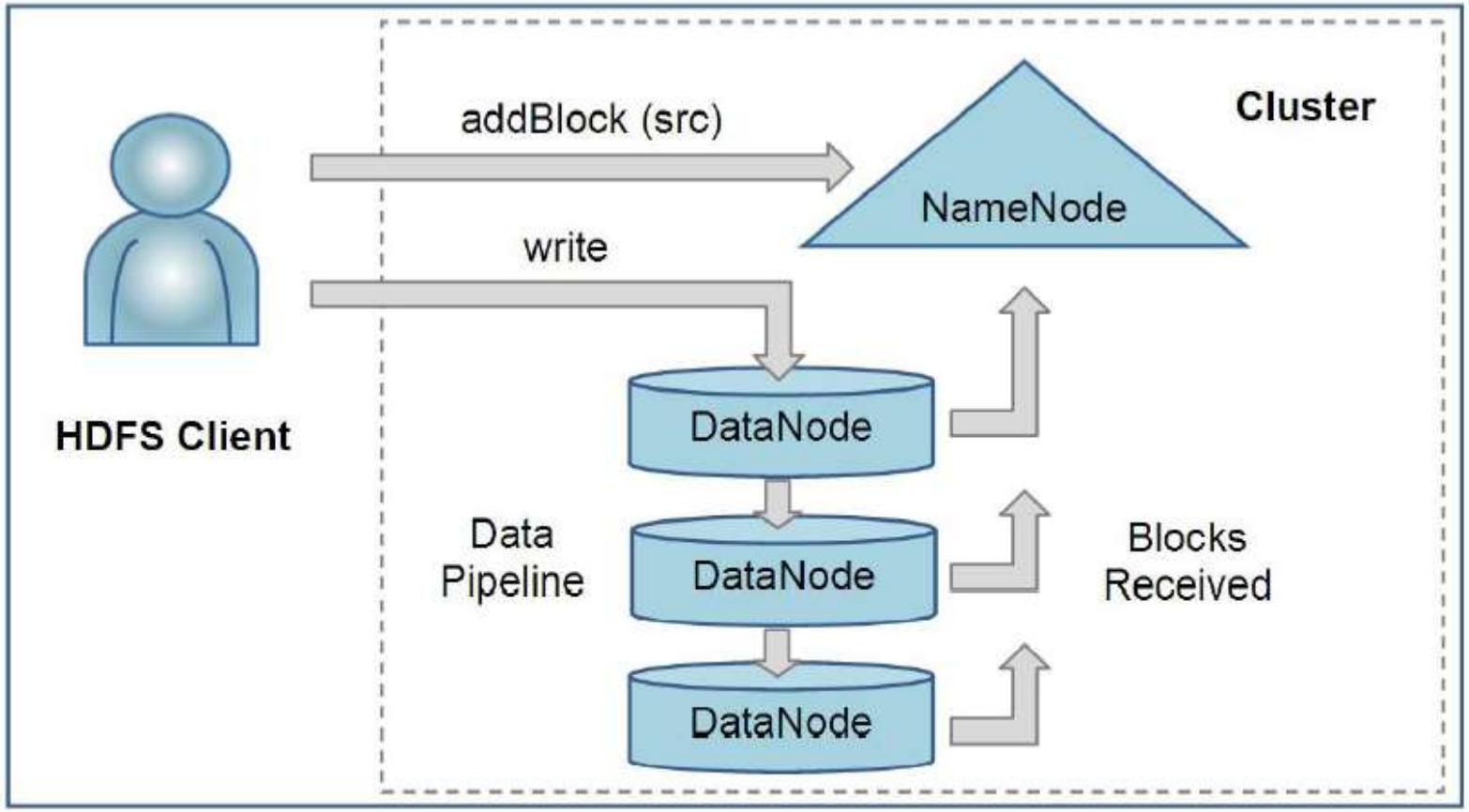
Data Pipelining

- 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.

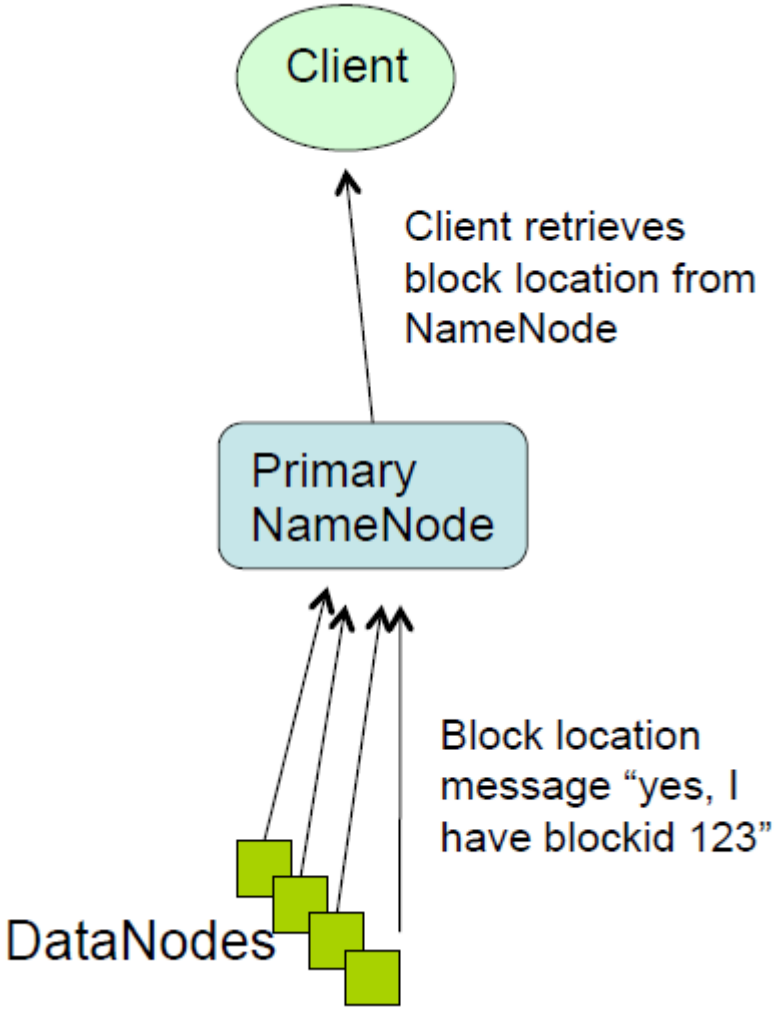


Namenode failure

- **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

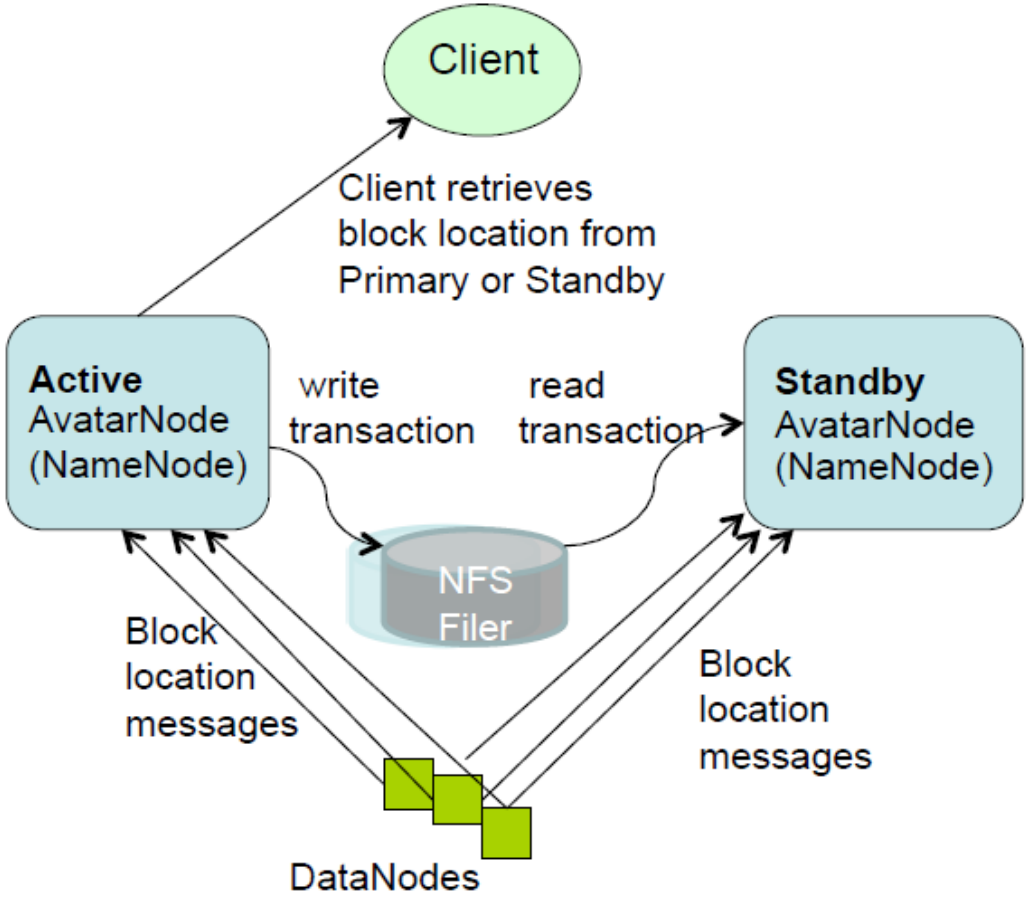
NameNode High Availability Challenges

- 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

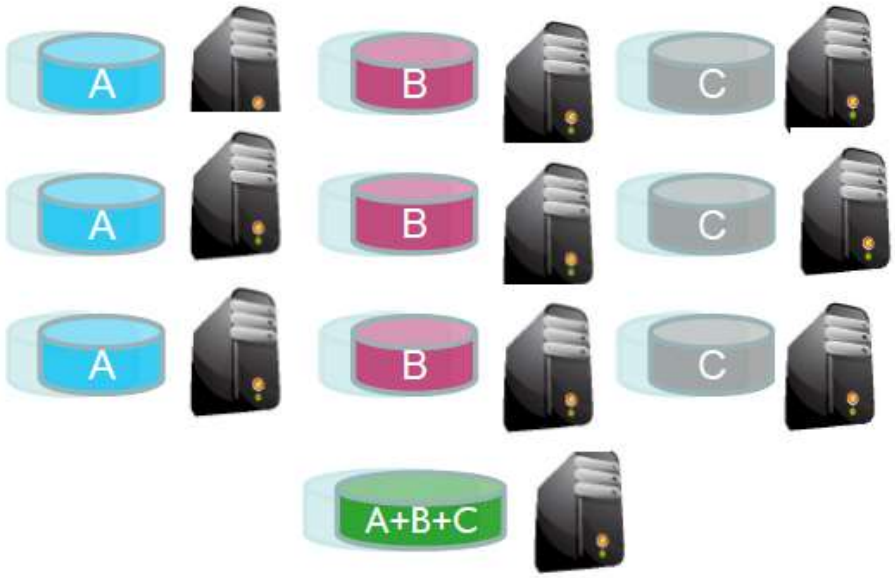


Rebalancer

- **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

HDFS Command

- HDFS Shell Command

Action	Command
Create a directory named /foodir	bin/hadoop dfs -mkdir /foodir
Remove a directory named /foodir	bin/hadoop dfs -rmr /foodir
View the contents of a file named /foodir/myfile.txt	bin/hadoop dfs -cat /foodir/myfile.txt

- HDFS Administrator Command

Action	Command
Put the cluster in Safemode	bin/hadoop dfsadmin -safemode enter
Generate a list of DataNodes	bin/hadoop dfsadmin -report
Recommission or decommission DataNode(s)	bin/hadoop dfsadmin -refreshNodes

4 Microsoft Azure and Ali DFS

Microsoft Azure Storage

Microsoft Azure

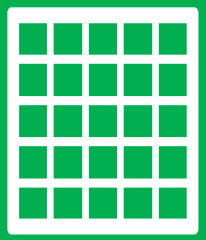
- **Blobs** – File system in the cloud
- **Tables** – Massively scalable structured storage
- **Queues** – Reliable storage and delivery of messages
- **Drives** – Durable NTFS volumes for Windows Azure applications

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.

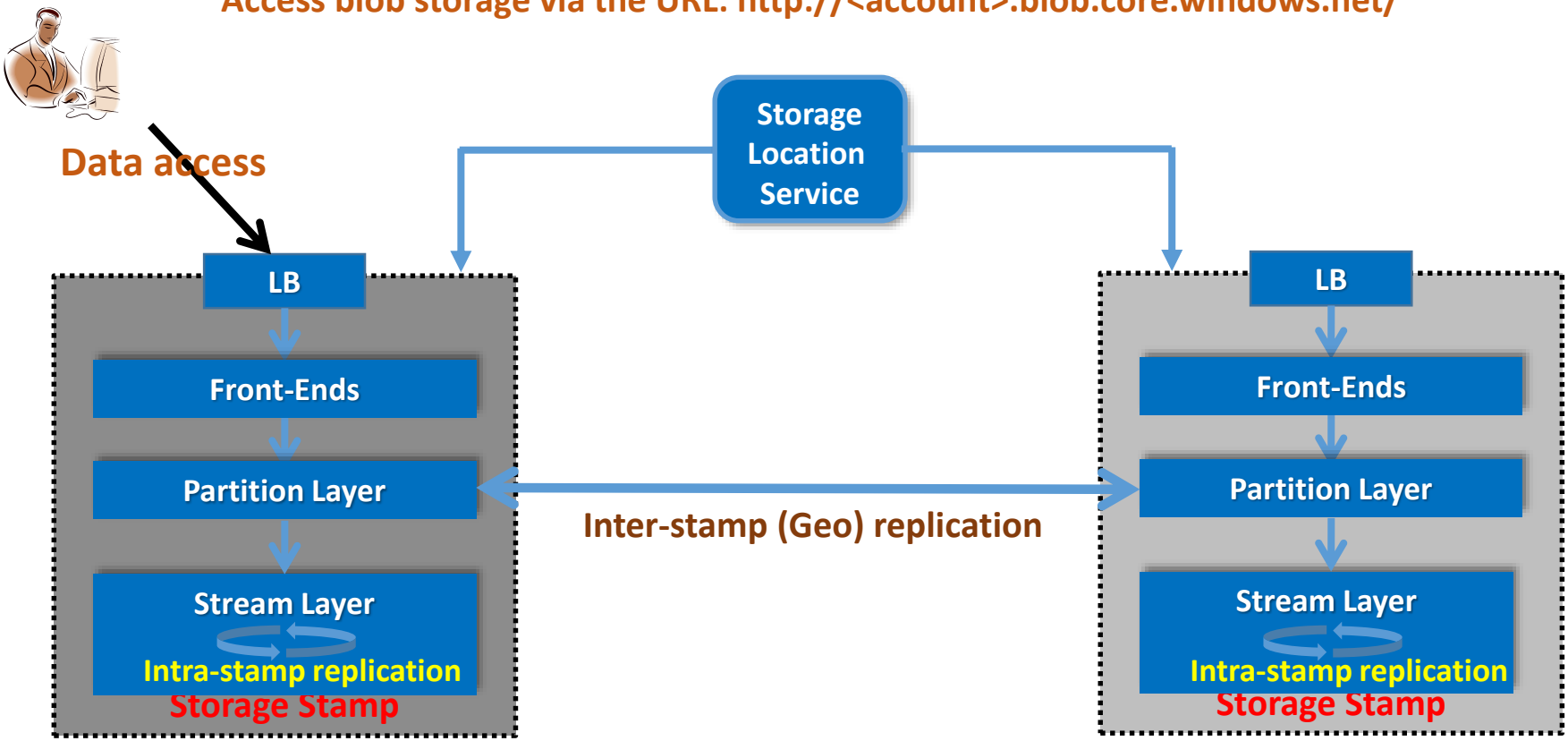
Queues

Reliable storage and delivery of messages for an application.

Windows Azure Storage Stamps

Microsoft Azure

Access blob storage via the URL: http://<account>.blob.core.windows.net/

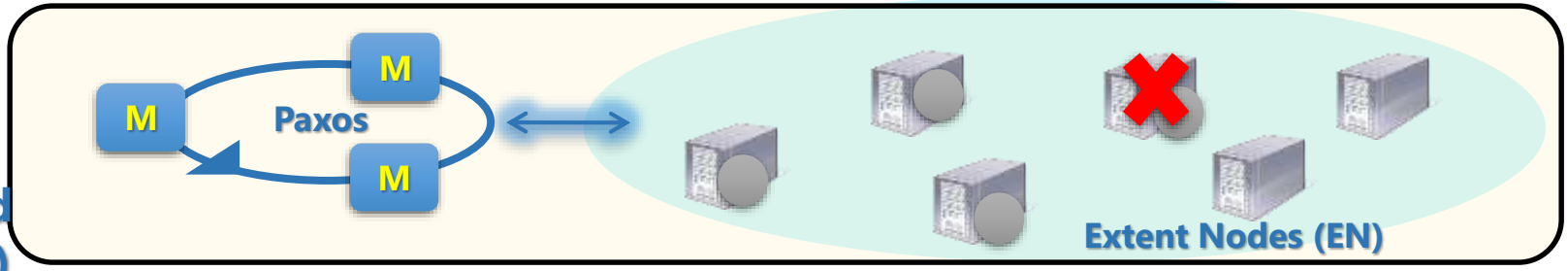


Storage Stamp Architecture – Stream Layer

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

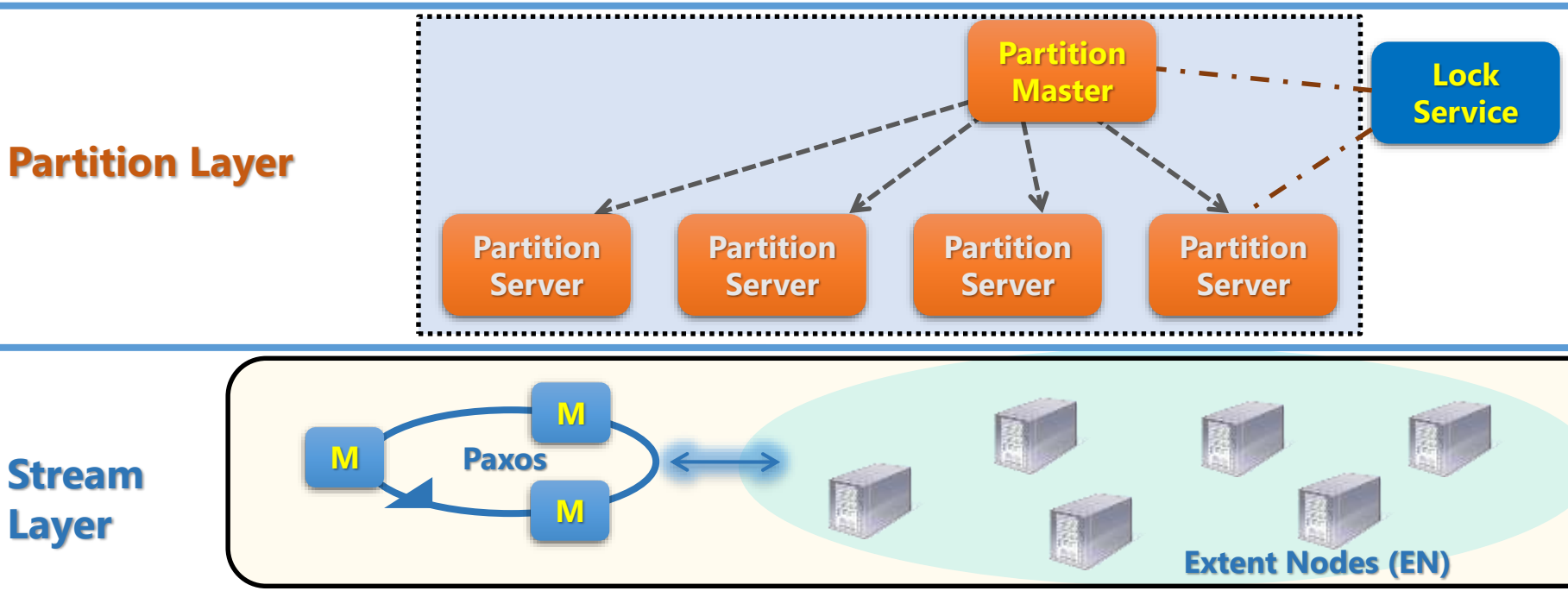
Stream Layer (Distributed File System)



Storage Stamp Architecture – Partition Layer

Microsoft Azure

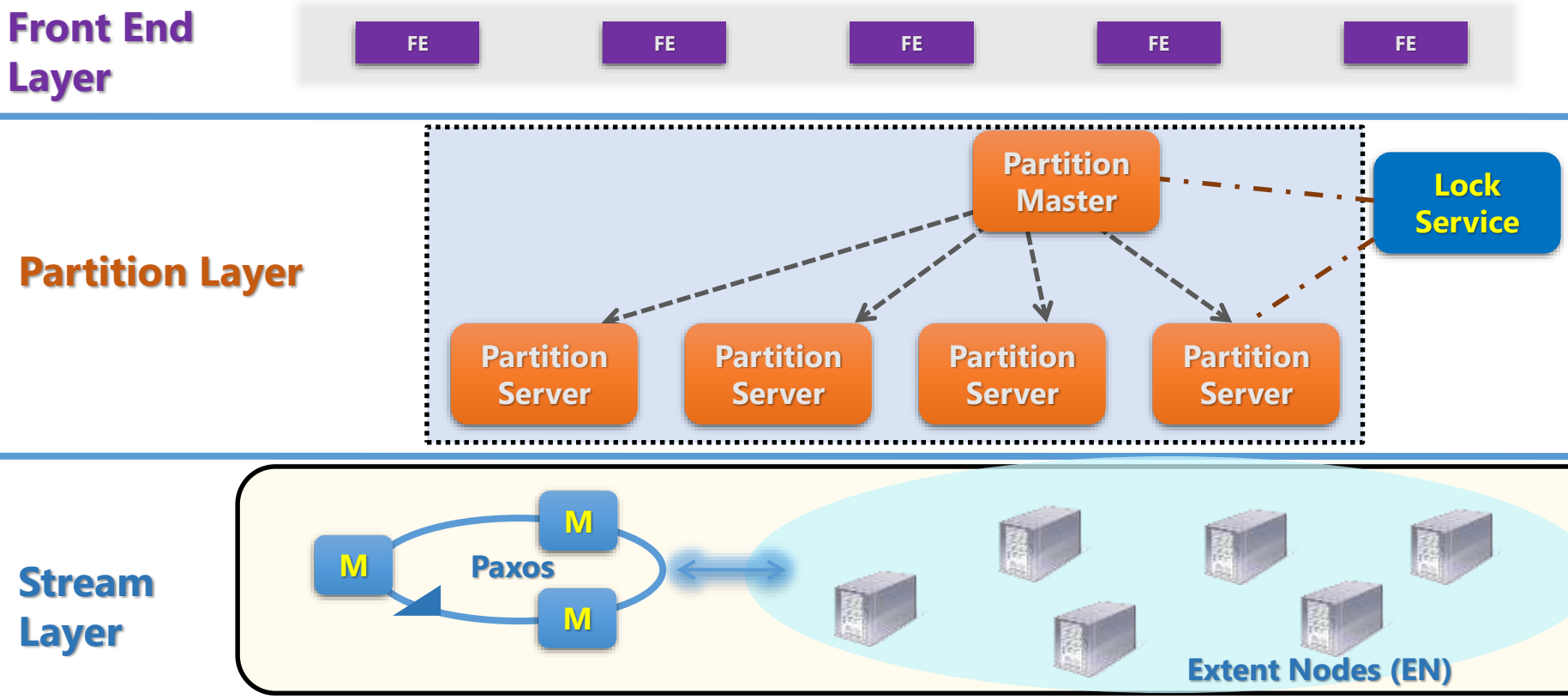
- Provide transaction semantics and strong consistency for Blobs, Tables and Queues
- Stores and reads the objects to/from extents in the Stream layer
- Provides inter-stamp (geo) replication by shipping logs to other stamps
- Scalable object index via partitioning



Storage Stamp Architecture – Front End Layer

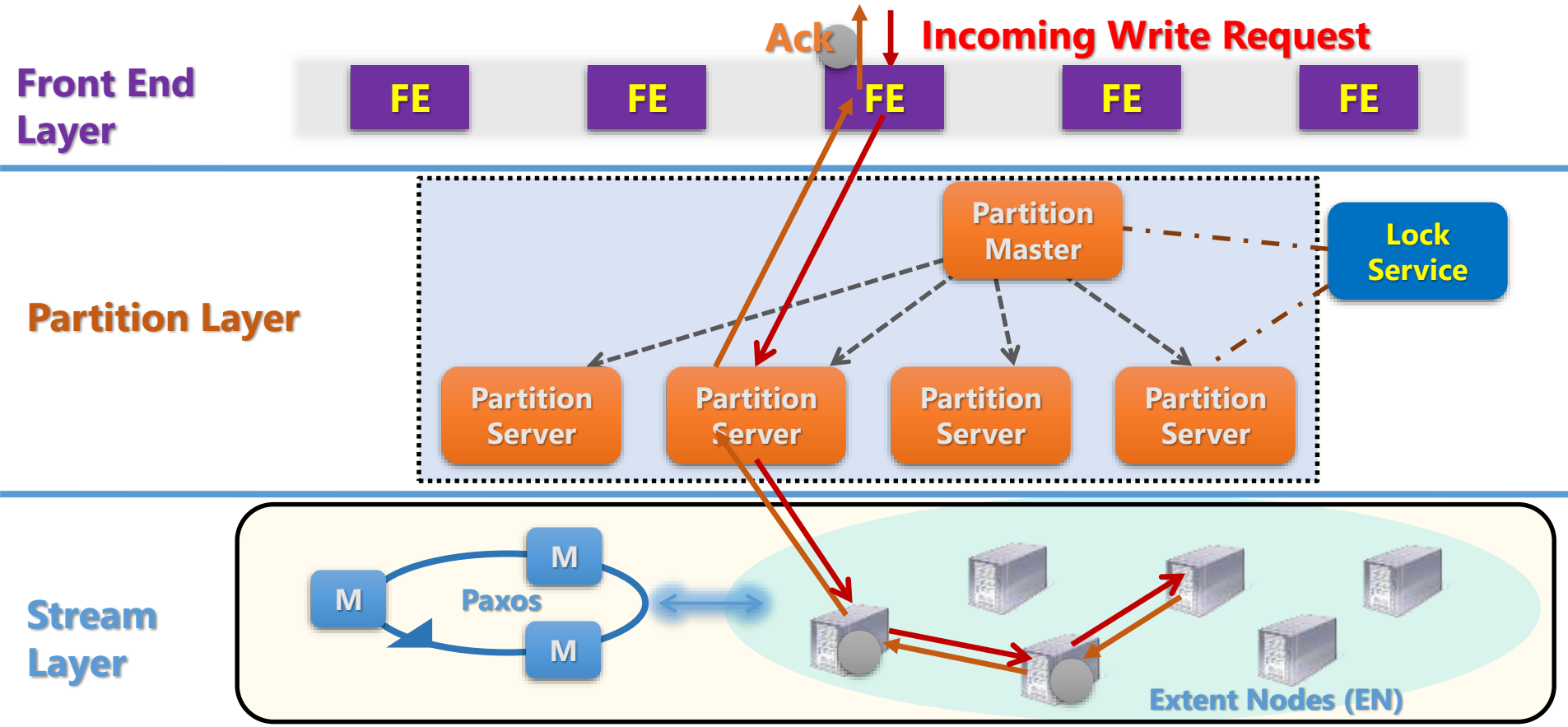
Microsoft Azure

- 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

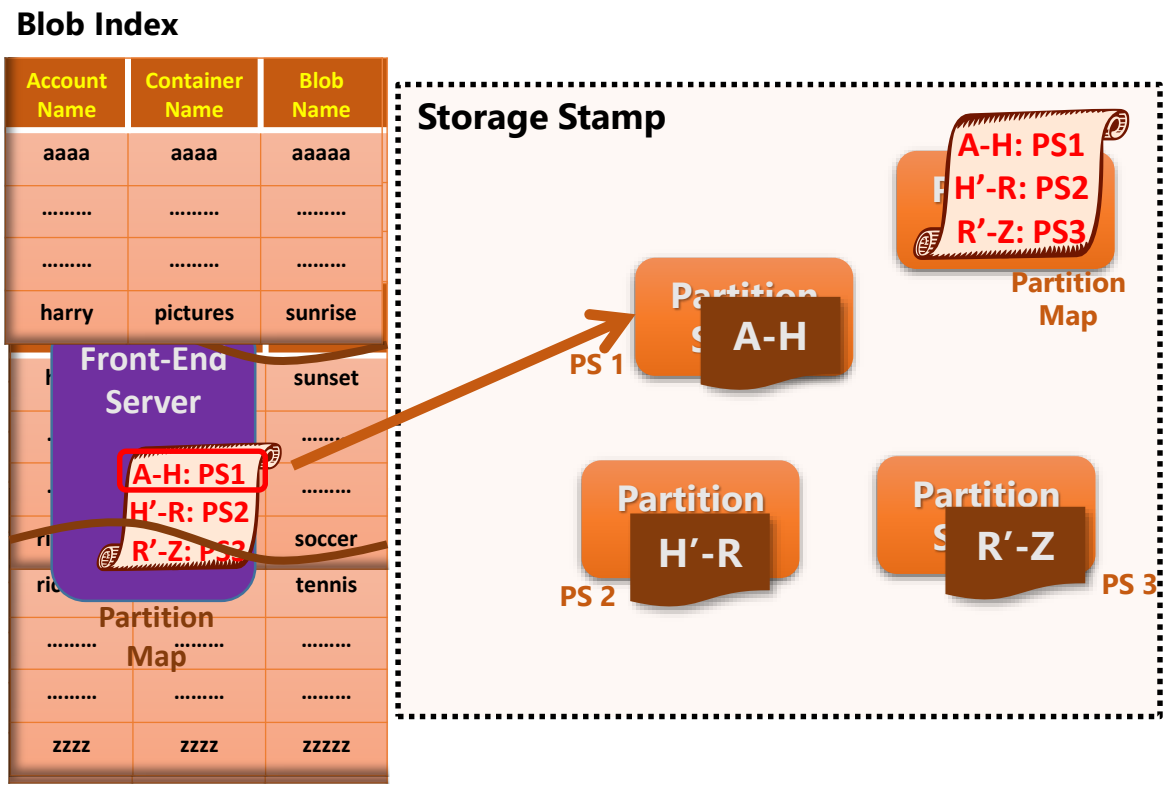
Microsoft Azure

- 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

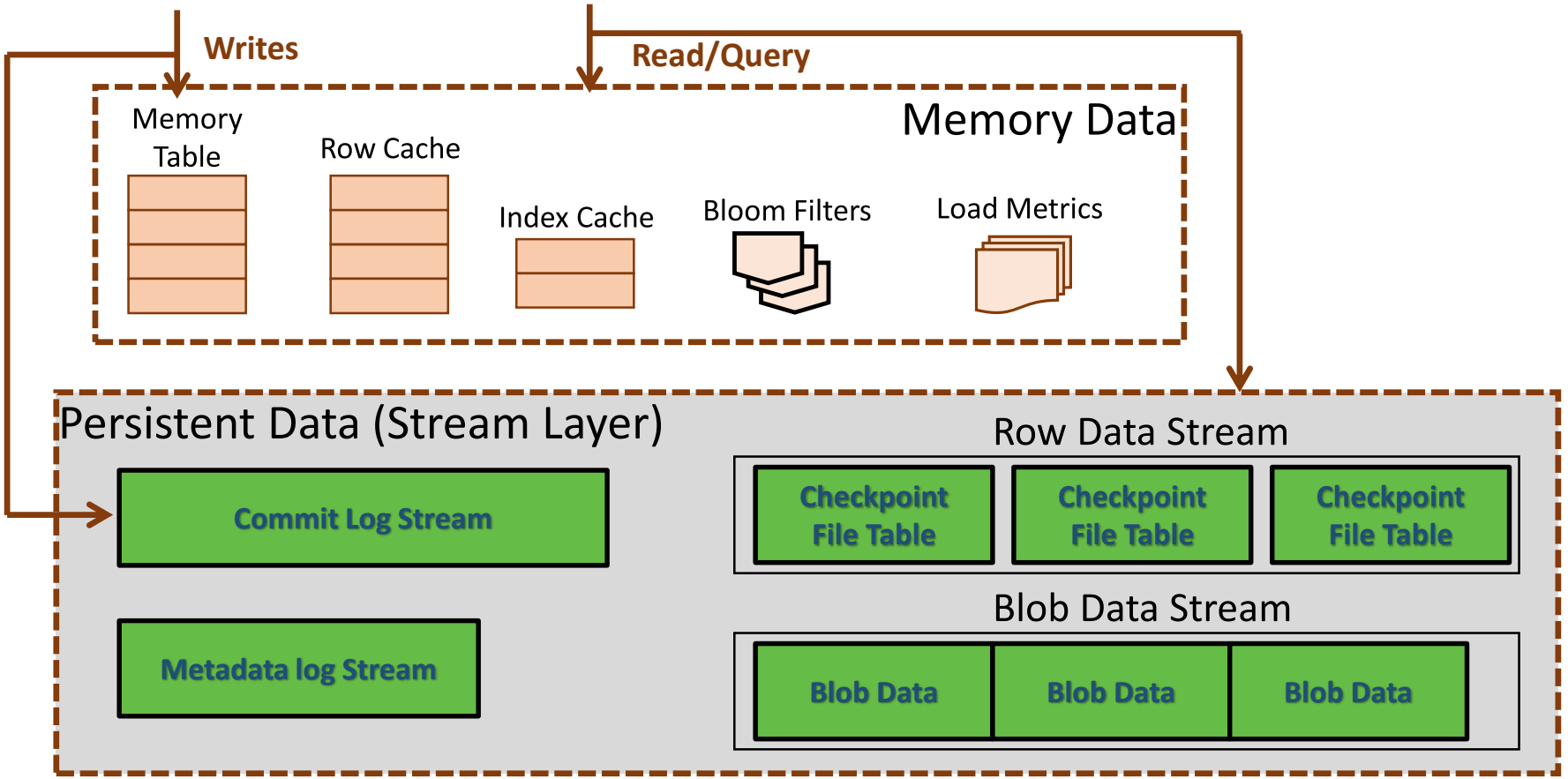
Microsoft Azure

- 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

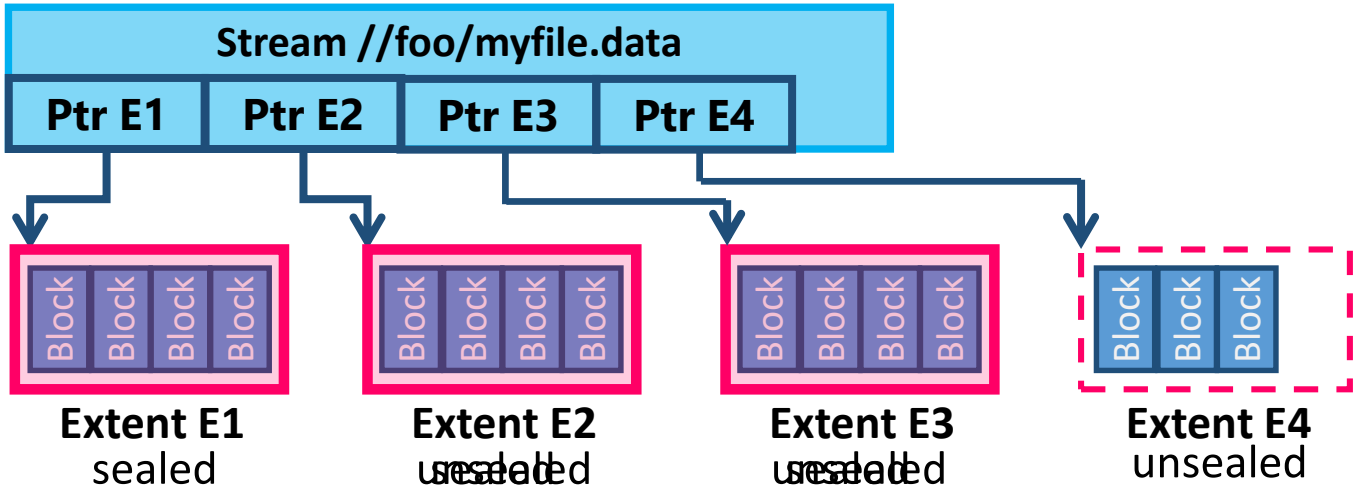
- Min unit of write/read
- Checksum
- Up to N bytes (e.g. 4MB)

Extent

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

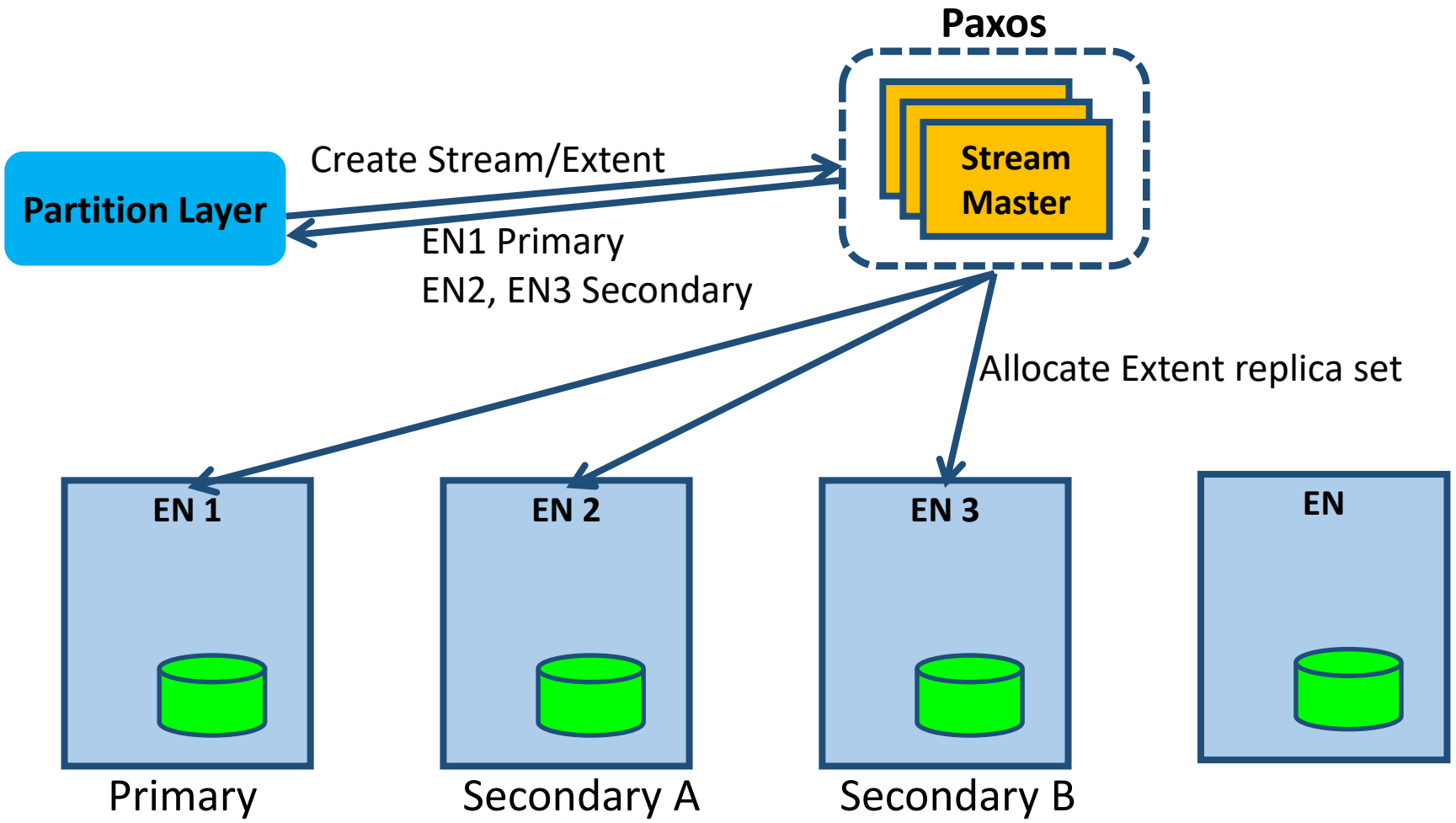
Stream

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



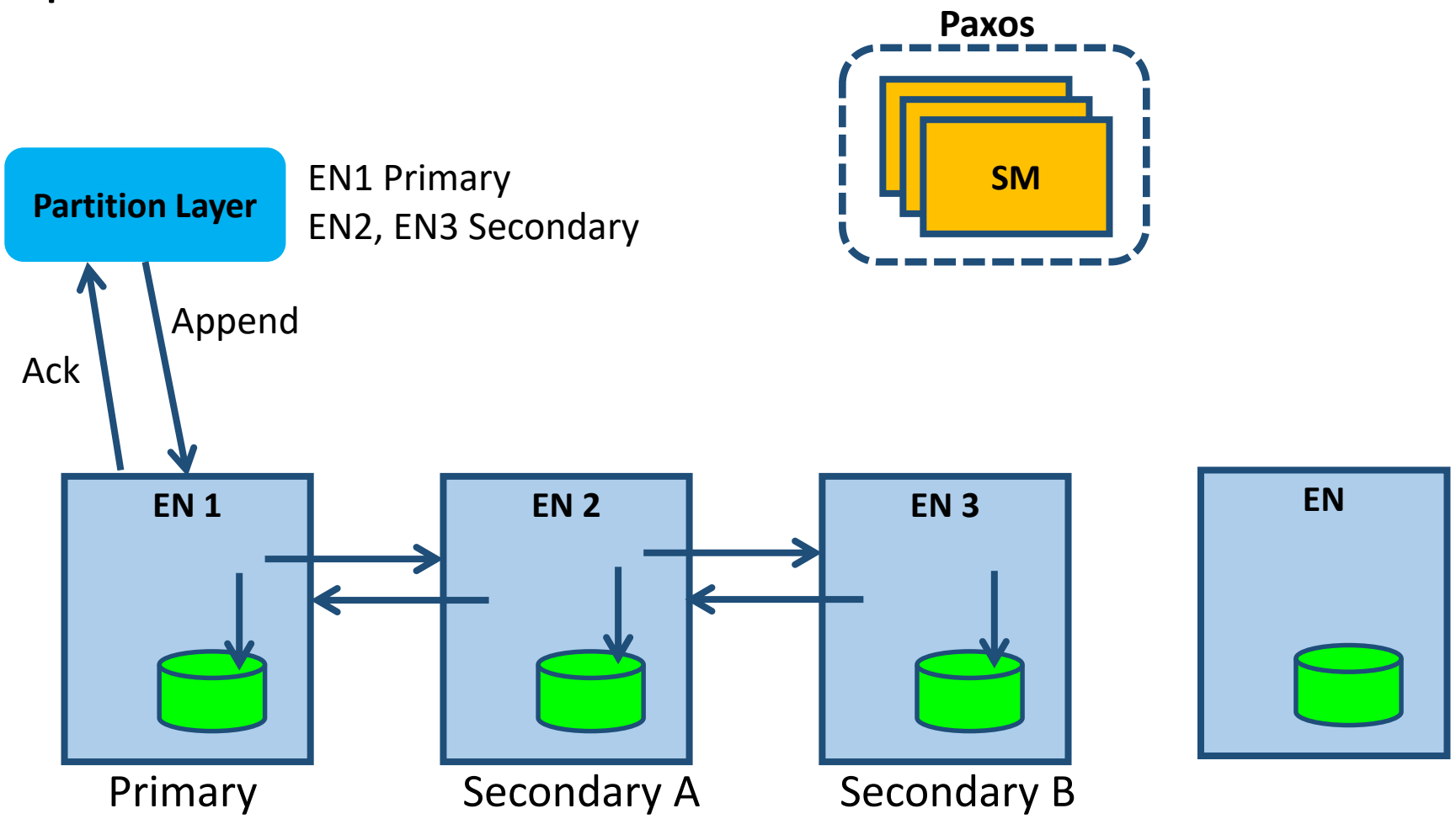
Creating an Extent

Microsoft Azure



Replication Flow

Microsoft Azure



Providing Bit-wise Identical Replicas

Microsoft Azure

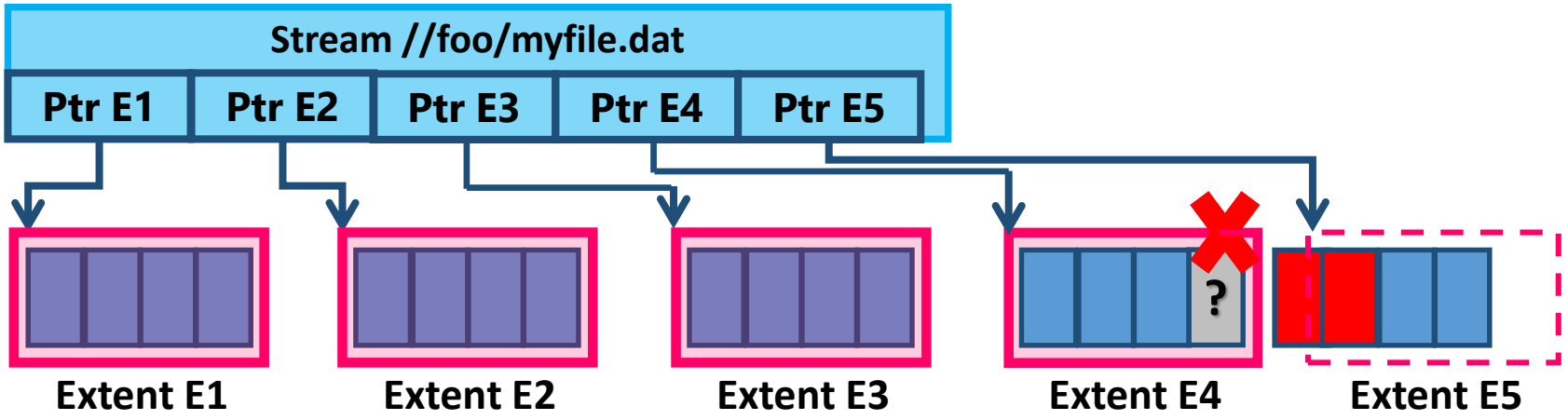
- Want all replicas for an extent to be bit-wise the same, up to a committed length
  - Want to store pointers from the partition layer index to an extent+offset
  - Want to be able to read from any replica
- Replication flow
  - All appends to an extent go to the Primary
  - Primary orders all incoming appends and picks the offset for the append in the extent
  - Primary then forwards offset and data to secondaries
  - Primary performs in-order acks back to clients for extent appends
    - Primary returns the offset of the append in the extent
    - An extent offset can commit back to the client once all replicas have written that offset and all prior offsets have also already been completely written
    - This represents the committed length of the extent

Dealing with Write Failures

Microsoft Azure

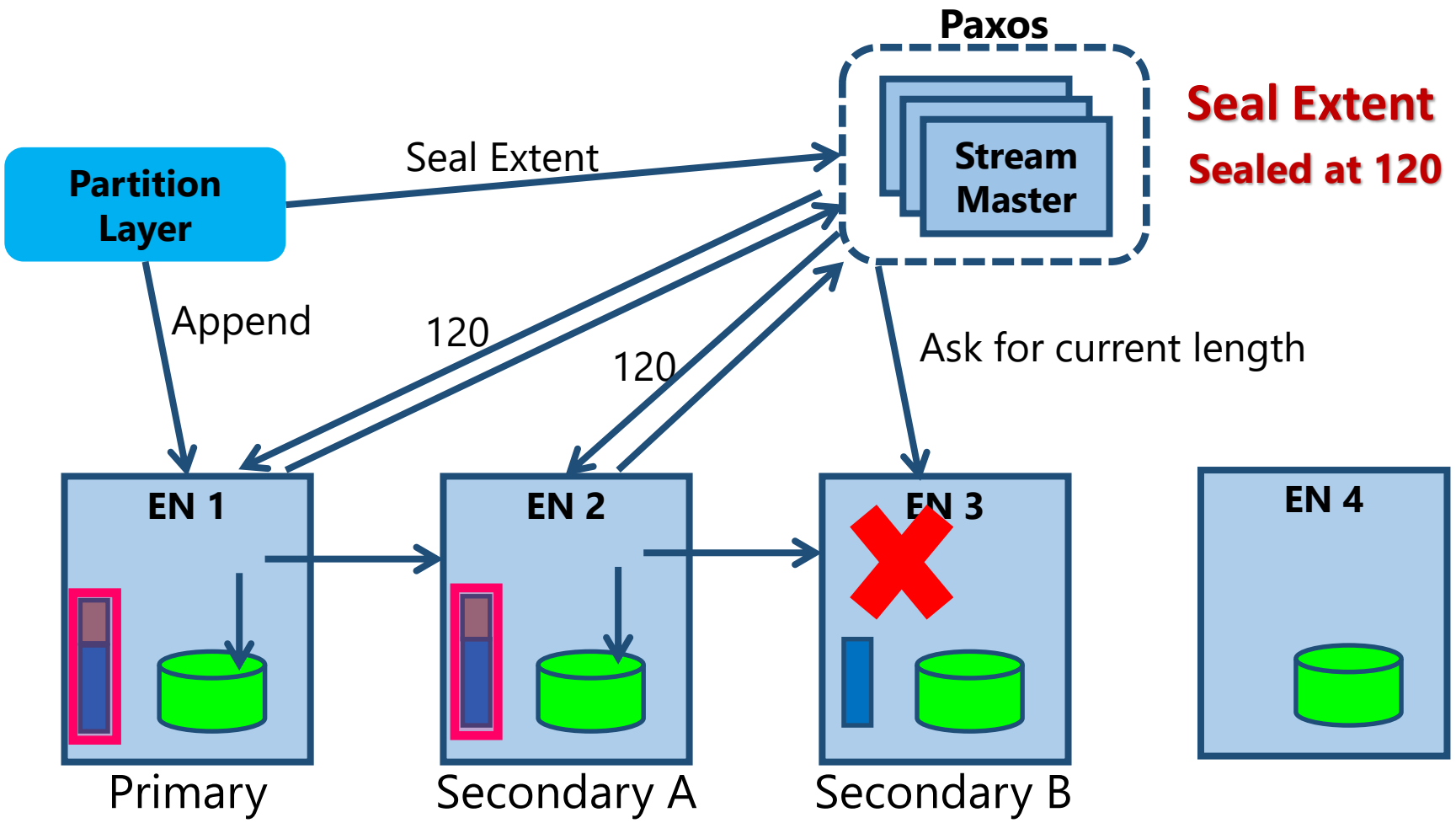
Failure during append

1. Ack from primary lost when going back to partition layer
   - Retry from partition layer can cause multiple blocks to be appended (duplicate records)
2. Unresponsive/Unreachable Extent Node (EN)
   - Append will not be acked back to partition layer
   - Seal the failed extent
   - Allocate a new extent and append immediately



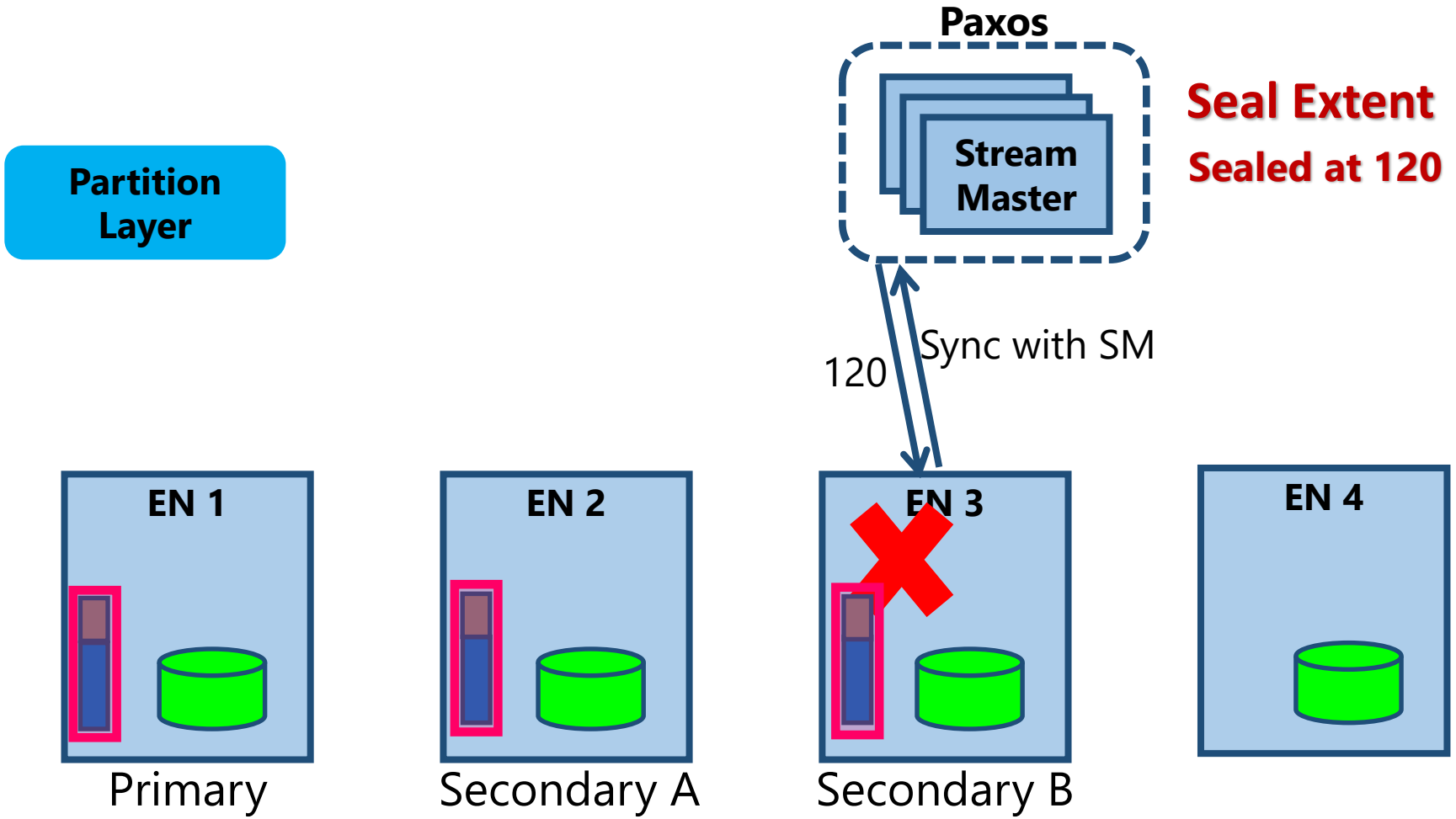
Extent Sealing (Scenario 1)

Microsoft Azure



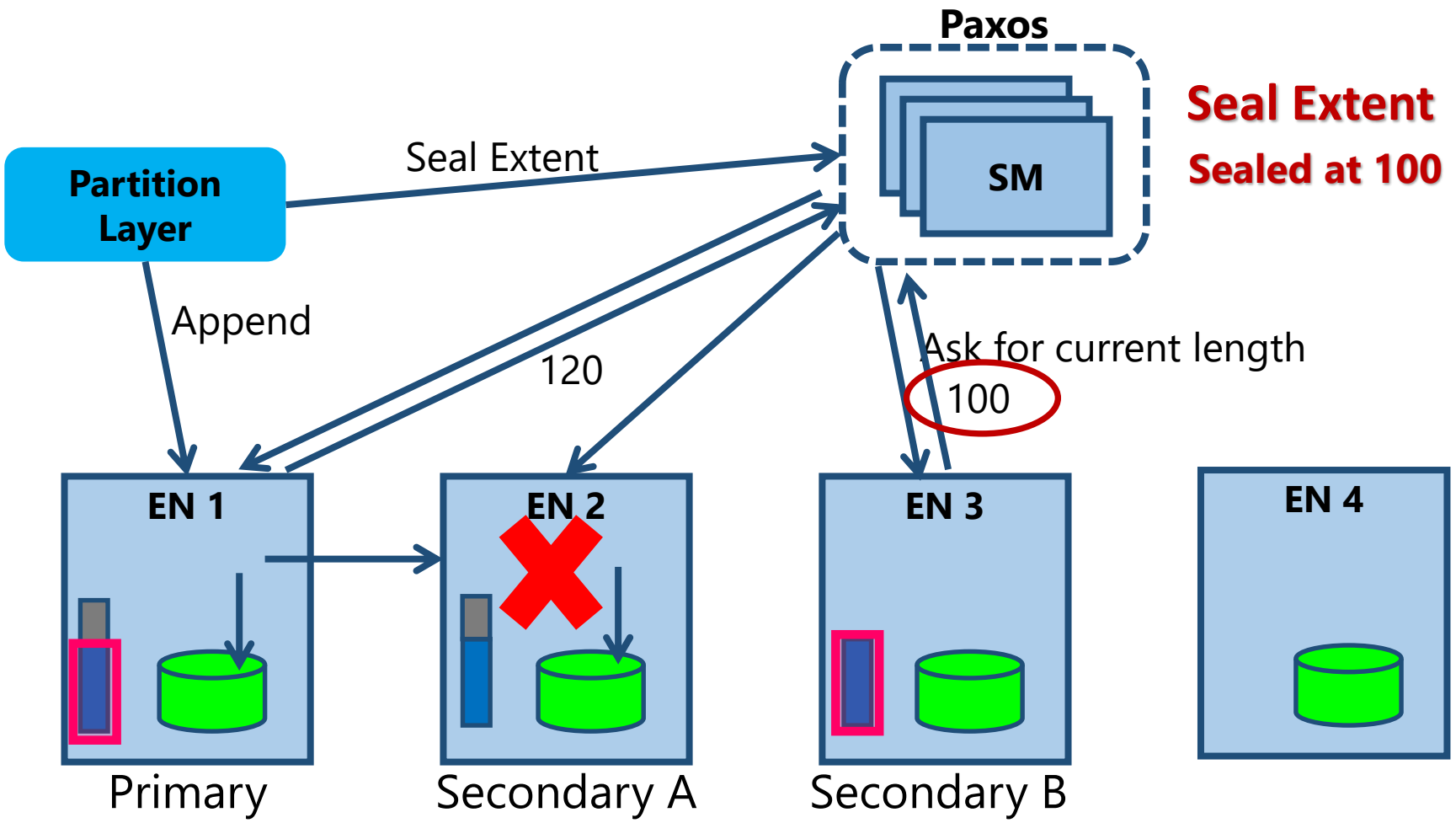
Extent Sealing (Scenario 1)

Microsoft Azure



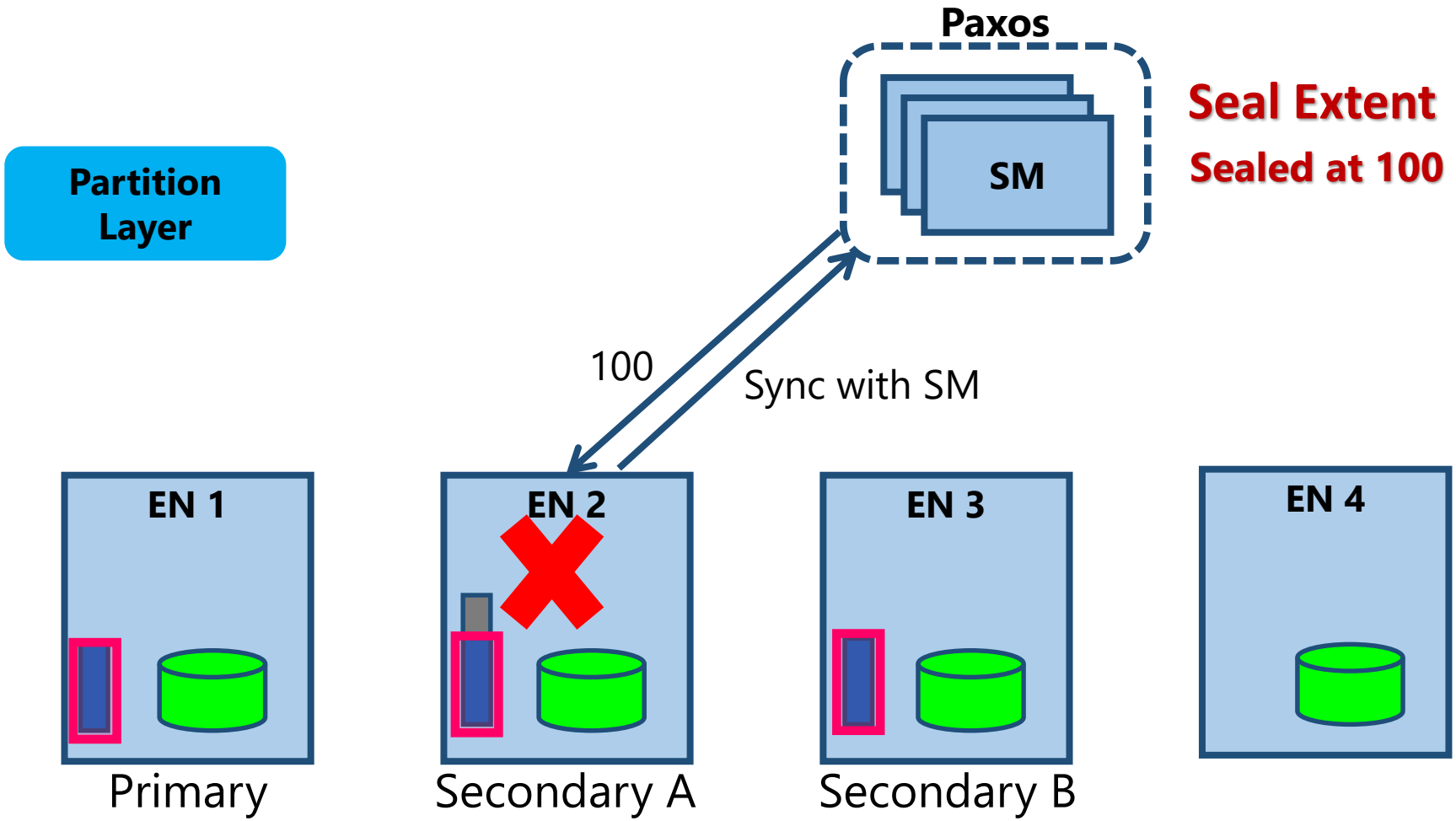
Extent Sealing (Scenario 2)

Microsoft Azure



Extent Sealing (Scenario 2)

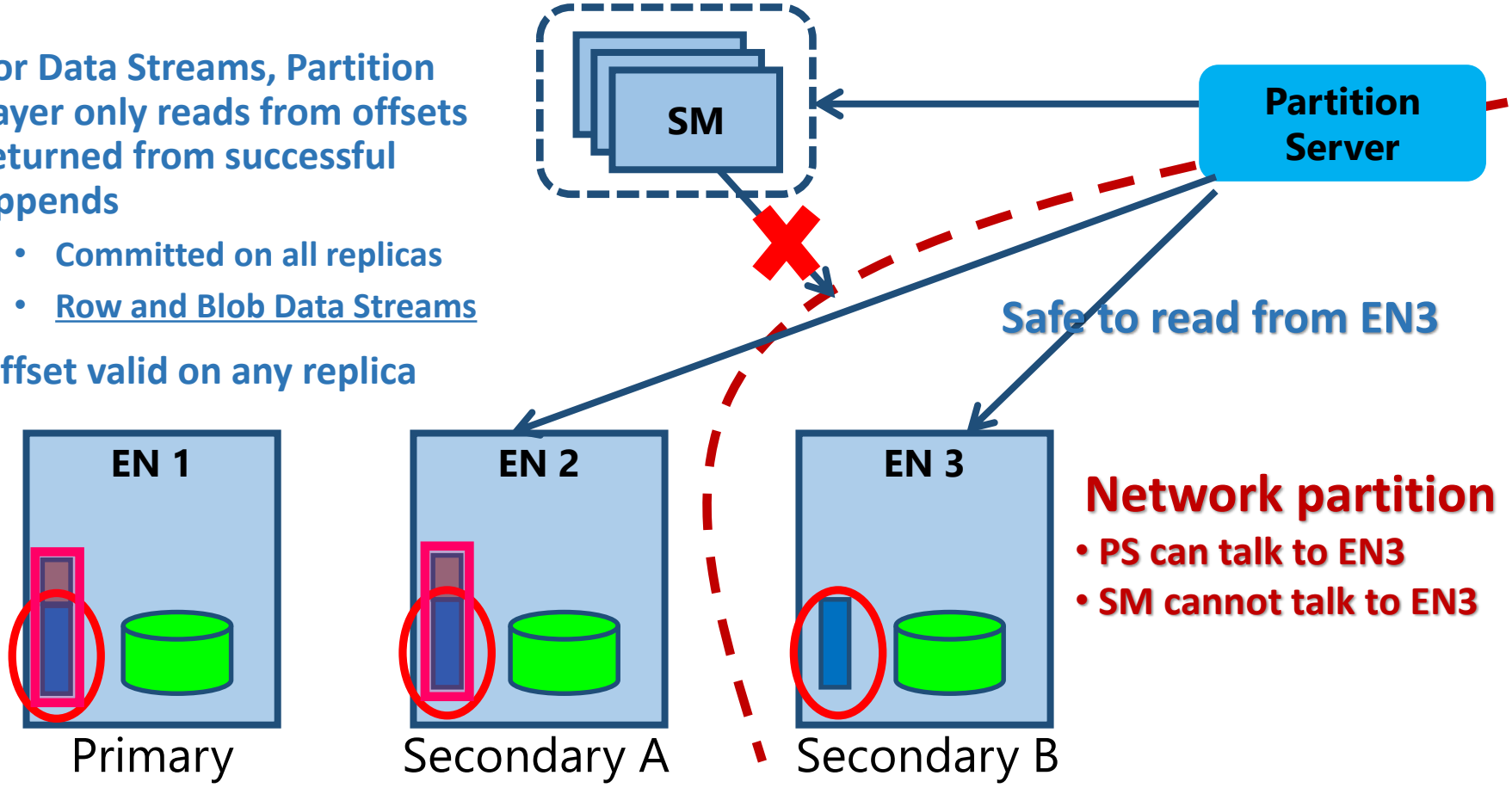
Microsoft Azure



Providing Consistency for **Data Streams**

Microsoft Azure

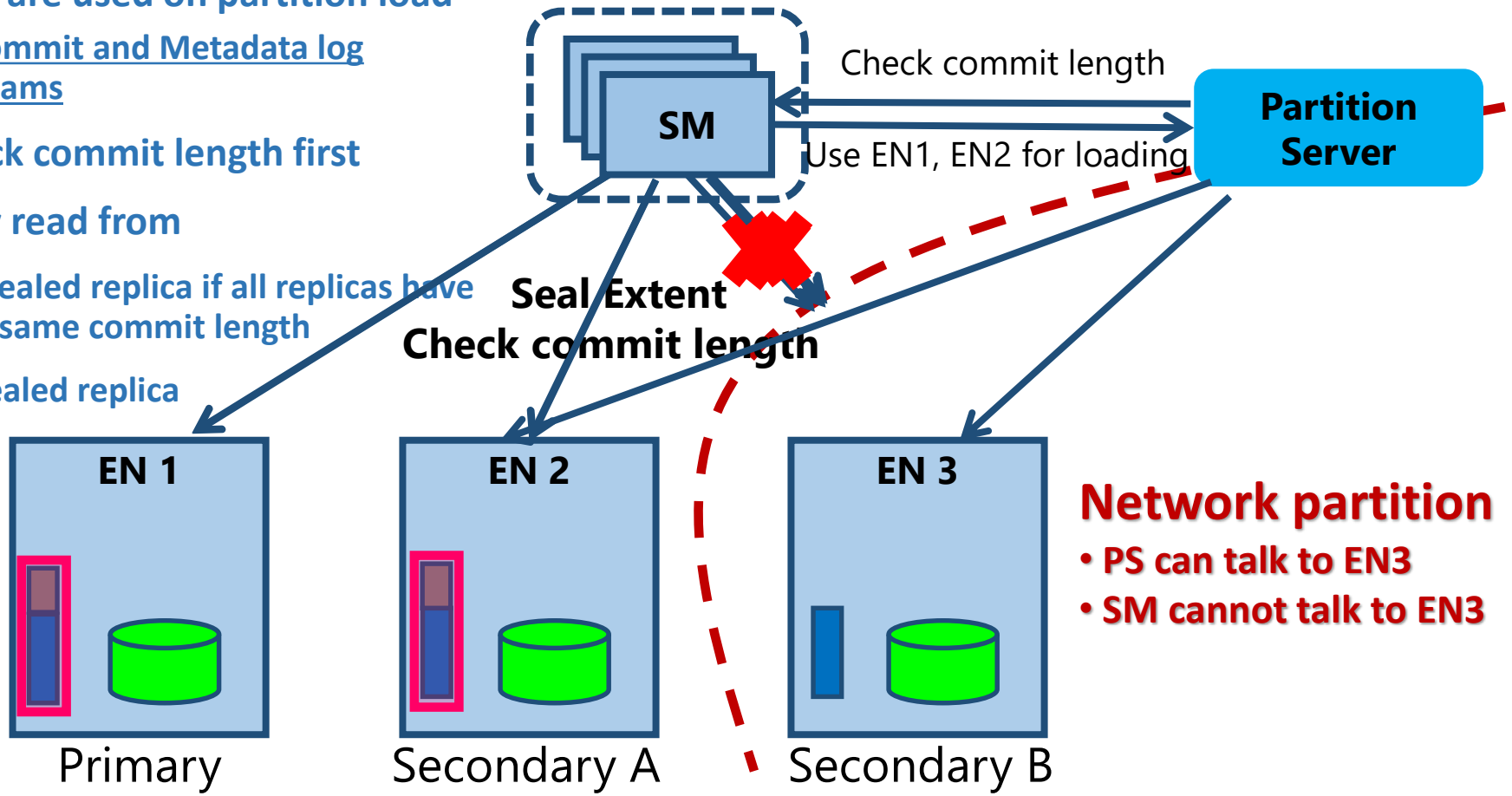
- For Data Streams, Partition Layer only reads from offsets returned from successful appends
  - Committed on all replicas
  - Row and Blob Data Streams
- Offset valid on any replica



Providing Consistency for **Log Streams**

Microsoft Azure

- **Logs are used on partition load**
  - **Commit and Metadata log streams**
- **Check commit length first**
- **Only read from**
- **Unsealed replica if all replicas have the same commit length**
- **A sealed replica**



Network partition

- **PS can talk to EN3**
- **SM cannot talk to EN3**

Design Consideration (1)

Microsoft Azure

- **Multi-Data Architecture**
  - Use extra resources to serve mixed workload for incremental costs
    - Blob -> storage capacity
    - Table -> IOPS
    - Queue -> memory
    - Drives -> storage capacity and IOPS
  - Multiple data abstractions from a single stack
    - Improvements at lower layers help all data abstractions
    - Simplifies hardware management
  - Tradeoff: single stack is not optimized for specific workload pattern

Design Consideration (2)

Microsoft Azure

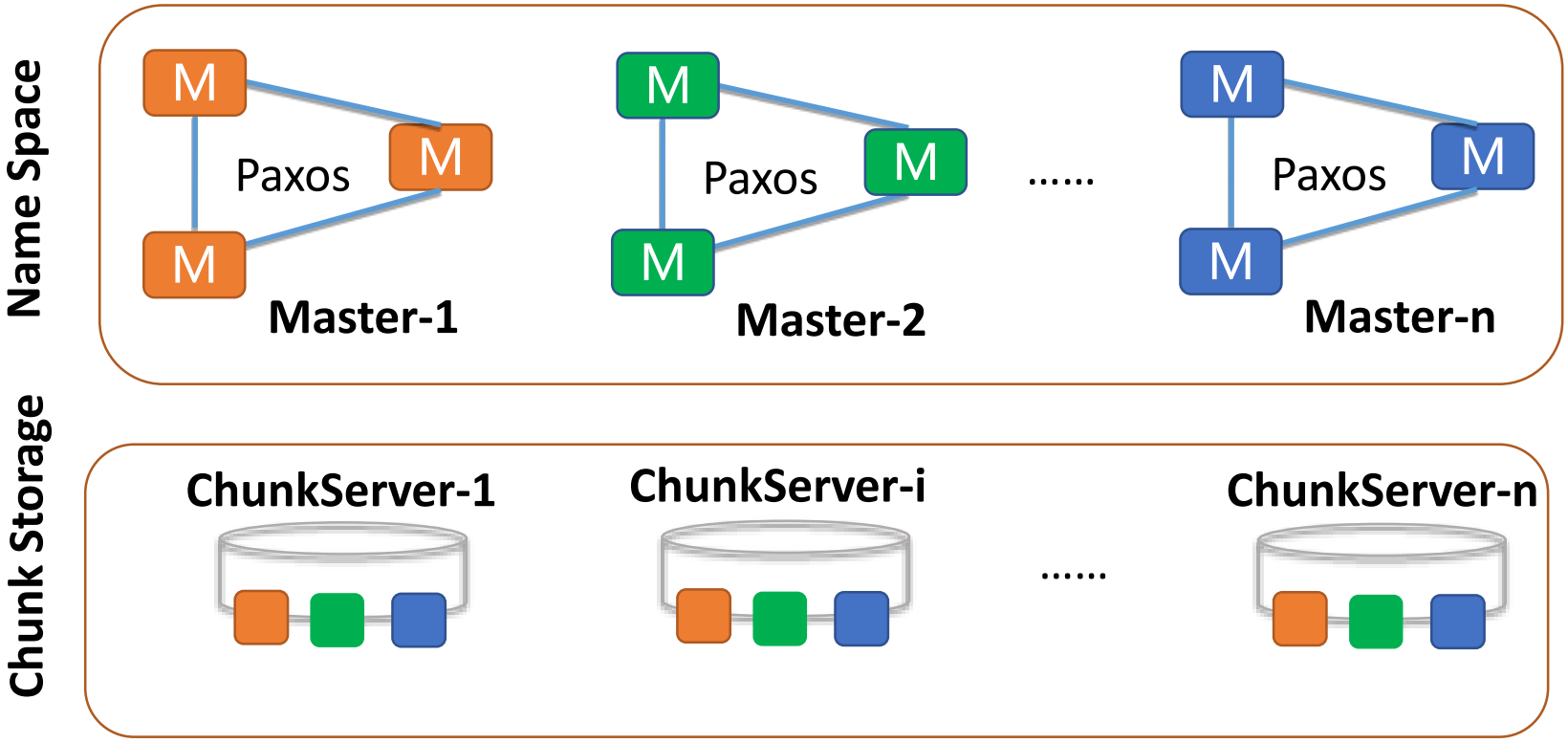
- **Append-only System**
  - Greatly simplifies replication protocol and failure handling
    - Consistent and identical replicas up to the extent's commit length
  - Keep snapshots at no extra cost
  - Benefit for diagnosis and repair
  - Erasure Coding
  - Tradeoff: GC overhead
- **Scaling Compute Separate from Storage**
  - Allows each to be scaled separately
  - Important for multitenant environment
  - Moving toward full bisection bandwidth between compute and storage
  - Tradeoff: Latency/BW to/from storage

Design Consideration (3)

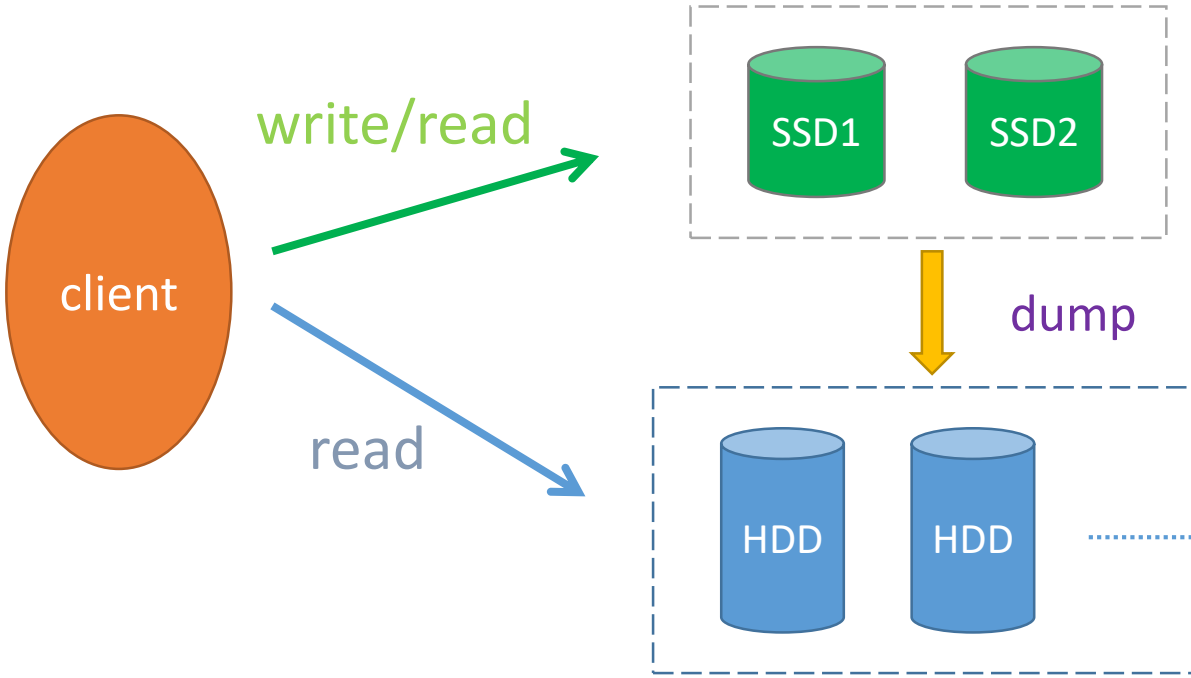
Microsoft Azure

- Automatic load balancing
  - Quickly adapt to various traffic conditions
    - Need to handle every type of workload thrown at the system
  - Built an easily tunable and extensible language to dynamically tune the load balancing rules
  - Need to tune based on many dimensions
    - CPU, Network, Memory, TPS, GC load, Geo-Rep load, Size of partitions, etc.
- Achieving consistently low append latencies
  - Ended up using journaling
- Efficient upgrade support
- Pressure point testing

Ali Pangu Architecture



Ali Pangu - Hybrid Read/Write Mode



- SSD
  - High IOPS
  - Limited Endurance
- HDD
  - High throughput
  - Low IOPS

Pangu→ Functions

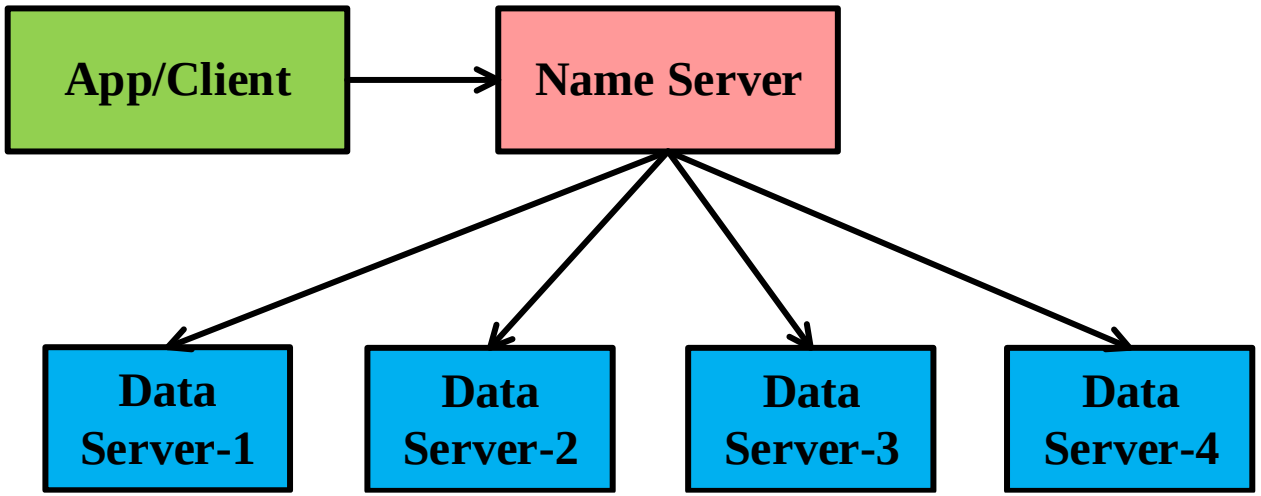
阿里云 aliyun.com

API	• Directory/File Structure • Create/Open/Close/Delete/Rename … Operations • Support Batch Processing/Asynchronous API
Data Security	• Multi-Master • Multiple Replication • Error Detection • Checksum • Garbage Collection
Availability & Performance	• Data Aggregation • Hotspot avoidance • Blacklist • Flow control • Hybrid Storage
Access type	• Random Access • Append-Only
Write Mode for Multiple Replications	• Chain Replication • Direct Replication • Primary-Secondary Replication
Management & Maintaince	• Capability Security Access • Quota Management • Disk Automatic online • Dynamic Scaling • Online Monitoring • Offline Analytics

5 **Project 3**

Distributed File System Design (1)

- Design a Mini Distributed File System (Mini-DFS), which contains
  - A client
  - A name server
  - Four data servers



Distributed File System Design (2)

- Mini-DFS is running through a process. In this process, the name server and data servers are different threads.
- Basic functions of Mini-DFS
  - Read/write a file
    - Upload a file: upload success and return the ID of the file
    - Read the location of a file based on the file ID and the offset
  - File striping
    - Slicing a file into several chunks
    - Each chunk is 2MB
    - Uniform distribution of these chunks among four data servers
  - Replication
    - Each chunk has three replications
    - Replicas are distributed in different data servers

Distributed File System Design (3)

- Name Server
  - List the relationships between file and chunks
  - List the relationships between replicas and data servers
  - Data server management
- Data Server
  - Read/Write a local chunk
  - Write a chunk via a local directory path
- Client
  - Provide read/write interfaces of a file

Distributed File System Design (4)

- Mini-DFS can show
  - Read a file (more than 7MB)
    - Via input the file and directory
  - Write a file (more than 3MB)
    - Each data server should contain appropriate number of chunks
    - Using MD5 checksum for a chunk in different data servers, the results should be the same
    - Check a file in (or not in) Mini-DFS via inputting a given directory
    - By inputting a file and a random offset, output the content

Distributed File System Design (5)

- Bonus points
  - Add directory management
    - Write a file in a given directory
    - Access a file via “directory + file name”
  - Recovery
    - Delete a data server (three data servers survive)
    - Recover the data in the lost data server
    - Redistribute the data and ensure each chunk has three replicas

Thank you!