Google File System

CSCI 333

Spring 2020

Last Class

BetrFS

- Full-path-indexing vs. inode-based designs
- Mapping VFS operations to B^{e}-tree operations
- Evaluating a system fairly

This Class

Google file system

- Who?
- Why?
- How?

Who?

When Reading a Paper

Look at authors

Look at institution

Look at past/future research

Look at publication venue

These things will give you insight into the

- motivations
- perspectives
- agendas
- resources

Think: Are there things that they are promoting? Hiding? Building towards?

Why?

Thought Experiment

Suppose you want to run a workload that does distributed batch processing (e.g., I have a bunch of data and I want to compute over various independent subsets of that data in parallel).

- What bottlenecks would I run into if I ran this workload on NFS?

Suppose I instead store my data as a bunch of files on different nodes in my "private cloud" of servers.

- What advantages do I get over NFS?
- What types of events/problems do I need to design my s system to handle?

GFS Design Targets/Constraints

Large files (and millions of them)

Frequent component failures

Append-only writes dominate the updates

Large sequential reads

Prioritize high sustained bandwidth over latency

No need to be strictly POSIX compliant, but must support:

- Standard ops:
  - read, write, open, close, create, delete
- Non-standard ops:
  - **Snapshot**: a copy of a file or directory tree at low cost
  - **Record append**: allows multiple clients to append to the same file concurrently, guaranteeing atomicity of each append

How?

Design

Files are divided into fixed-sized chunks

Clients write files to chunk servers

A single master server coordinates the system, but the “real work” happens locally at individual nodes

GFS Architecture

A single master multiple chunk servers

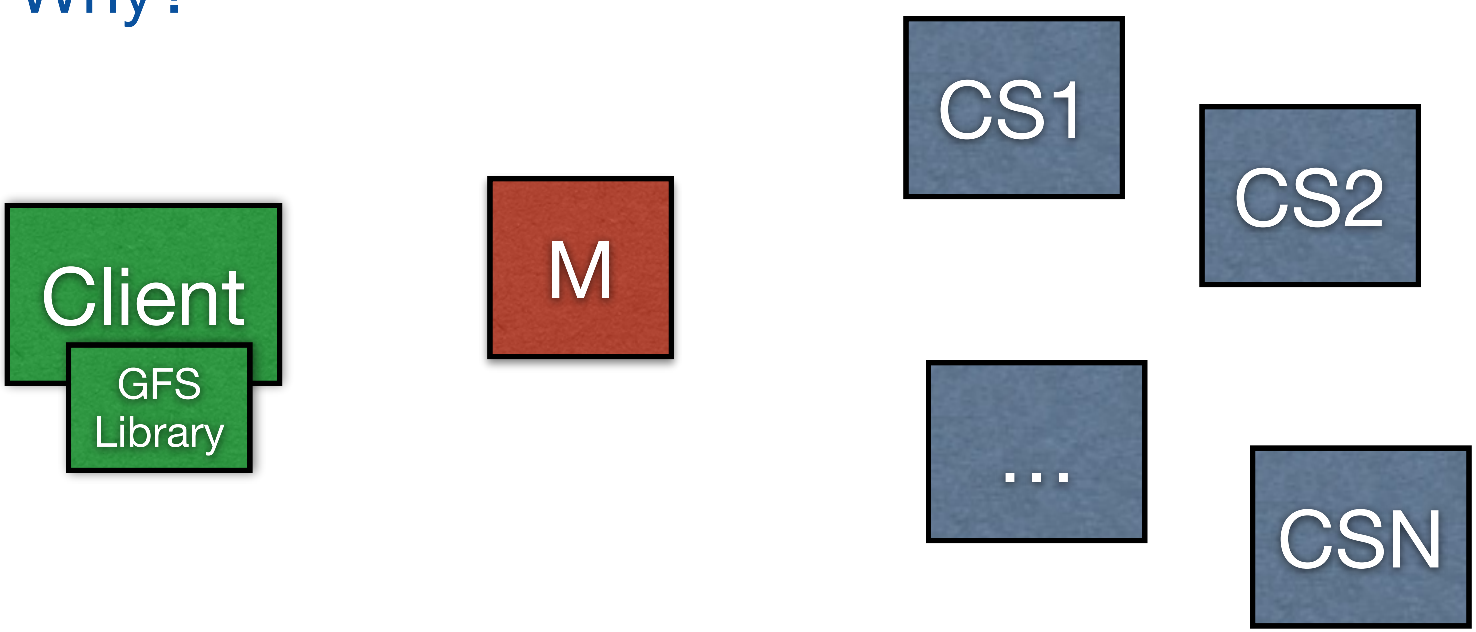
Files are composed of 64MiB chunks

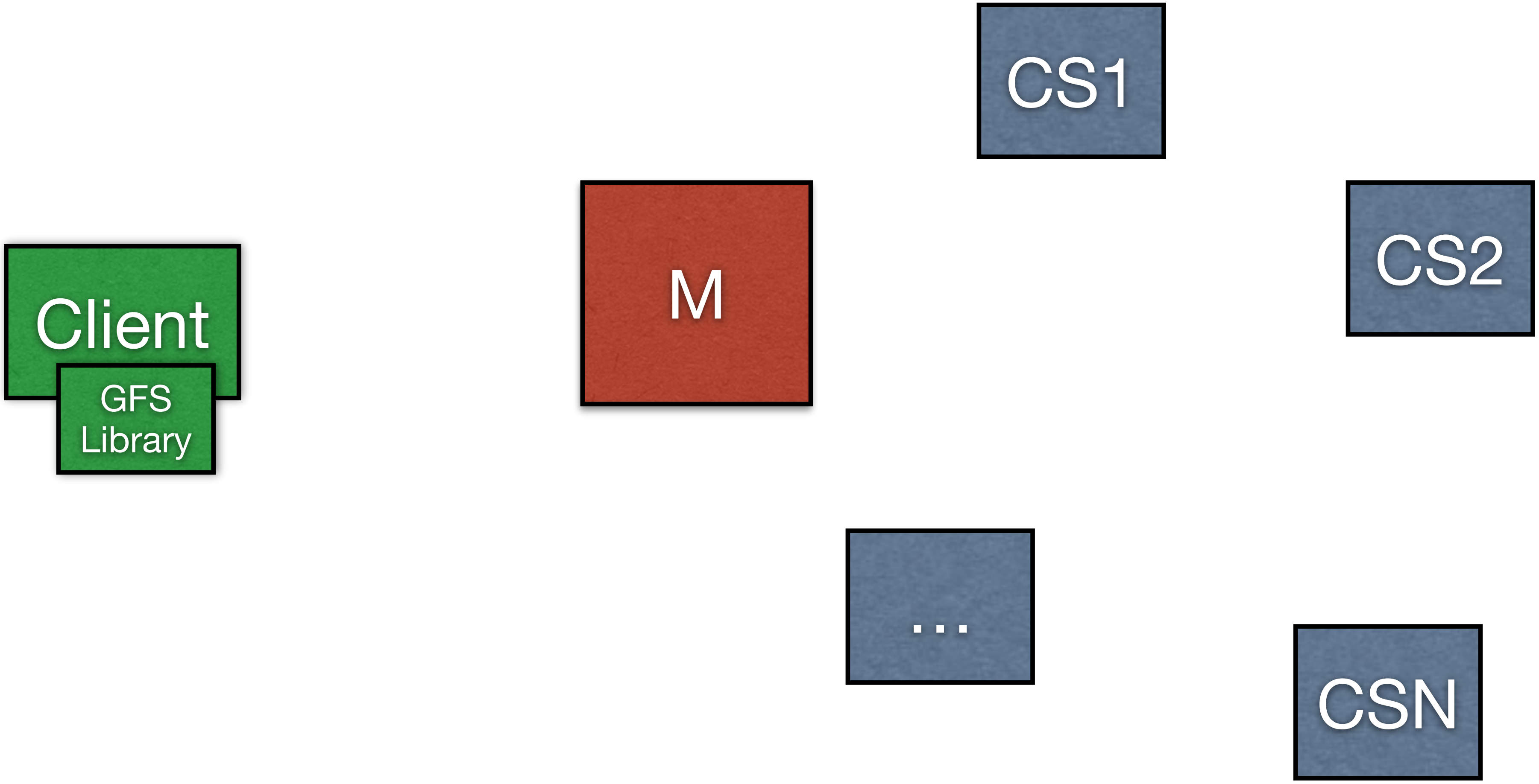
- Chunks are represented as local files on chunk server FSes

Chunks are replicated (3 copies by default)

FS interface provided by a client library, not VFS

- Why?





Client
GFS
Library
M
CS1
CS2
...
CSN

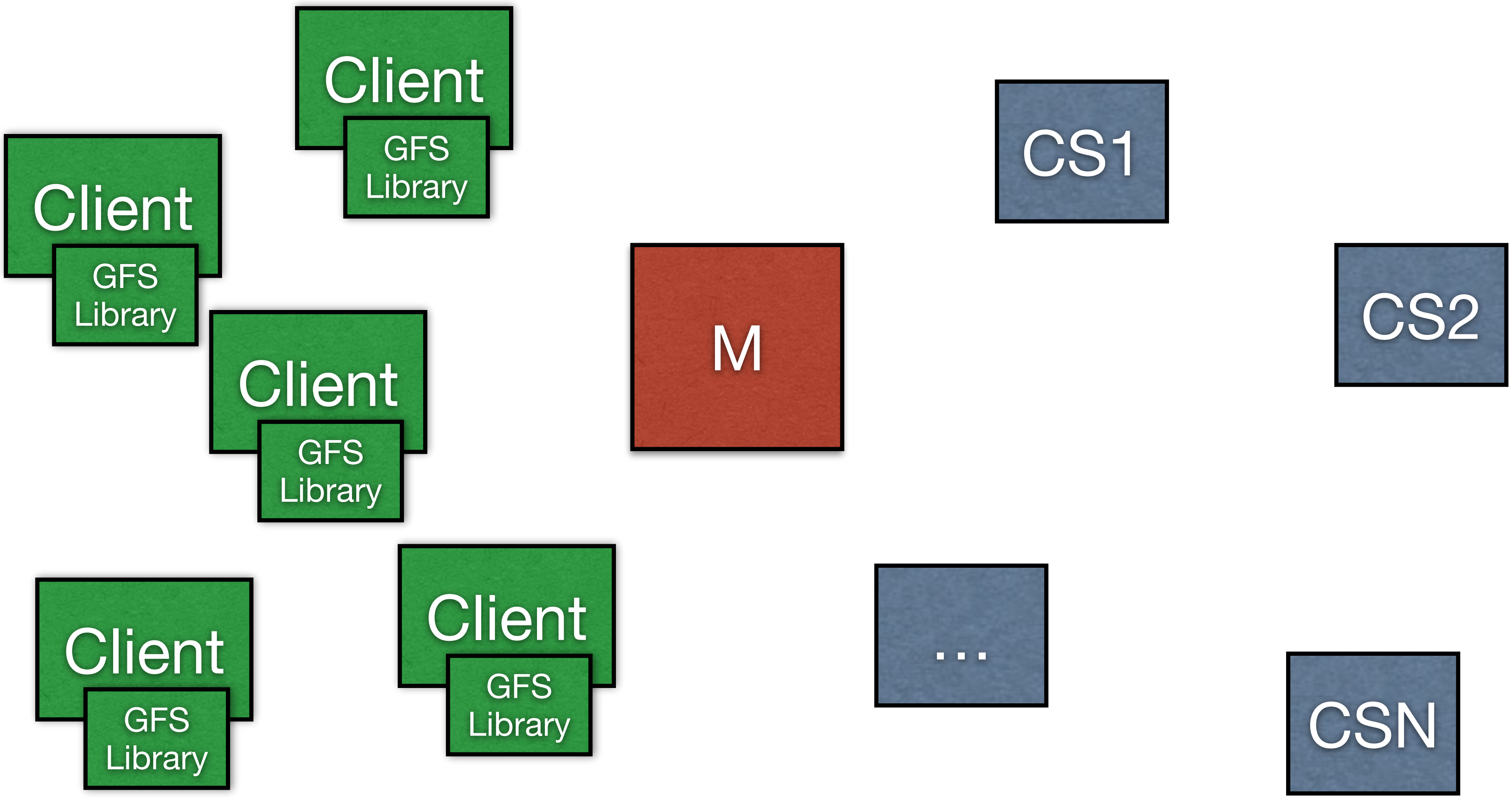
Single Master Node

Master maintains all FS metadata

- Namespace
- Access control
- File -> chunk mappings
- Chunk locations

Master controls all system-wide activities

- Garbage collection
- Lease management
- Chunk migration (balancing)
- Heartbeat messages
  - Periodic master <-> chunk server messages to give instructions / collect state


Client
GFS
Library
Client
GFS
Library
Client
GFS
Library
Client
GFS
Library
Client
GFS
Library
M
CS1
CS2
...
CSN

Avoiding the Master Bottleneck

Don't want system bottlenecked the master

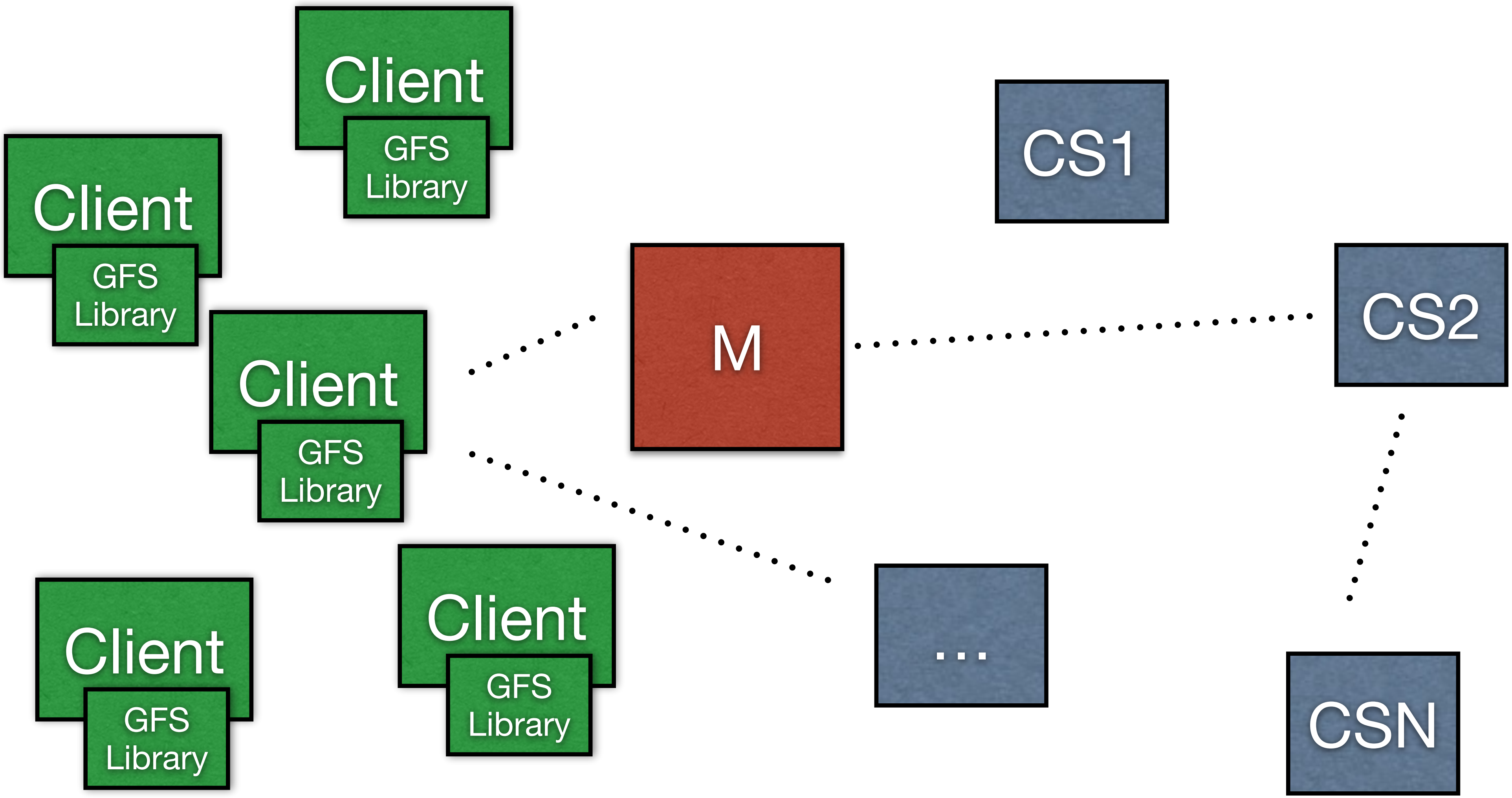
- … so we want to minimize master involvement. How?

Clients

- Clients get all metadata from the master, but interact with chunk servers directly
- Clients do not cache data -> no cache coherency issues

Chunk Servers

- Heartbeats and leases


Client
GFS Library
Client
GFS Library
Client
GFS Library
Client
GFS Library
Client
GFS Library
M
CS1
CS2
...
CSN

Chunks

64 MiB, but stored as a regular file

What “optimizations” for target environment?

- Lazy space allocation
  - only extended when needed, so no internal fragmentation
- Big chunks mean that even for large files, very few chunk indices must be cached by clients
  - However “hot spots” can show up for popular chunks

Remember, all chunks are regular files, so local FS’s optimizations and drawbacks apply

Managing Metadata

Master keeps several types of metadata

(1) System filenames and chunk namespaces

(2) Index w/ {file -> {chunks}} **(like "recipes" in dedup)**

(3) Locations of chunk replicas

How?

- (1) and (2) kept in operation log persistently
- (3) queried by master at startup, maintained with heartbeat messages

Operation Log

The Operation Log keeps the only persistent record of metadata

- Files and chunks are versioned using the timestamps in the operation log
- The operation log is replicated on multiple machines
  - GFS does not respond to a client operation until the operation log entry is flushed locally *and* remotely
- GFS can recover file system state by replaying the log
  - Takes periodic checkpoints to keep the log small
    - Flush all pending operations
    - Clear the consistent log prefix

Consistency Model

Metadata is handled exclusively by the master, so namespace mutations are atomic (e.g., file create)

A file region is consistent when

- no matter which replica a client reads from, same data returned

File data mutations can be writes or record appends

- On record append, data is appended atomically and *at least* once, at an offset of GFS's choosing
- To deal with padding and duplicates, applications should build in checksums or another method of writing self-validating data

GFS applies mutations to chunks in the same order at all replicas, and uses version numbers to detect stale chunks

Leases

For a given chunk, master grants a lease to one of the replicas

This primary replica chooses the mutation ordering

- All other replicas perform mutations in that order

This delegation of work keeps some of the management overhead off of the master

Snapshots

Snapshot goal: create a copy of a file or directory tree at low cost

Snapshot operation steps:

- Master revokes all outstanding leases on all chunks that comprise the “to-be-snapshotted” files
- Master adds snapshot operation to operation log
- Master duplicates the metadata
  - Reference count is now >1 for all chunks in to-be-snapshotted files

When a new operation is requested, reference count >1 so copy-on-write techniques are used

Garbage Collection

Space is not reclaimed immediately

- Deleted files are renamed to a hidden name that includes a deletion timestamp
- During regular FS scan, reclaim space from deleted files older than some threshold (e.g., 3 days)
  - Delayed reclamation prevents accidental deletion

Stale replicas are also deleted during garbage collection

- A replica is stale if its version number is not up-to-date with current lease's version number

Big Picture Lessons

Tradeoff of generality and performance

- Don't need POSIX, can rethink with application in mind

Don't hide failures from the application

- Design sensible abstractions to tolerate common failure modes
- Give applications easy-to-reason-about models

Think back to LFS motivations

- What trends motivated LFS? Still true?
- Compare to motivation for GFS.
  - How are they different? The same?