Distributed File Systems

Jinshu Liu, Pranay Mundra

Distributed File System(DFS) ?

A distributed file system(DFS) is a file system with data stored on one or multiple servers. The data is accessed and processed as if it was stored on the local client machine.

Core Concepts

1. Availability
   - is the system up and running most of the time?
2. Scalability
   - does the performance scale according to the input?
3. Reliability
   - does the system continue to function in case of hardware/software failure?
4. Consistency
   - how does the system handle parallel accesses and imitate and ensure atomicity.

Today's Topics

1. Google File System (GFS)
2. Windows Azure Storage (WAS)
3. General Parallel File System (GPFS)

Google File System (GFS)

Assumptions

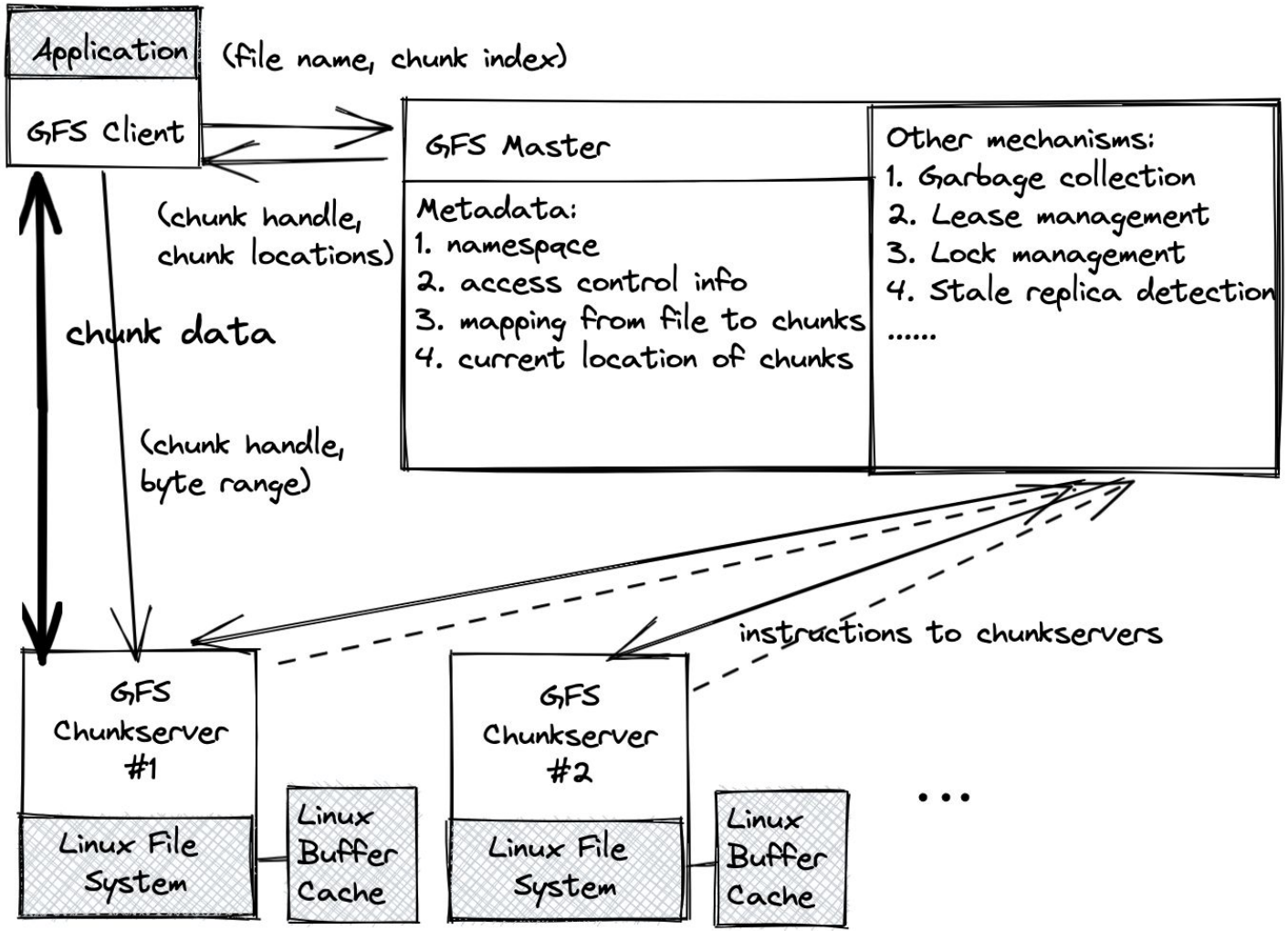
1. **Component failures are norm**
2. **Large files**: 100 MB to multi-GB files
3. Reads: (1) **large streaming reads**; (2) **small random reads**
4. Writes: mostly **large and sequential writes**
5. **Appends concurrently** (producer-consumer queues / many-way merging)
6. **High bandwidth** > low latency

Overview

1. Architecture
2. Operations: read, write, append, snapshot, delete
3. Consistency model
4. Namespace management and locking

Architecture

A master + multiple chunkservers



Architecture

Chunk size: 64 MB

1. reduces clients' need to interact with the master

2. a client is more likely to perform many operations on a given chunk

3. reduces the size of the metadata stored on the master

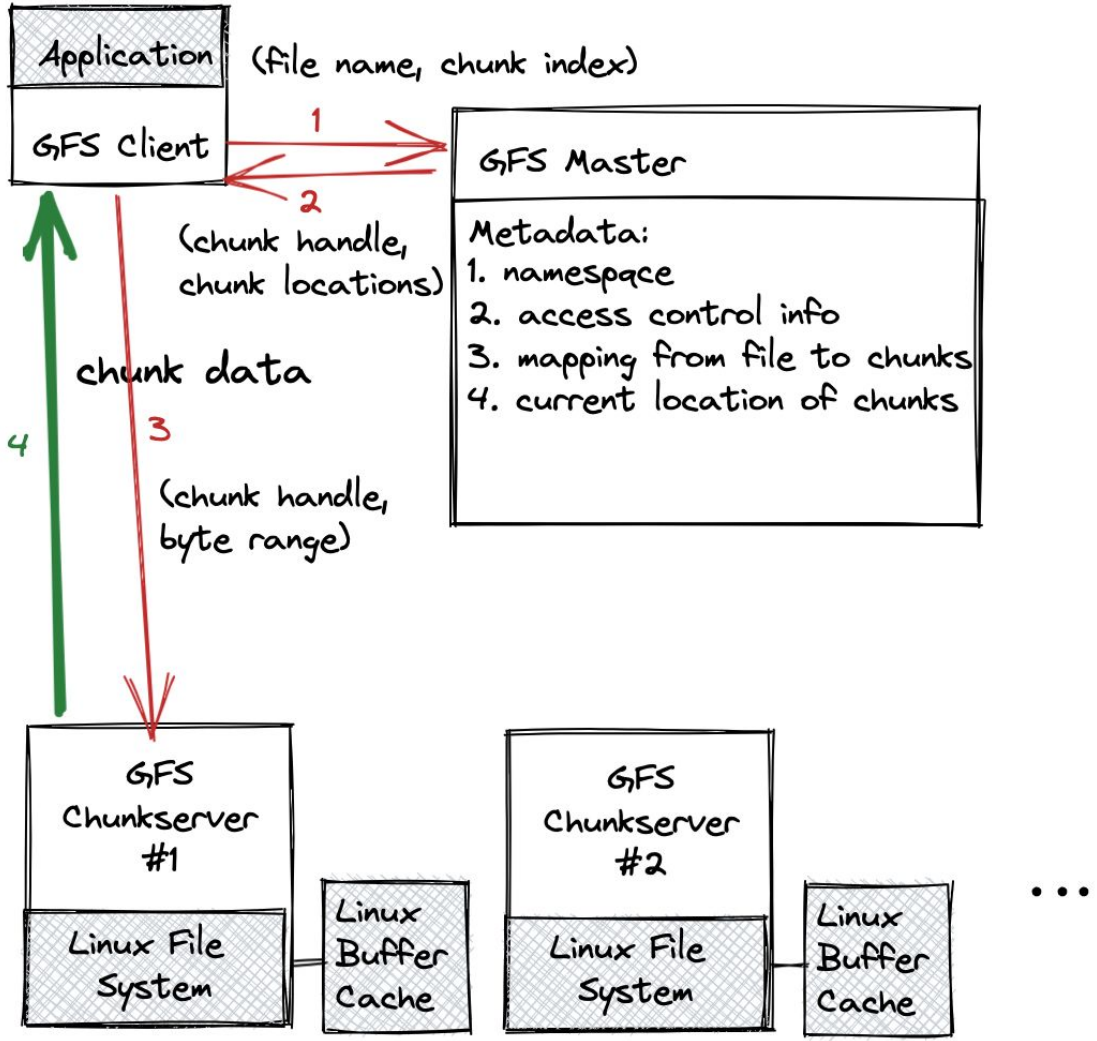
Metadata (In-Memory)

1.the file and chunk namespaces, 2. the mapping from files to chunks, 3. the locations of each chunk's replicas

1 + 2 in operation log, 3 obtained at startup

Operations (Read)

1. sends a request to master
2. master replies with chunk handle and locations of the replicas
3. client then sends a request to one of the replicas, most likely the closest one



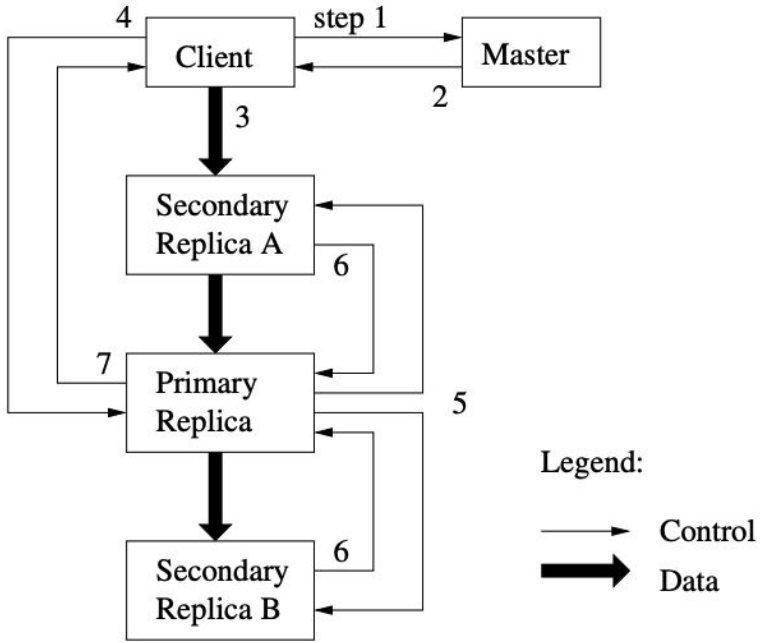
Operations (Write)

Lease mechanism: to maintain a consistent mutation order across replicas

Primary: A replica assigned by the master to pick a serial order for all mutations, other replicas follow this order

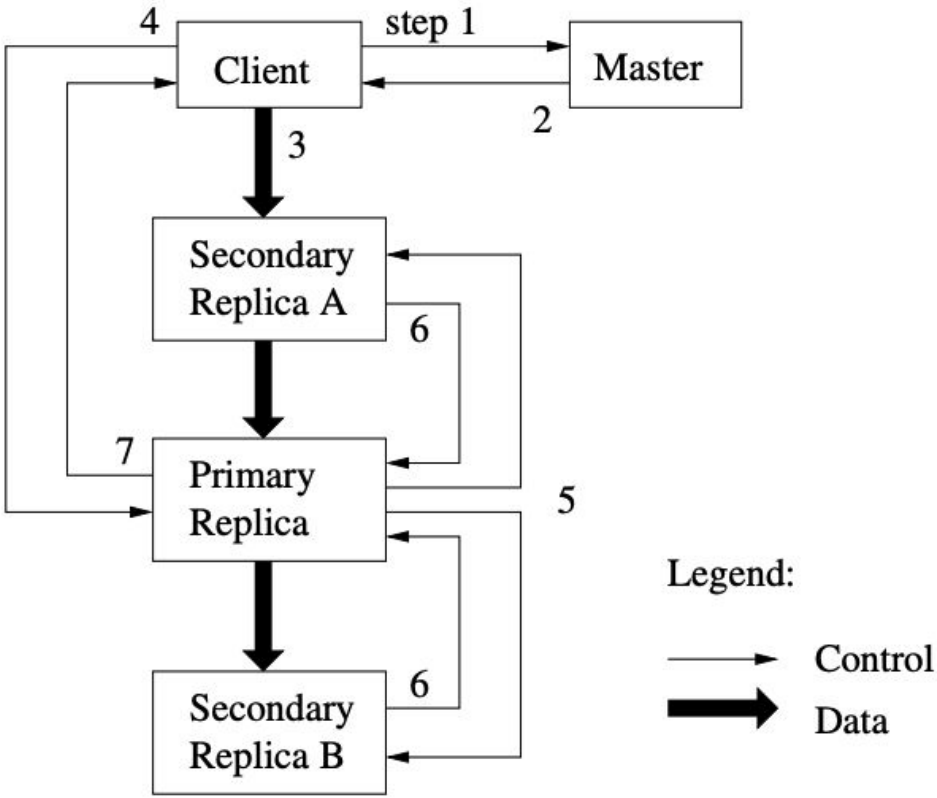
Master grants **lease** to **primary.**

Lease maintains a order, given by **primary**



Operations (Write)

1. The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas.
2. The master gives the info
3. Client pushed data to all replicas
4. Client sends a write request to the primary
5. Primary forwards the write request
6. The secondaries all reply to the primary indicating that they have completed the operation
7. The primary replies to the client



Operations (Write)

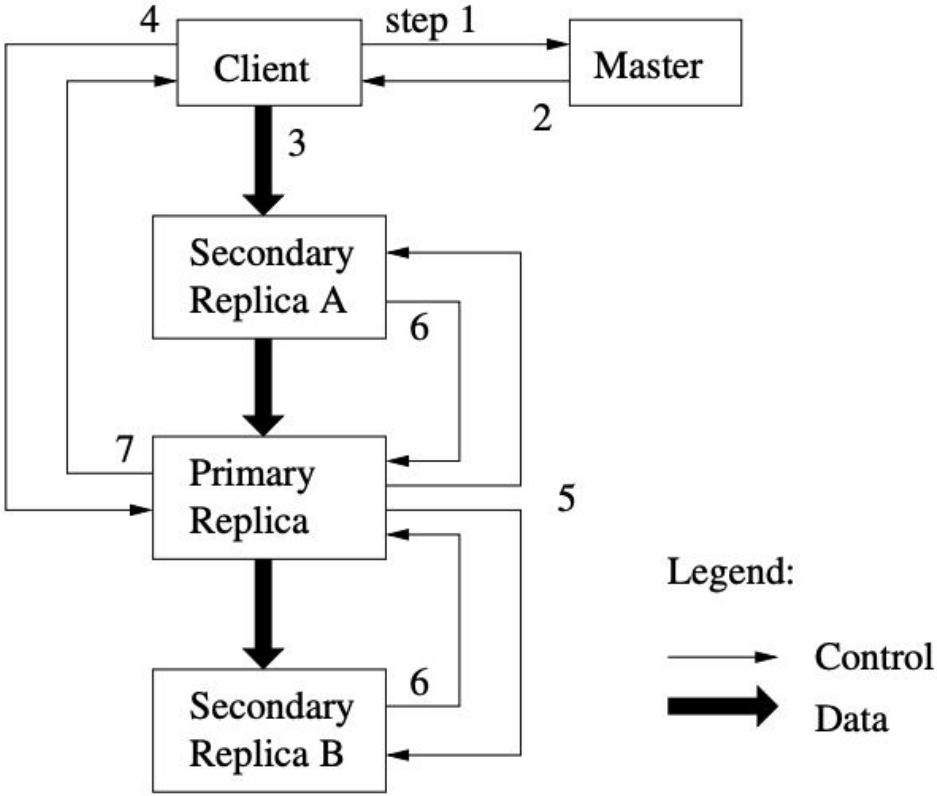
7. Error ? (succeed at the primary and a subset of the secondary replicas.)

↠ Our client code handles such errors by retrying the failed mutation. It will make a few attempts at steps (3) through (7) before falling back to a retry from the beginning of the write.

Decouple data flow and control flow

Data flow:

To fully utilize network bandwidth, the data is pushed linearly along a chain of chunkservers rather than distributed in some other topology (e.g., tree).



Operations (Record append)

Similar to write operation.

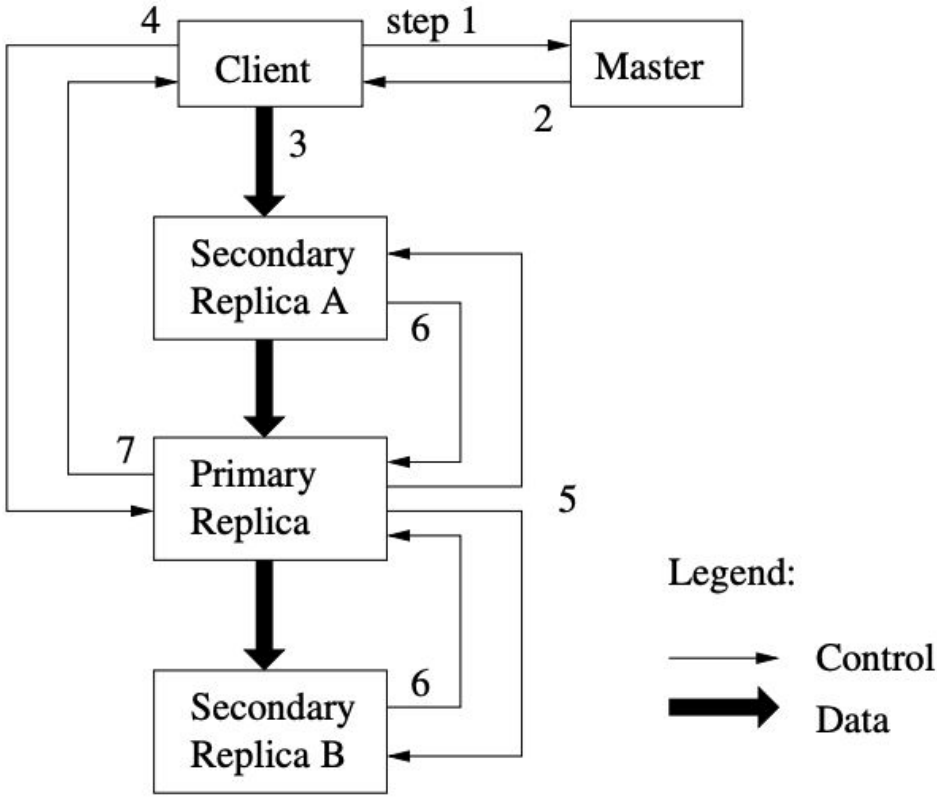
After 4: check,

If to_append > chunk_empty_space:

The primary pads the chunk to the maximum size, tells secondaries to do the same, and replies to the client indicating that the operation should be retried on the next chunk.

Else:

Similar to write operation

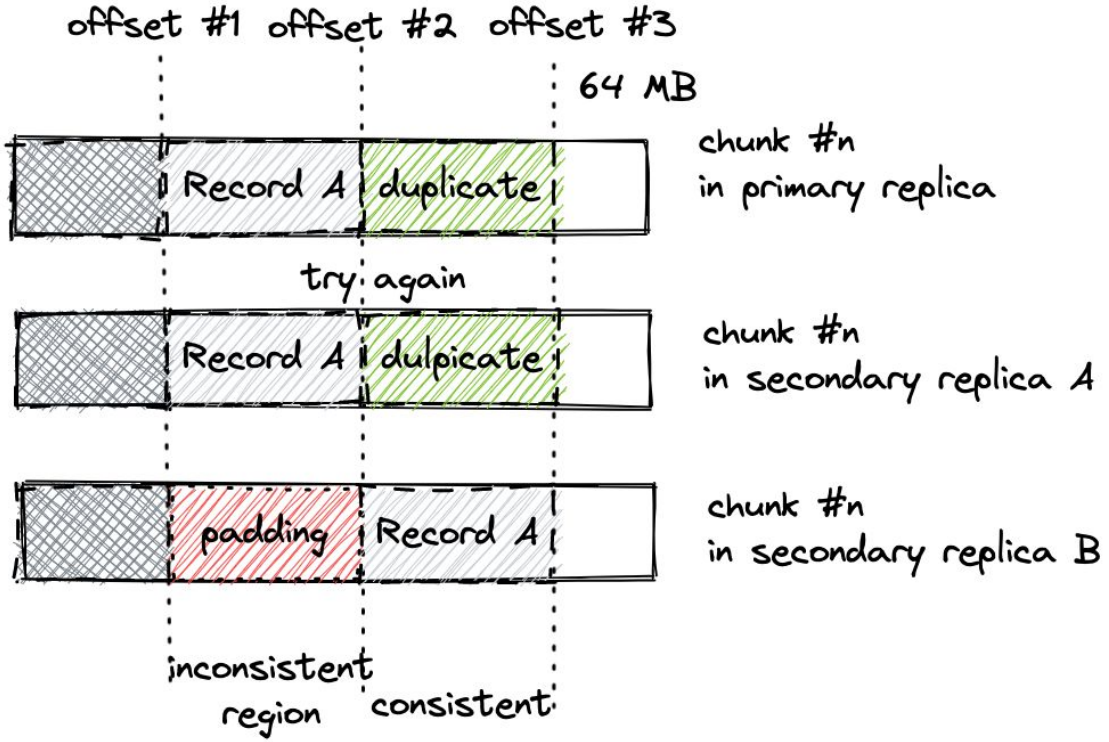


Operations (Record append)

Atomic

At-least-once: If append fails, the client retries the operation

↠ generates inconsistent regions !



Consistency Model

States after a data mutation (write / append):

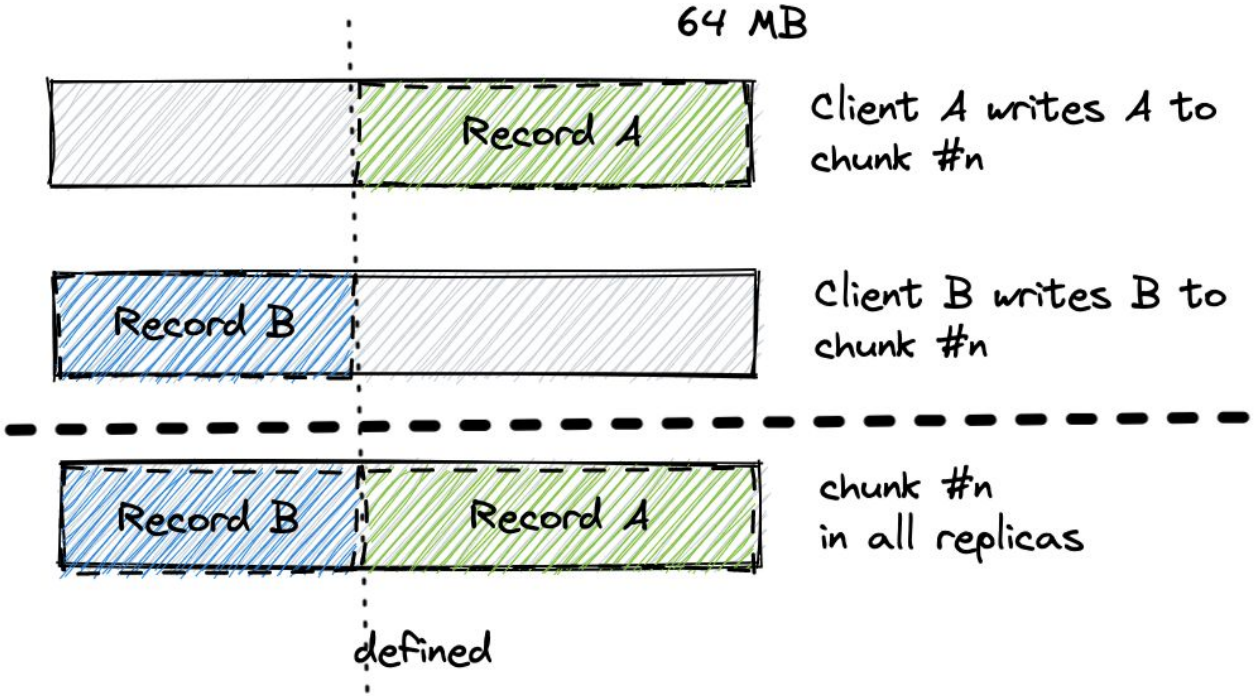
Consistent: all clients will always see the same data, regardless of which replicas they read from

Inconsistent: clients will see the different data from replicas

...

Consistency Model

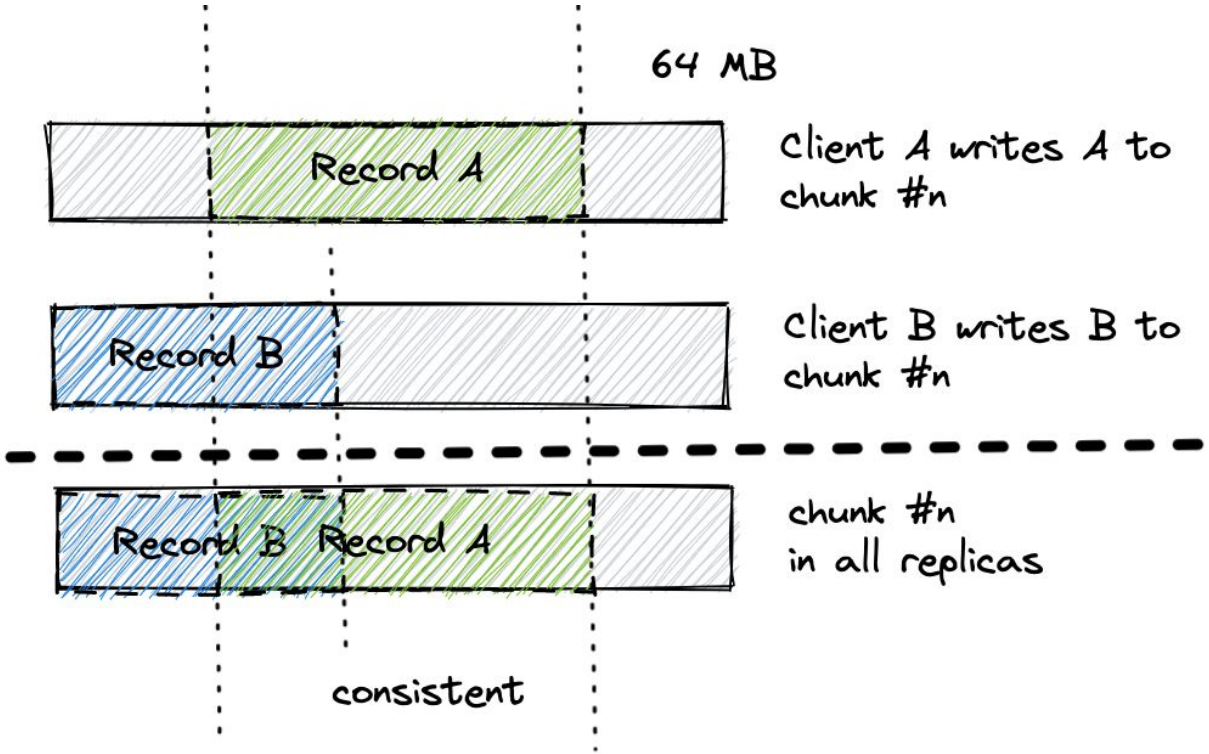
Defined: consistent and clients will see what the mutation writes in its entirety



Consistency Model

Consistent but undefined: all clients see the same data, but it may not reflect what any one mutation has written

(concurrent writes)

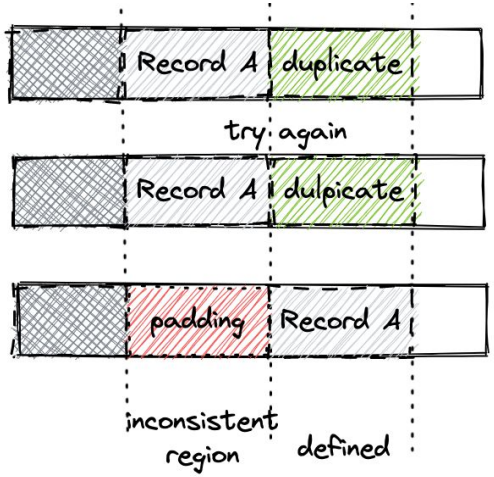


Consistency Model

Defined interspersed with inconsistent:

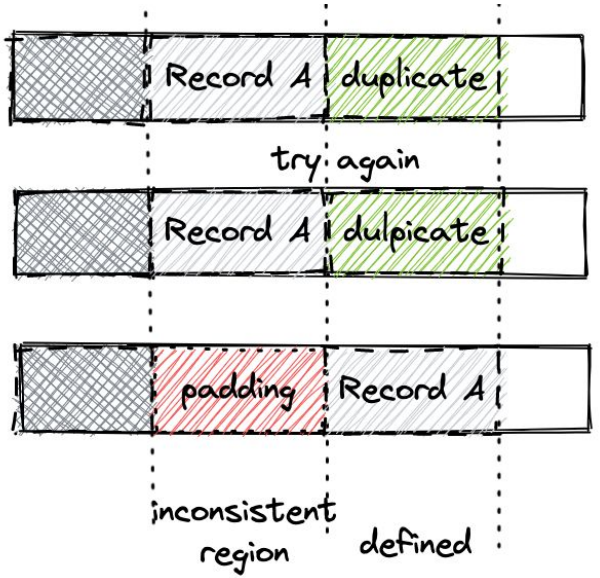
A **record append** causes data (the “record”) to be appended atomically at least once even in the presence of concurrent mutations

GFS may insert **padding** or **record duplicates** in between



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

Operations (Record append)



Append fails ?

The client retries the operation ↠ duplicates and padding

↠ generates inconsistent regions !

How to accommodate this relaxed model for GFS Applications?

For reads:

1. **Checksum**: to discard invalid **record** (e.g., padding)
2. **Unique ID**: to discard duplicates

GFS provides a library for the applications to implement **Checksum** and **Unique ID** easily

Operations (Snapshot)

Make a **copy** of file or a directory tree (exposed to users)

(1) Master first revokes any outstanding leases on the chunks in the files it is about to snapshot.
(2) Master duplicates the metadata for the source file or directory tree

>> copy-on-write

When client writes to a chunk C after the snapshot operation:

(1) Client sends a request to the master to find the current lease holder
(2) The master notices that reference count of C > 1
(3) The master picks a new chunk handle C', asks copy replicas to copy C to C' locally
(4) The master grants one of the replicas a lease on the new C' and replies to the client

Operations (Delete)

1. Master adds deletion to logs
2. Rename the file with a hidden name (the file can still be back to normal)

>>>> After a period of time … (**Lazy garbage collection**)

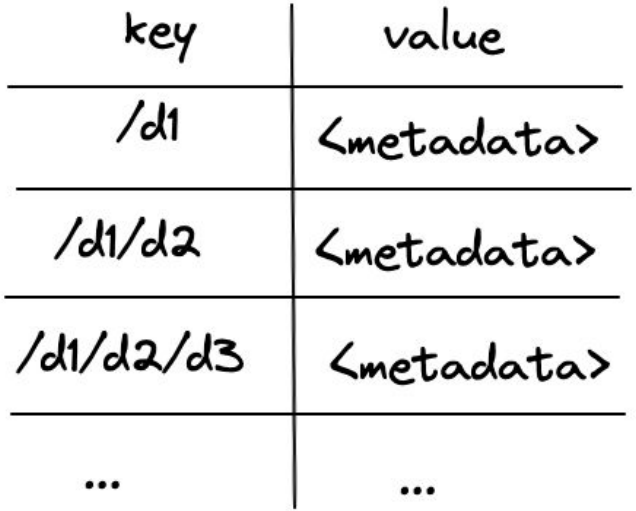
3. During master's **file namespace**'s regular scan, remove the filename
4. During master's **chunks namespace**'s regular scan, remove the chunkname with reference count < 1
5. When chunkserver sends the chunk's information to the master, master cannot find the chunk in chunk namespace, then the master will ask the chunkserver to delete the chunk locally

Namespace Management and Locking

GFS logically represents its namespace as a **lookup table** mapping **full pathnames** to **metadata**

(GFS does not have a per-directory data structure that lists all the files in that directory)

Prefix compression can store the key into memory efficiently



Namespace Management and Locking

Lock:

Example 1, do write operation on /d1/d2/leaf:

Acquire read lock on /d1, /d1/d2, and acquire write operation on /d1/d2/leaf

Example 2: create a file /home/user/foo:

Acquire read lock on /home, /home/user, and acquire write lock on /home/user/foo

Don’t have to acquire write lock on /home/user

Locks are acquired in a consistent total order to prevent deadlock: they are first ordered by level in the namespace tree and lexicographically within the same level

Windows Azure Storage (WAS)

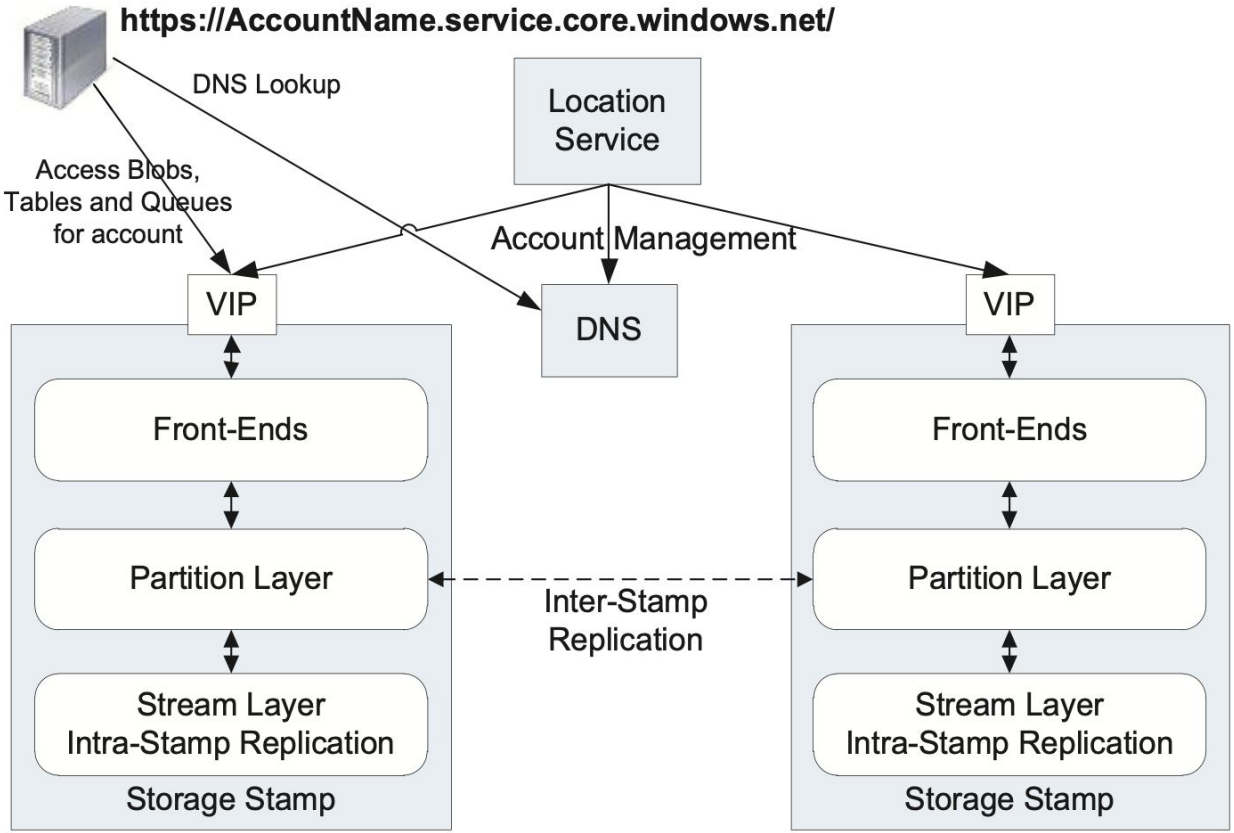
Slide Credits: Calder et al., Windows Azure Storage: A Highly Available Cloud Storage Service with Strong Consistency

Introduction

Scalable Cloud Storage System

Supported Data Abstractions:

- Blobs (user files)
- Tables (structured storage)
- Queues (message delivery)
- Drives (NTFS volumes)



High Level Architecture

WAS Architectural Components

Storage Stamps :

- Cluster of N racks of storage nodes, where each rack is built out as a separate fault domain with redundant networking and power.

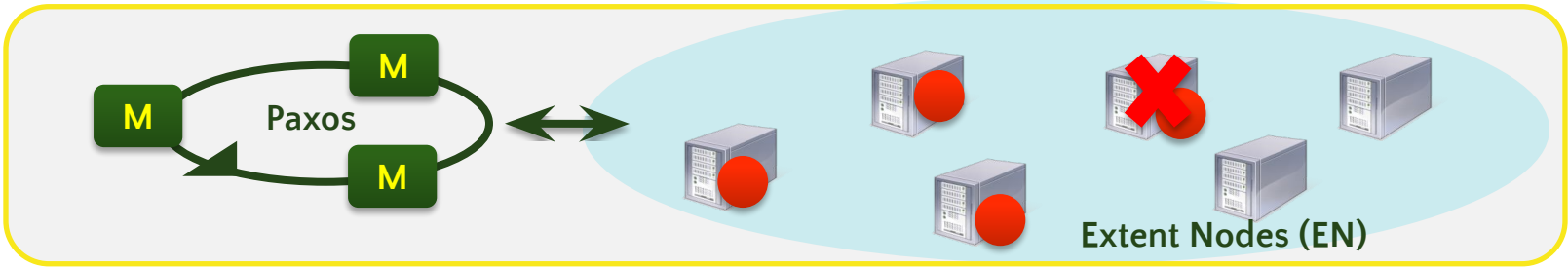
Location Service:

- Manages all storage stamps for disaster management and load balancing
- Allocates accounts to storage stamps
- Distributed across **two** geographical locations for its own disaster recovery

Storage Stamp Architecture – Stream Layer

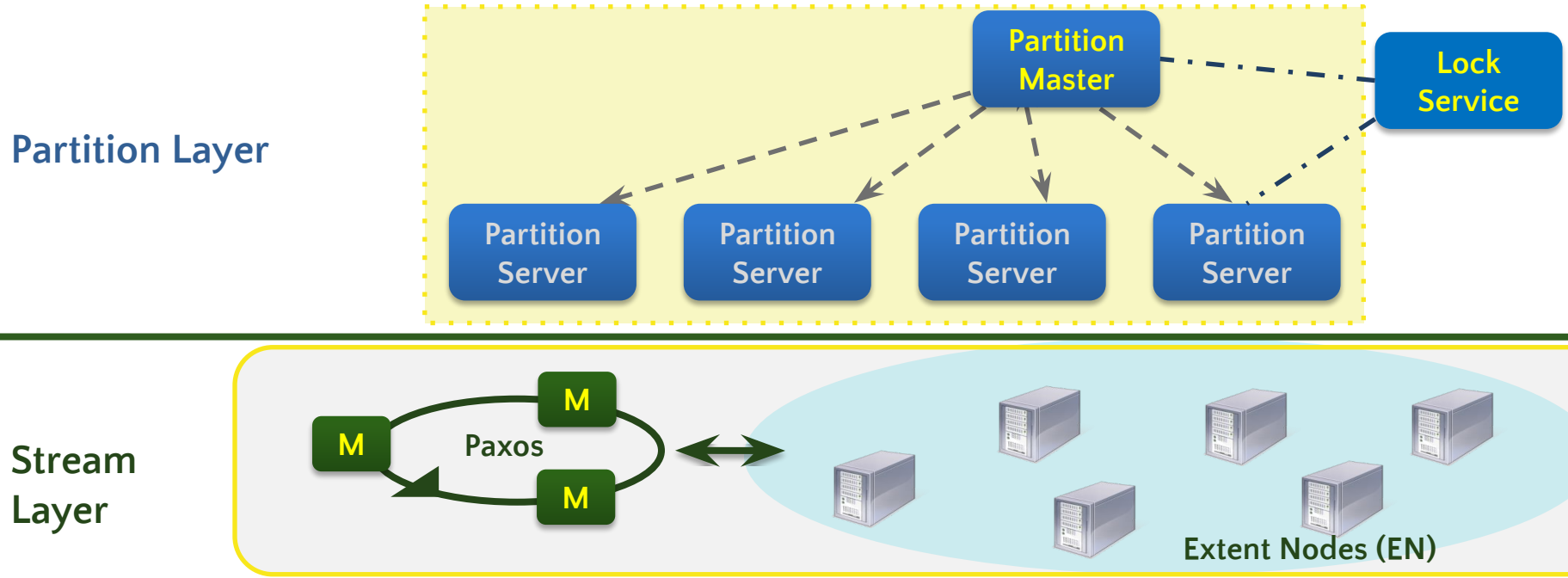
- Append-only distributed file system
- All data from the Partition Layer is stored into files (extents) in the Stream layer
- An extent is replicated 3 times across different fault and upgrade domains
  - With random selection for where to place replicas for fast MTTR
- Checksum all stored data
  -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

Stream Layer (Distributed File System)



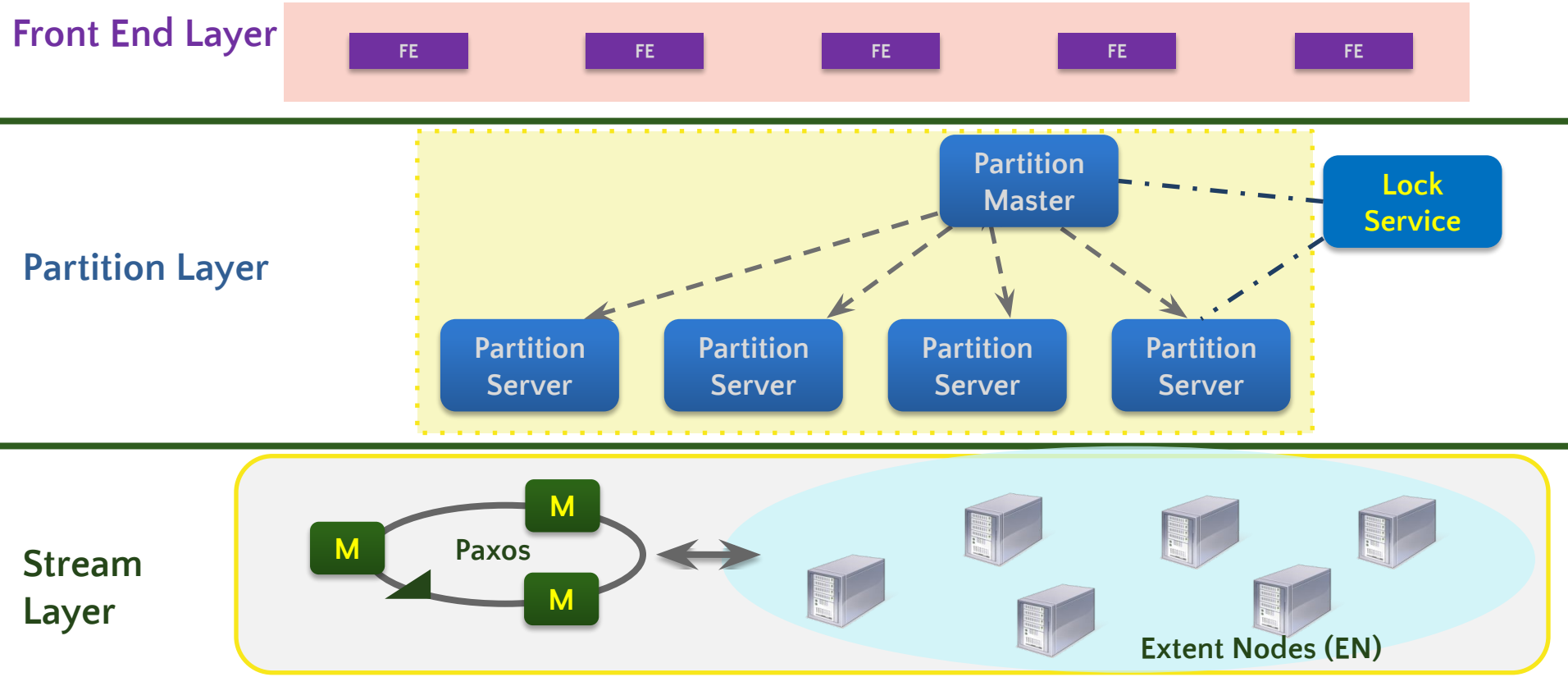
Storage Stamp Architecture – Partition Layer

- 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

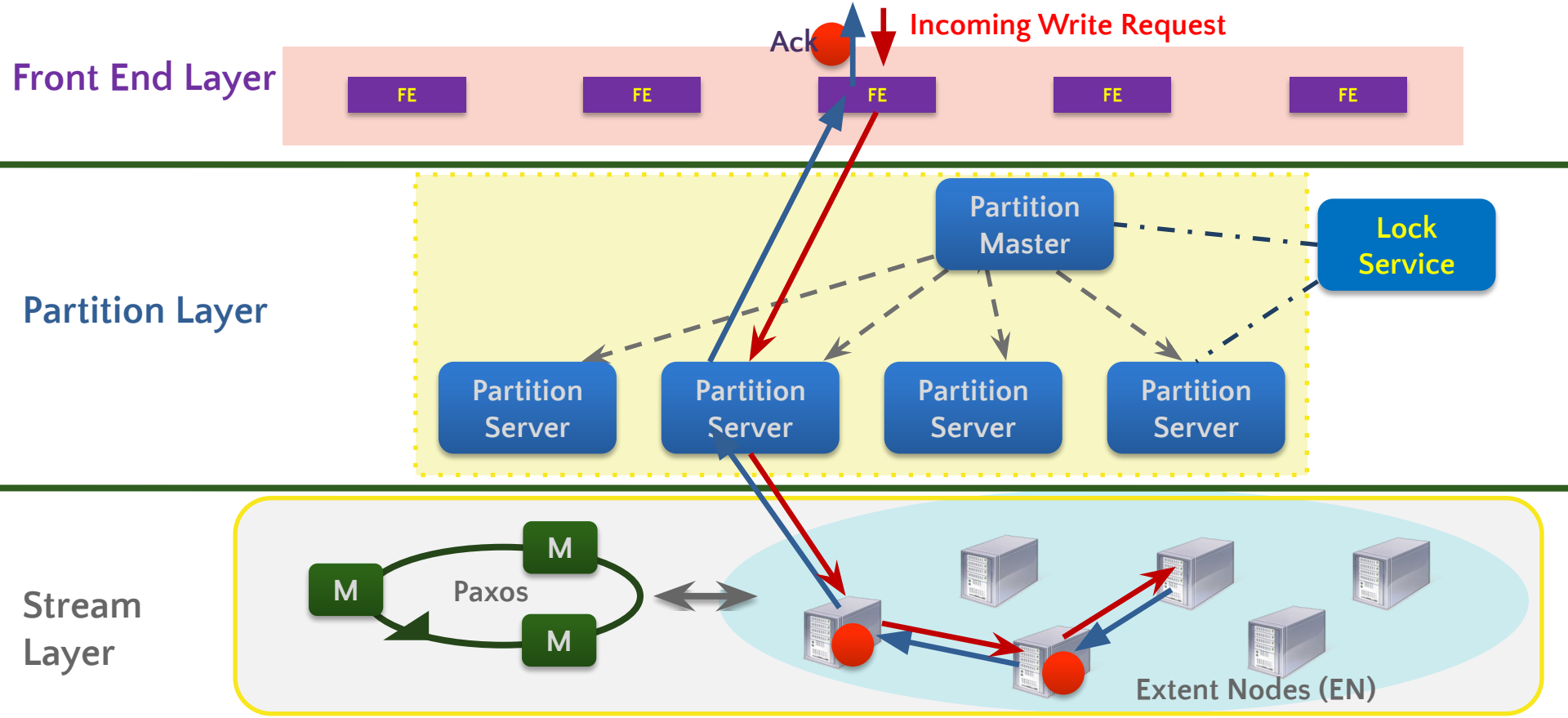


Storage Stamp Architecture

- Stateless Servers
- Authentication + authorization
- Request routing



Storage Stamp Architecture



Partition Layer

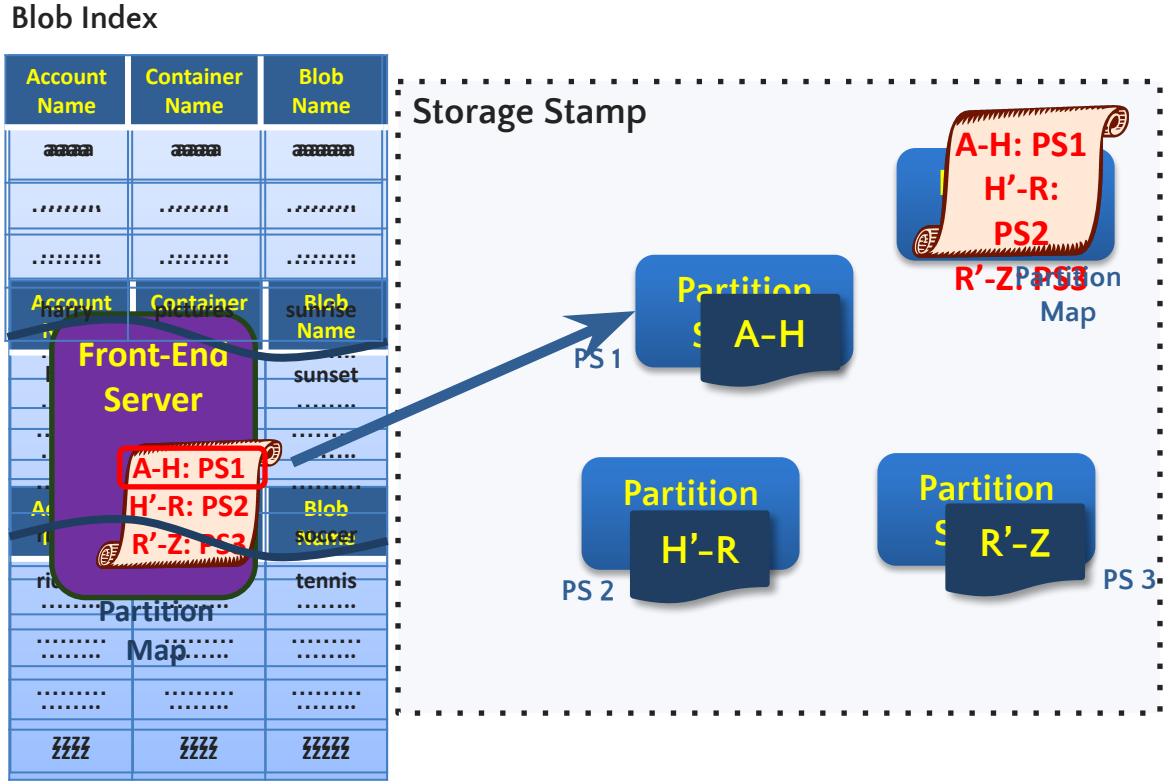
Scalable Object Index via Partitioning

- 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

- 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



Stream Layer

Stream Layer

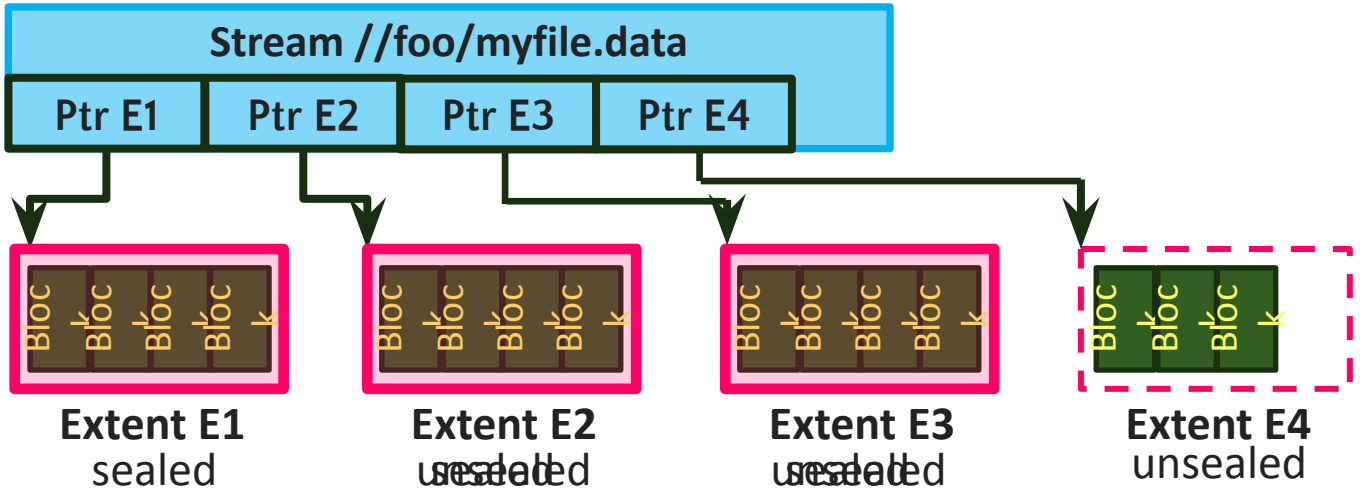
- Append-Only Distributed File System
- Streams are very large files
  - Has file system like directory namespace
- Stream Operations
  - Open, Close, Delete Streams
  - Rename Streams
  - Concatenate Streams together
  - Append for writing
  - Random reads

Stream Layer Concepts

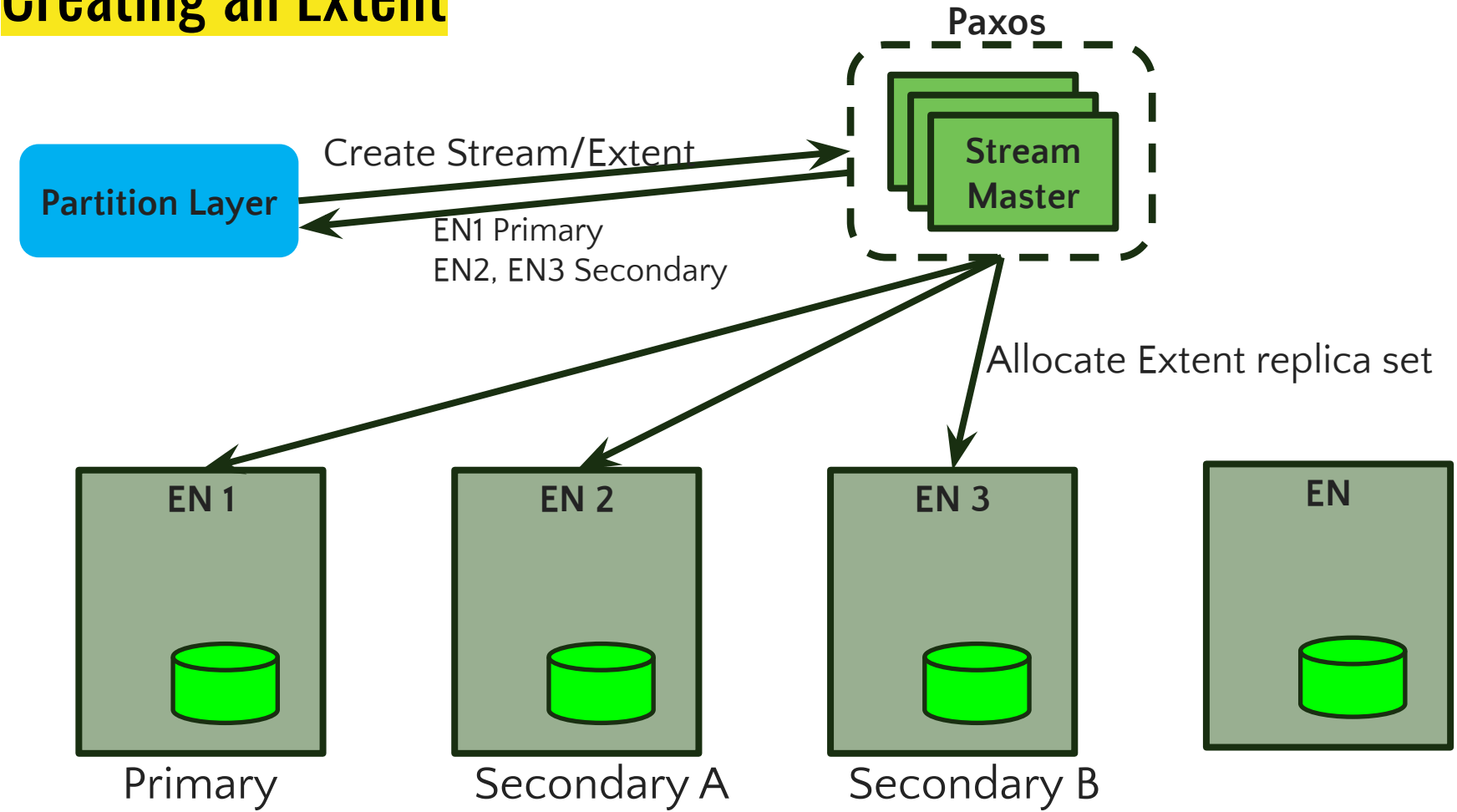
Block
- Min unit of write/read
- Checksum
- Up to N bytes (e.g. 4MB)

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

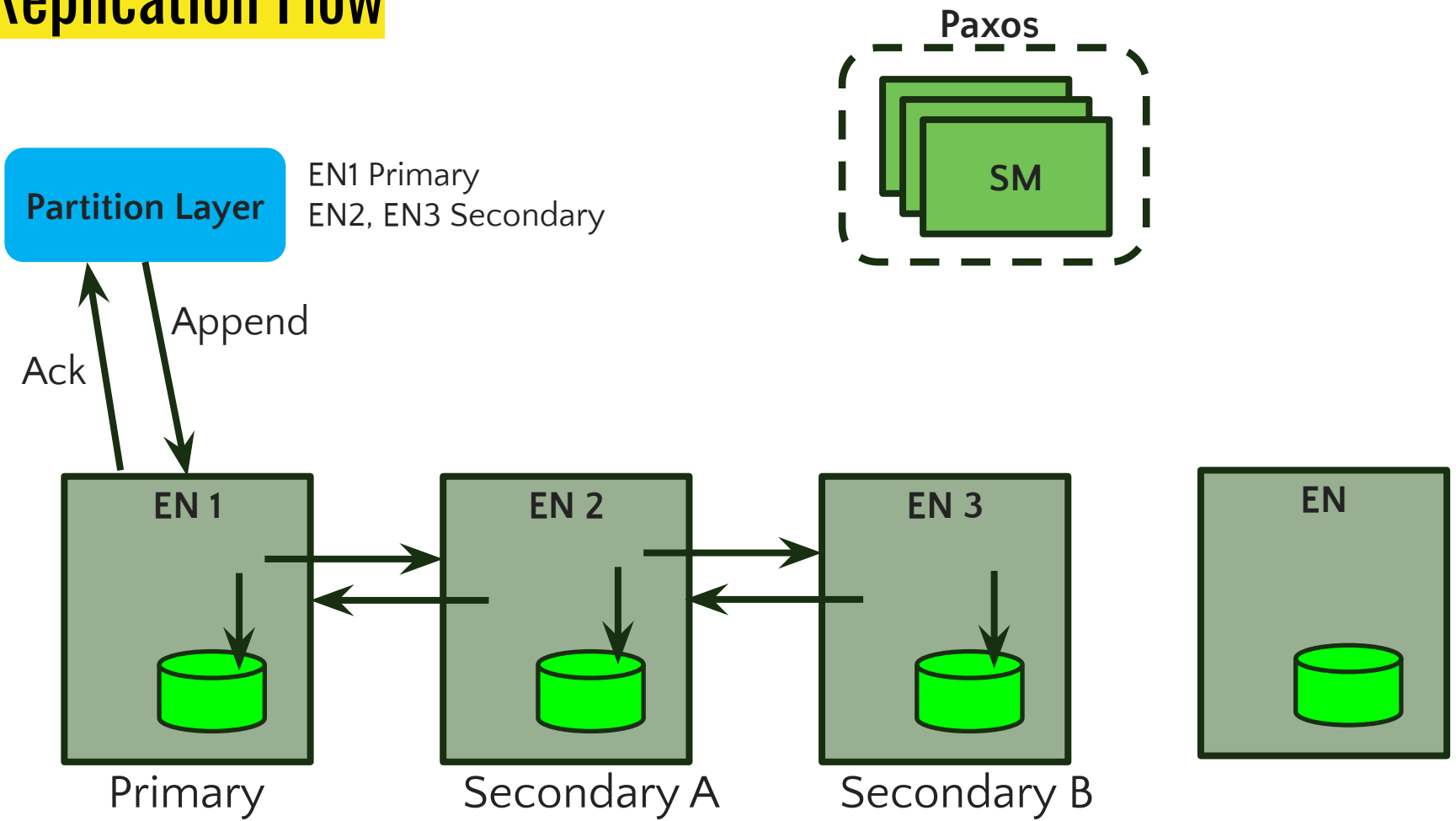
Stream
- Hierarchical namespace
- Ordered list of pointers to extents
- Append/Concatenate



Creating an Extent



Replication Flow



Providing Bit-wise Identical Replicas

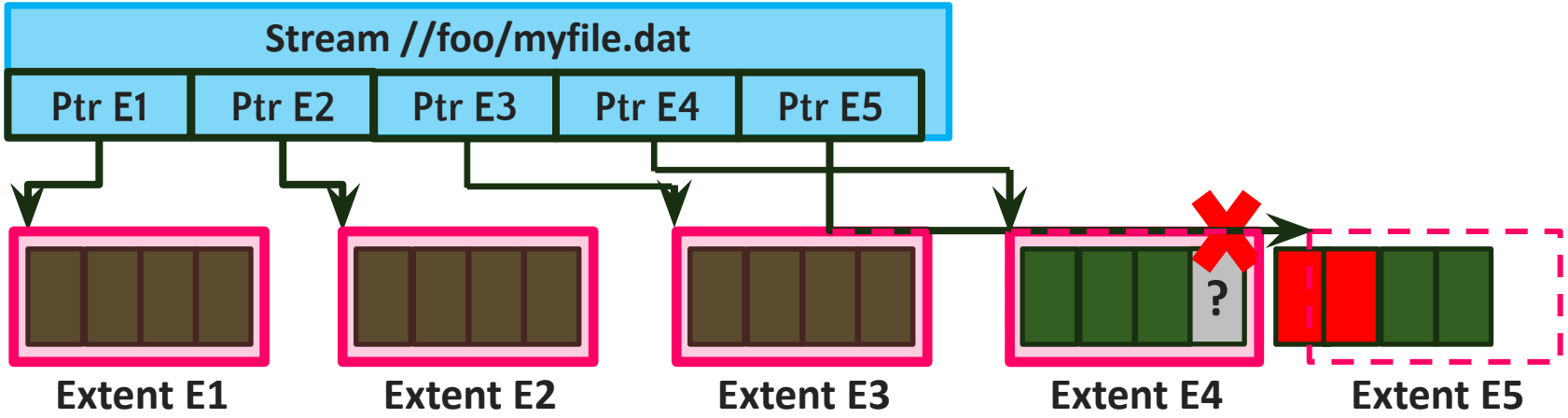
- Want all replicas for an extent to be bit-wise the same, up to a committed length
  - Want to store pointers from the partition layer index to an extent+offset
  - Want to be able to read from any replica

- 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

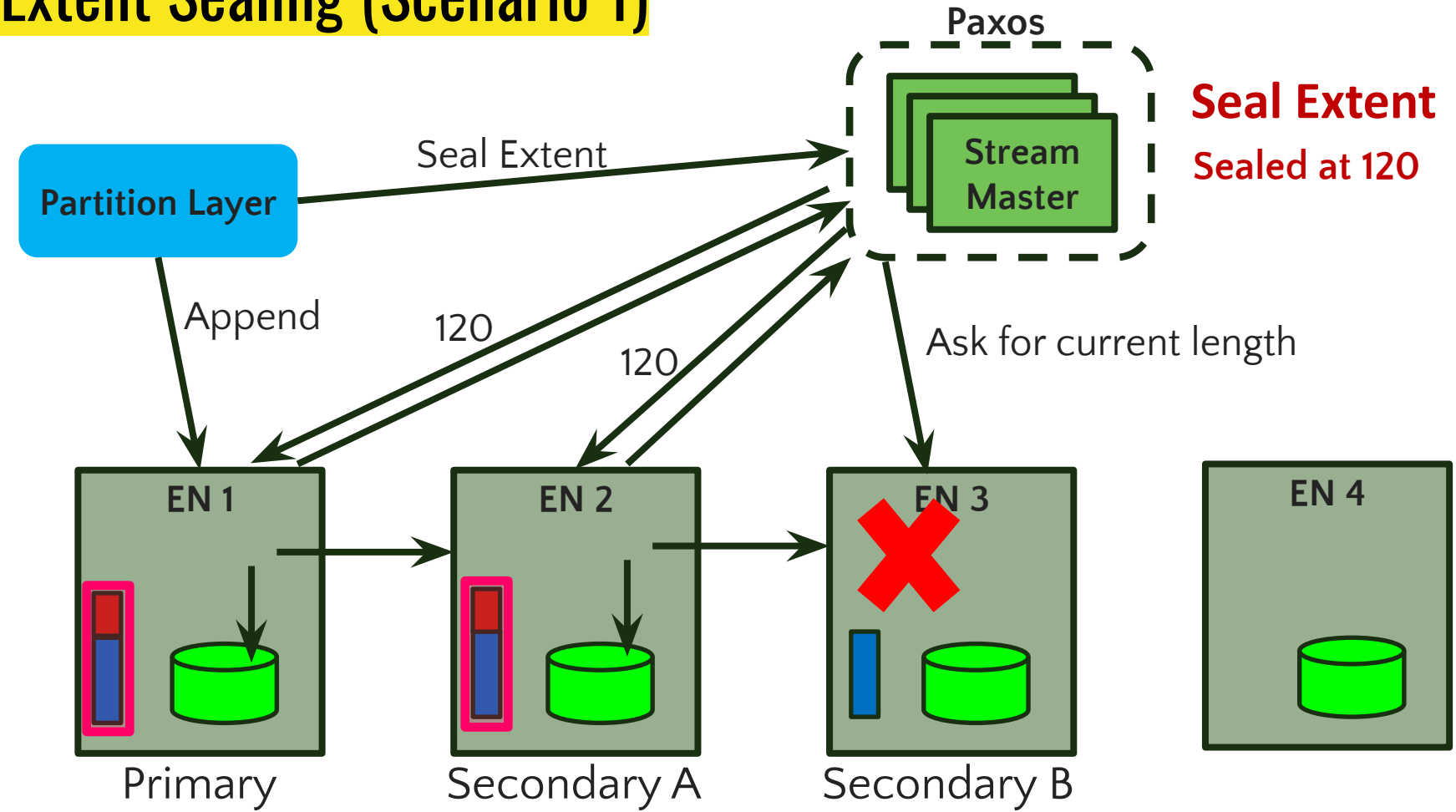
Dealing with Write Failures

Failure during append

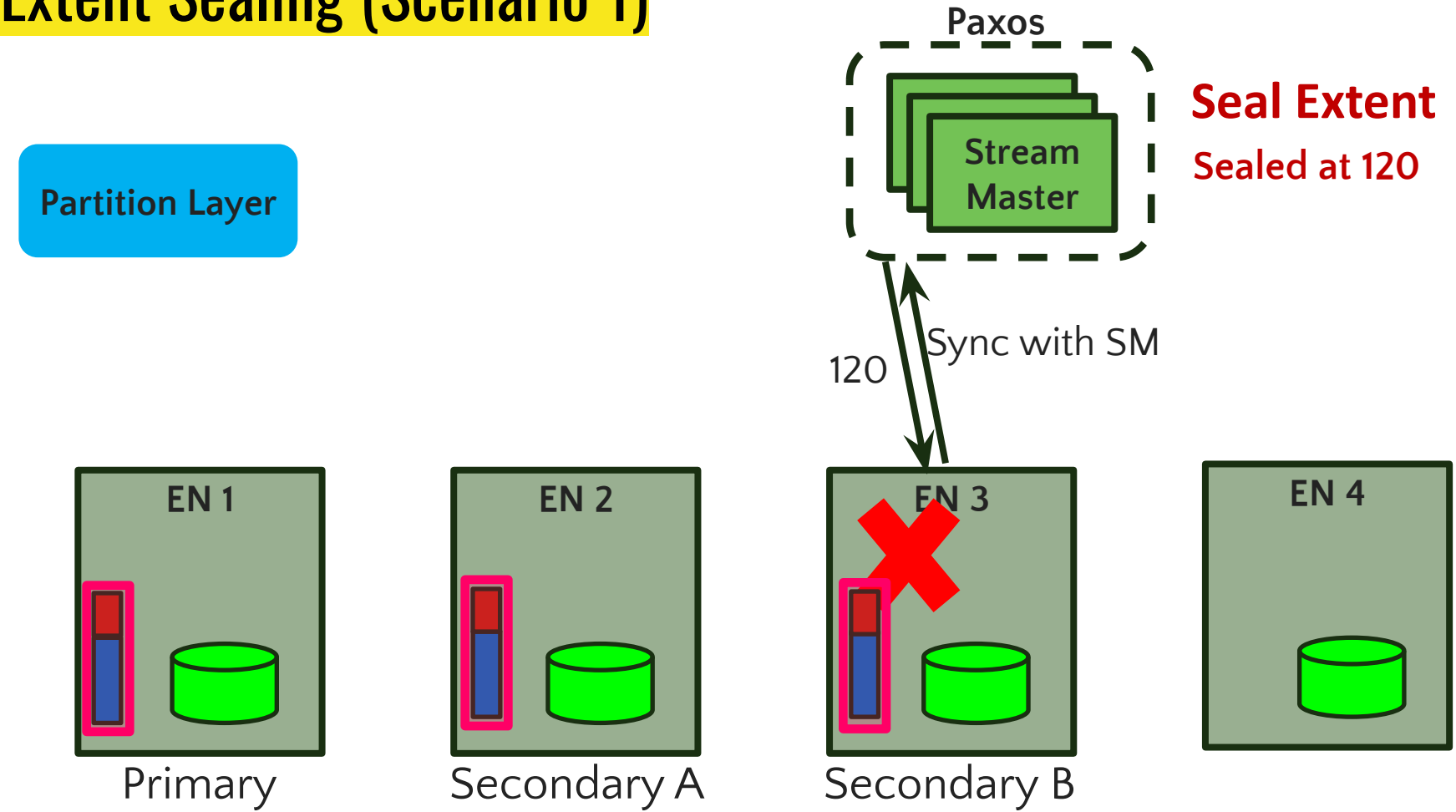
1. Ack from primary lost when going back to partition layer
   - Retry from partition layer can cause multiple blocks to be appended (duplicate records)
2. Unresponsive/Unreachable Extent Node (EN)
   - Append will not be acked back to partition layer
   - Seal the failed extent
   - Allocate a new extent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



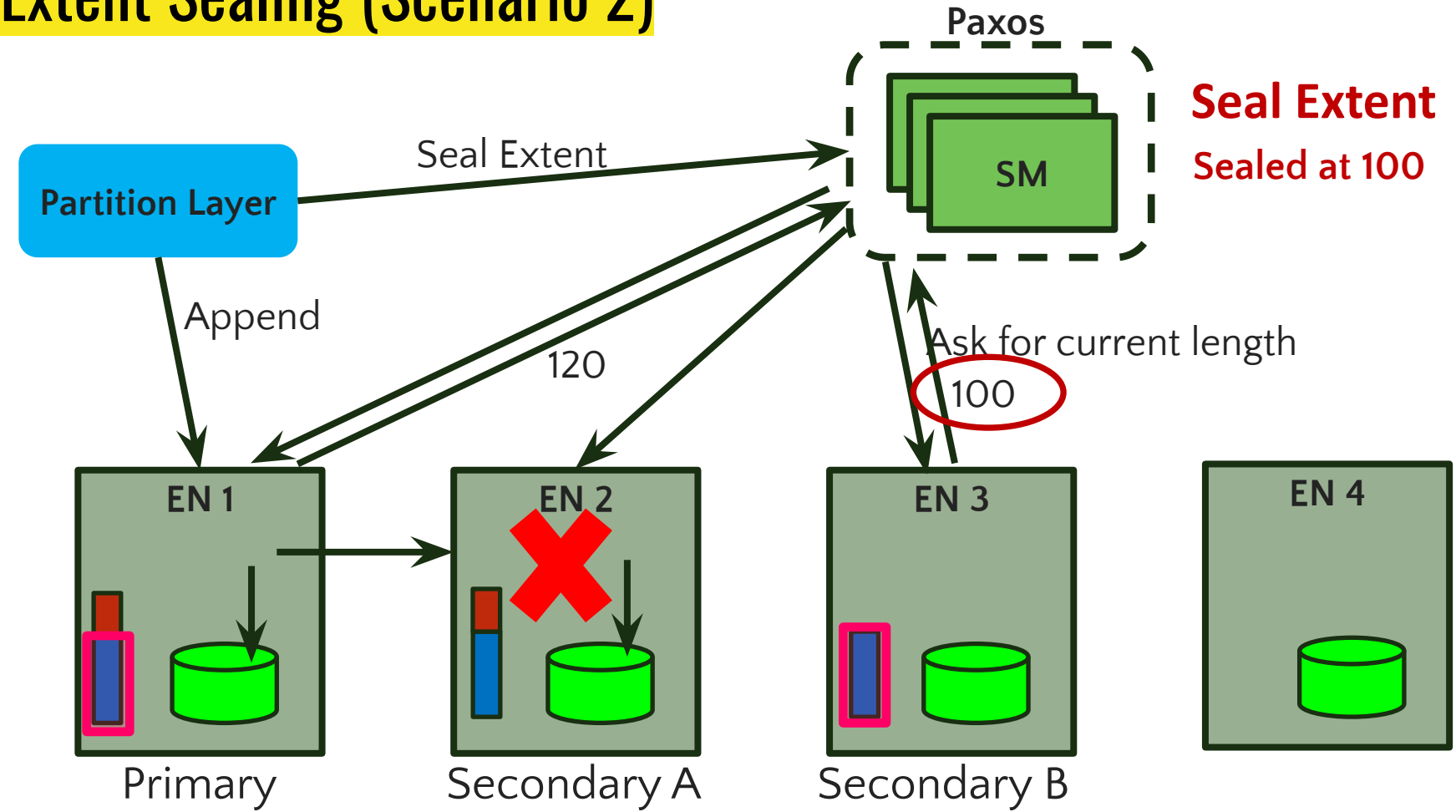
Extent Sealing (Scenario 1)



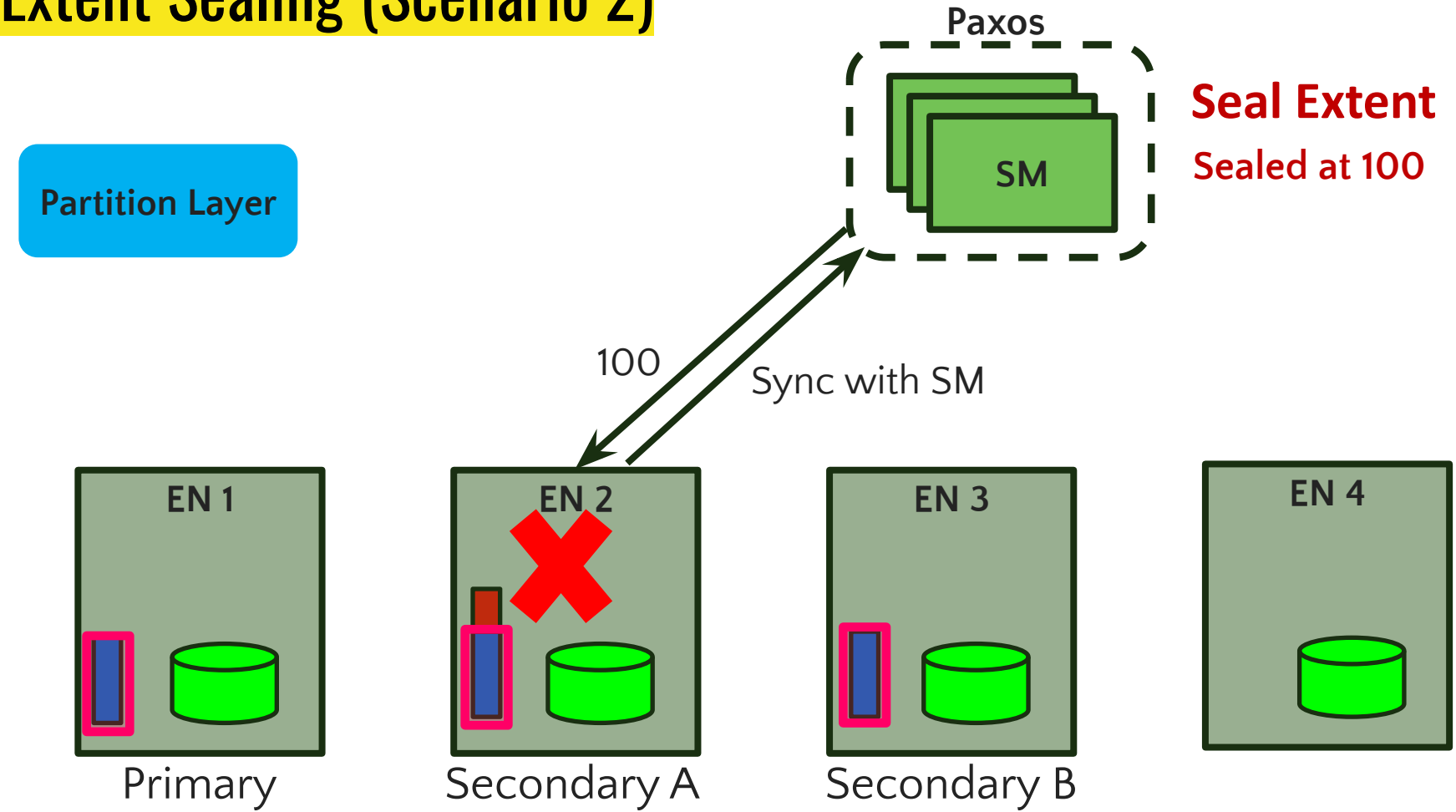
Extent Sealing (Scenario 1)



Extent Sealing (Scenario 2)

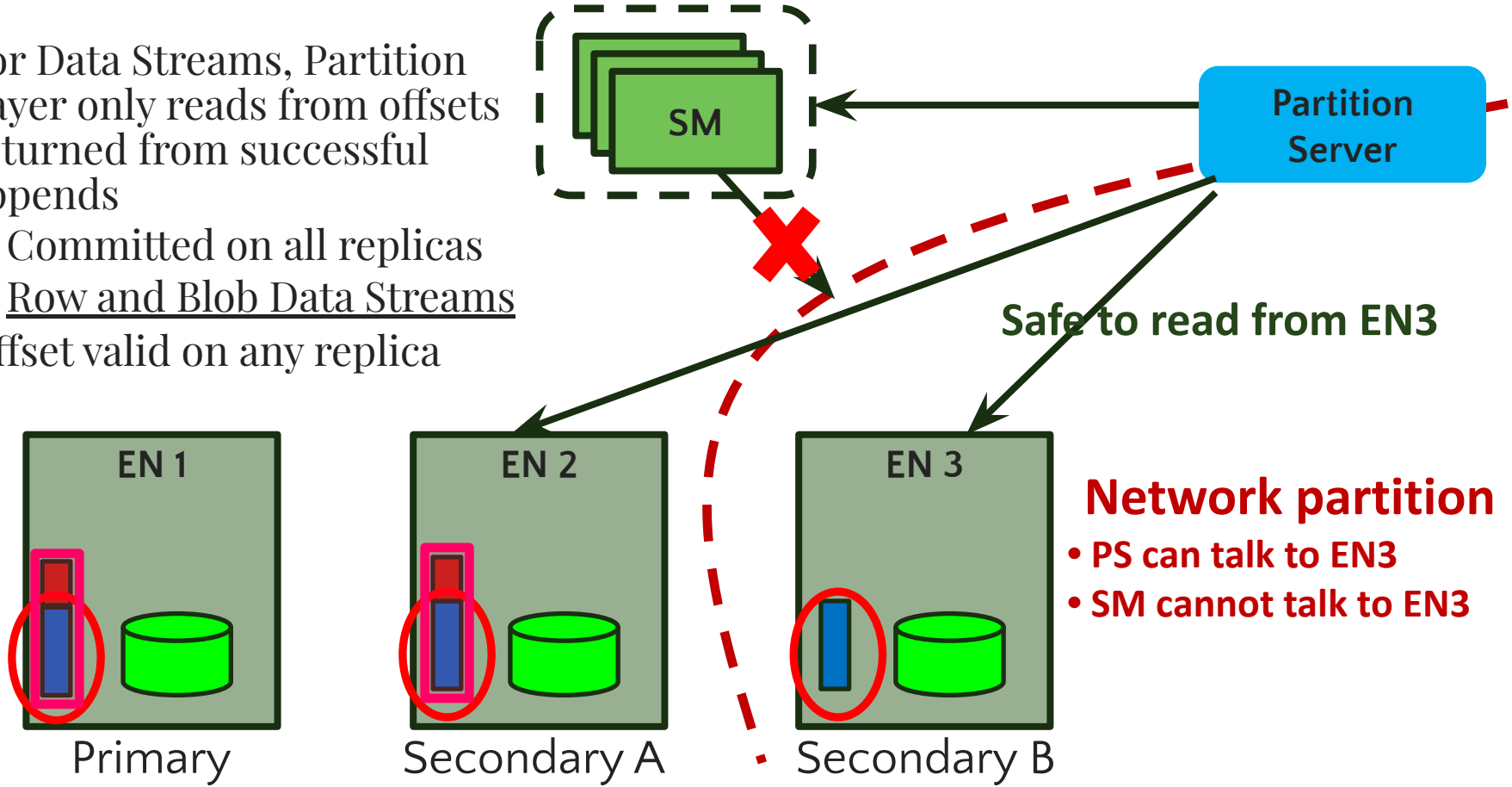


Extent Sealing (Scenario 2)



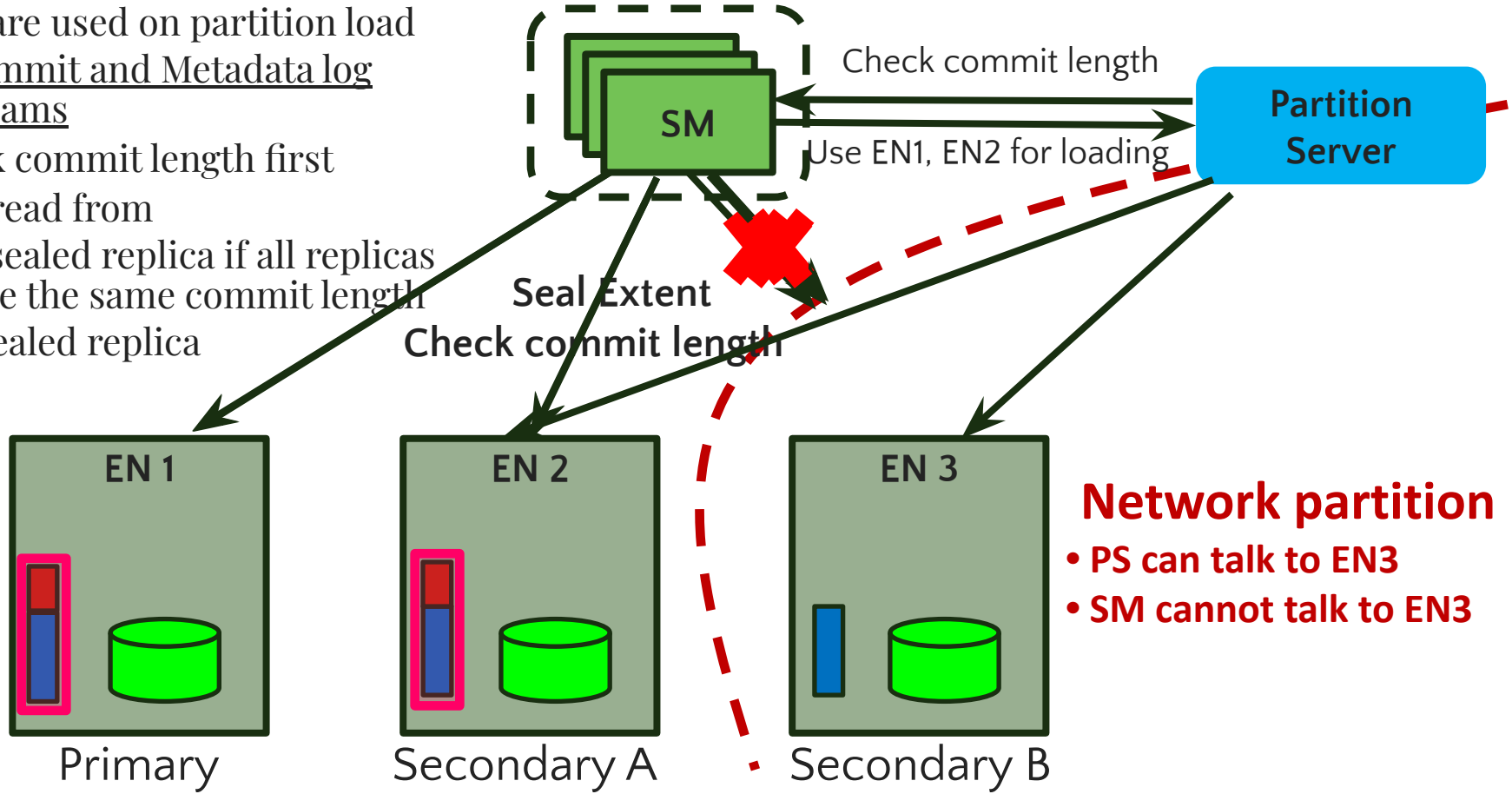
Providing Consistency for Data Streams

- 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

- 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**

WAS approach to CAP Theorem

- Layering and co-design provides extra flexibility to achieve “C” and “A” at same time while being partition/failure tolerant(“P”) for the fault model
  - Stream Layer
    - Availability with Partition/failure tolerance
    - For Consistency, replicas are bit-wise identical up to the commit length
  - Partition Layer
    - Consistency with Partition/failure tolerance
    - For Availability, RangePartitions can be served by any partition server and are moved to available servers if a partition server fails

- Designed for specific classes of partitioning/failures seen in practice
  - Process to Disk to Node to Rack failures/unresponsiveness
  - Node to Rack level network partitioning

General Parallel File System (GPFS - IBM)

Clusters

Collections of machines connected together for the purpose of parallel processing.

Composed of independent and redundant components. More specifically, they are a collection of regular computers put closely together.

Running applications in parallel

Problem

Connect individual computer together, such that, we have fully distributed memory and storage.

How to communicate data between all nodes?

How to handle parallel access to same the file or execution of the same program from different nodes?

Solution - GPFS

- Put all disks together and make them accessible to all nodes through a single point.
- Imitate as closely as possible the behaviour of a general purpose POSIX filesystem running on a single machine
- Conceptually similar to a Big Single Multicore Machine

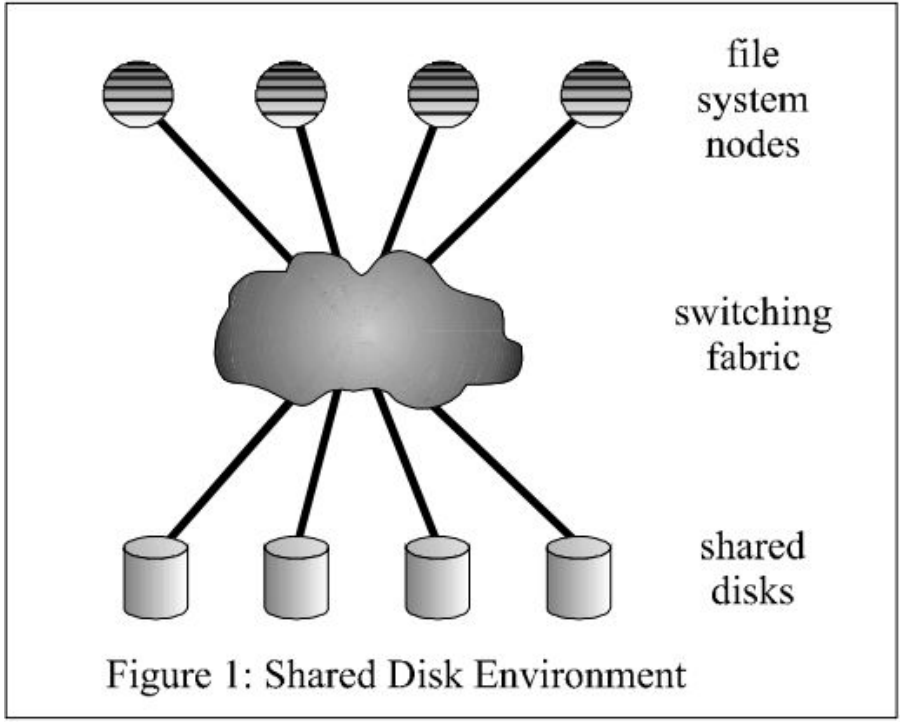


Figure 1: Shared Disk Environment

GPFS Implementation Details

- Synchronization -
  - parallel read/write disk accesses from different nodes
- Load Balancing -
  - equal utilization of all disks
- Logging and Recovery
- Fault Tolerance -
  - node failure, disk failure, communication failure

Synchronization

Distributed locking

- GPFS assigns one of the does as the lock token manager
- Byte range file lock tokens - the first node to access the file gets access to the full range of the file. Any other node there after will get a token in the byte range that they try to write to
- Good for infrequent parallel accesses
- Concurrent writes to the same file to different parts is as fast as writing to different files and then merging.

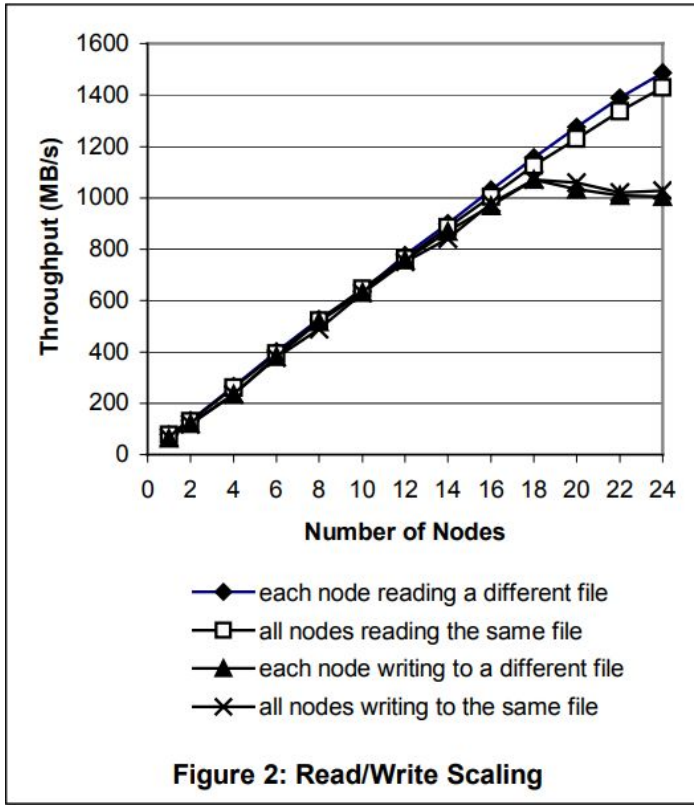


Figure 2: Read/Write Scaling

Synchronization

Centralized Management

- Nodes are assigned parts of files and are solely responsible for all IO to that part of the file
- If a node wants to update part of the file it doesn't own, it must send a request to the node responsible for that part
- Good for frequent parallel accesses and fine grain sharing

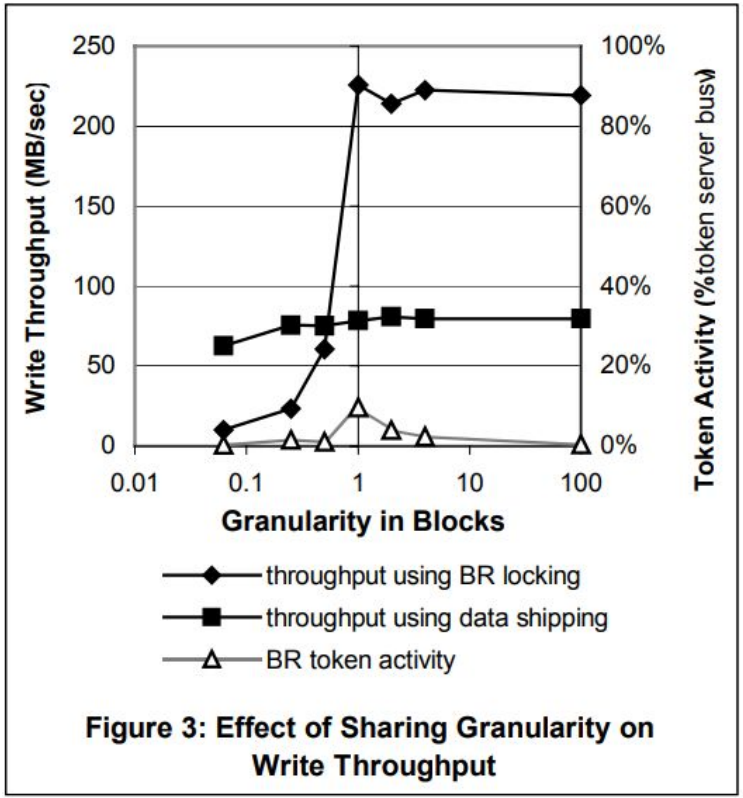


Figure 3: Effect of Sharing Granularity on Write Throughput

Load Balancing

- Data Striping
  - Large writes are divided into equal parts and written to all disks at the same time.
  - Equally distributes the load to all disks
  - Improves Throughput
  - Decreases Latency
- Prefetching and buffering
  - Maintains a BufferPool
  - Allows to prefetch expected data or prepare data for pending requests

Logging & Recovery

- Logs are important in case of failure. They contain information about what the nodes wanted to do to the filesystem.
- Updating the file system is not atomic, hence if a node fails while it was trying to write back to disk, this change should be redoable.
- Each node has a separate log file for each file system it mounts, and is located on that file system
- Since any node can access the log on a file system, any node can perform recovery operations in case of node failures.

Fault Tolerance - Node Failure

- Need to restore the file system and release lock from the failed node
- File system restoration is easy since logs are stored on the file system and can be accessed by any other node
- Lock token protocol ensures that failed node was the only one accessing the data at that time
- Lock token manager releases the lock after recovery procedure completes

Fault Tolerance - Communication Failure

- Some messages cannot reach some nodes, need to be able to detect that
- Periodic heartbeat message service ensures that all nodes are reachable and running
- Monitor node will send out a message to everyone and wait for a response from every single node

Fault Tolerance - Disk Failure

- Hardware failures of the hard drives
- Need data redundancy
  - RAID
  - Replication

GFS vs WAS vs GPFS

1. Google File System and Windows Azure Storage are geographically distributed, where as General Parallel File System is used to create a cluster within a limited region
2. Windows Azure Storage uses a global namespace to narrow down the data location, where as Google File System uses a single master and allows the client to talk to each chunk server (partition server counterpart) directly.