CS x55: Distributed Systems
[The Google File System]

Chunks and Memory

A file has many a chunk
  all the same size, of course
Keep track of these chunks
  not just the *where*,
  but also the *how many*.

Keep this all in memory
  at the controller
To then decide
  who goes *where*
  or if you need to rebalance.

The data may roam *free*
  but the controller always keeps score.

Shrideep Pallickara
Computer Science
Colorado State University

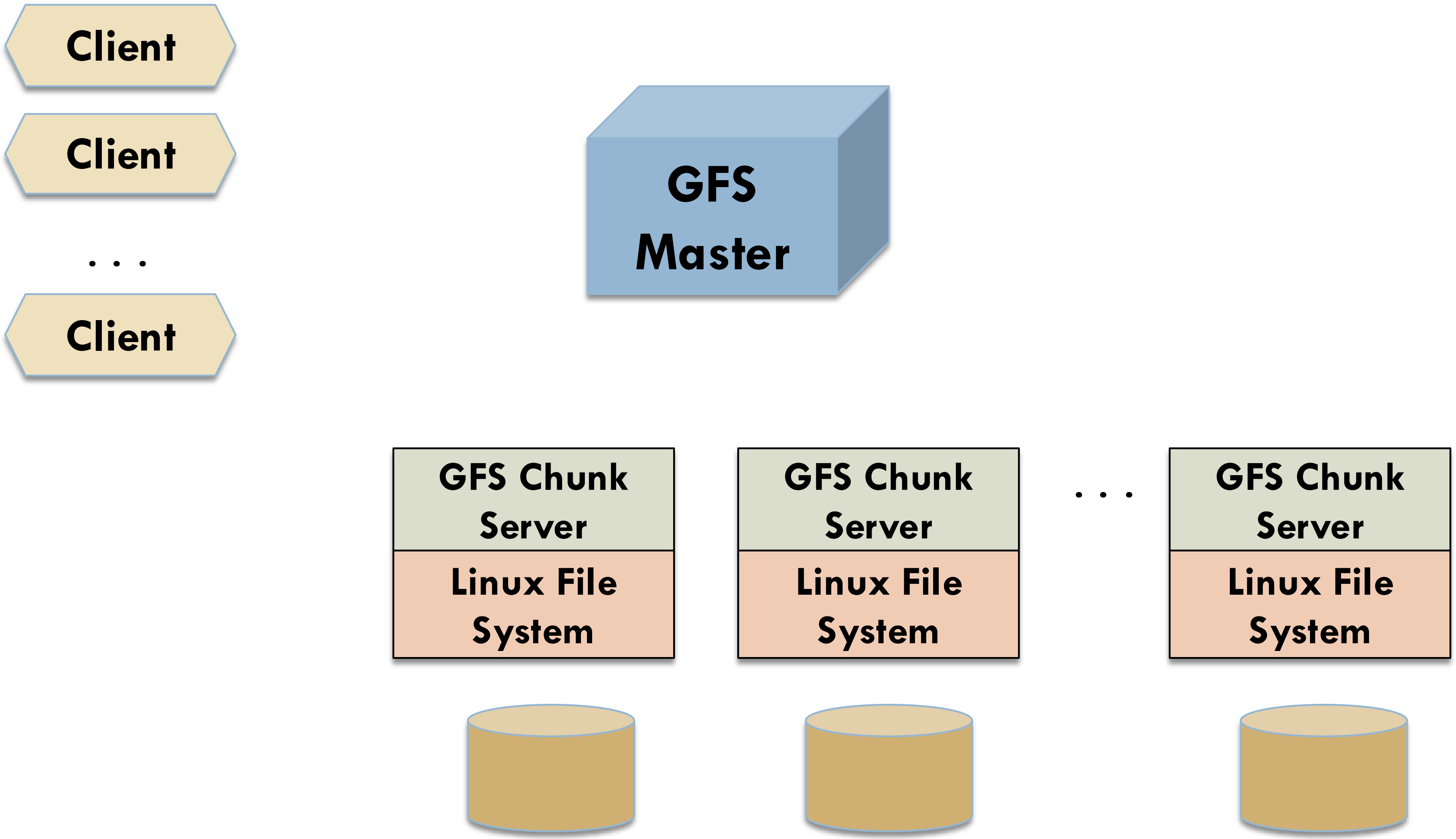
Frequently asked questions from the previous class survey

- Does the O(N) routing table pose issues for Dynamo's scalability?
- What's the preferred file system for systems such as Dynamo or GFS?
- Does the GFS master node have a disk?

Topics covered in this lecture

- The Google File System

Architecture of GFS



In GFS a file is broken up into fixed-size chunks

- Obvious reason
  - The file is too big

Map-Reduce ↓

- **Set the stage** for computations that operate on this data
  - Parallel I/O
  - I/O seek times are 14 x 10^6 slower than CPU clock cycles

In GFS a file is broken up into fixed-size chunks

- Each chunk has a 64-bit globally unique ID
  - Assigned by the Master
- Chunks are stored by chunk servers
  - On local disks as LINUX files
- Each chunk is **replicated**
  - Default is 3

Master operations

- Manage system **metadata**
- **Leasing** of chunks
- **Garbage collection** of orphaned chunks
- Chunk **migrations**

ALL system metadata is managed by the Master and stored in **main memory**

1. File and chunk namespaces
2. Mapping from files to chunks
3. Location of chunks

} *Logs mutations into a permanent log*

Why have a single Master?

- Vastly **simplifies** design
- Easy to use global knowledge to **reason about**
  - Chunk placements
  - Replication decisions

Communications with the chunk servers

- Periodic communications using **heartbeats**
  - Instructions to the chunk server
  - Collect/retrieve state from the chunk server

Chunk size

- This is fixed at **64 MB**
  - Much larger than typical filesystem block sizes (512B up to 4KB)

- **Lazy space allocation**
  - Stored as plain Linux file
  - Extended only as needed

But why this big?

- **Reduces client interaction** with the master
  - Can cache info for a multi-TB working set
- Reduce network **overhead**
  - With a large chunk, client performs more operations
  - Persistent connections
- Reduce **size of metadata** stored in the master
  - 64 bytes of metadata per 64 MB chunk

Why keep the entire metadata in memory?

- **Speed**
- Master can **scan** its **state** in the background
  - Implement chunk garbage collection
  - Re-replicate if there are failures
  - Chunk migration to balance load and space
- **Add** extra memory to increase file system size

Size of the file system with 1 TB of RAM: Assume file sizes are exact multiples of chunk sizes

- Number of entries = $2^{40}/2^{6}$
- MAXIMUM SIZE of the file system

$$= \text{Number of entries} \times \text{Chunk size}$$

$$= \frac{2^{40}}{2^{6}} \times 2^{6} \times 2^{20}$$

$$= 2^{60} = 1 \text{ EB}$$

Tracking the chunk servers

- Master **does not** keep a persistent copy of the location of chunk servers

- List maintained via **heart-beats**
  - Allows list to be in *sync* with reality despite failures
  - Chunk server has final word on chunks it holds

Caching at the client/chunk servers

- Clients **do not cache** file data
  - At client, the working set may be *too large*
  - Simplify client; eliminate *cache-coherence* problems
- Chunk servers **do not cache** file data either
  - Chunks are stored as local files
  - Linux's buffer cache *already keeps* frequently accessed data in memory

Mutation: it is the key to our evolution.
Charles Xavier, X-Men.

MANAGING MUTATIONS

Mutations

- **Mutation** changes the content and/or metadata of a chunk
  - Write
  - Append

- Each mutation is performed at **all** chunk replicas

GFS uses leases to maintain consistent mutation order across replicas

- Master grants **lease** to one of the replicas
  - **PRIMARY**

- Primary picks **serial-order**
  - For all mutations to the chunk
  - Other replicas *follow* this order
    - When applying mutations

Lease mechanism designed to minimize communications with the master

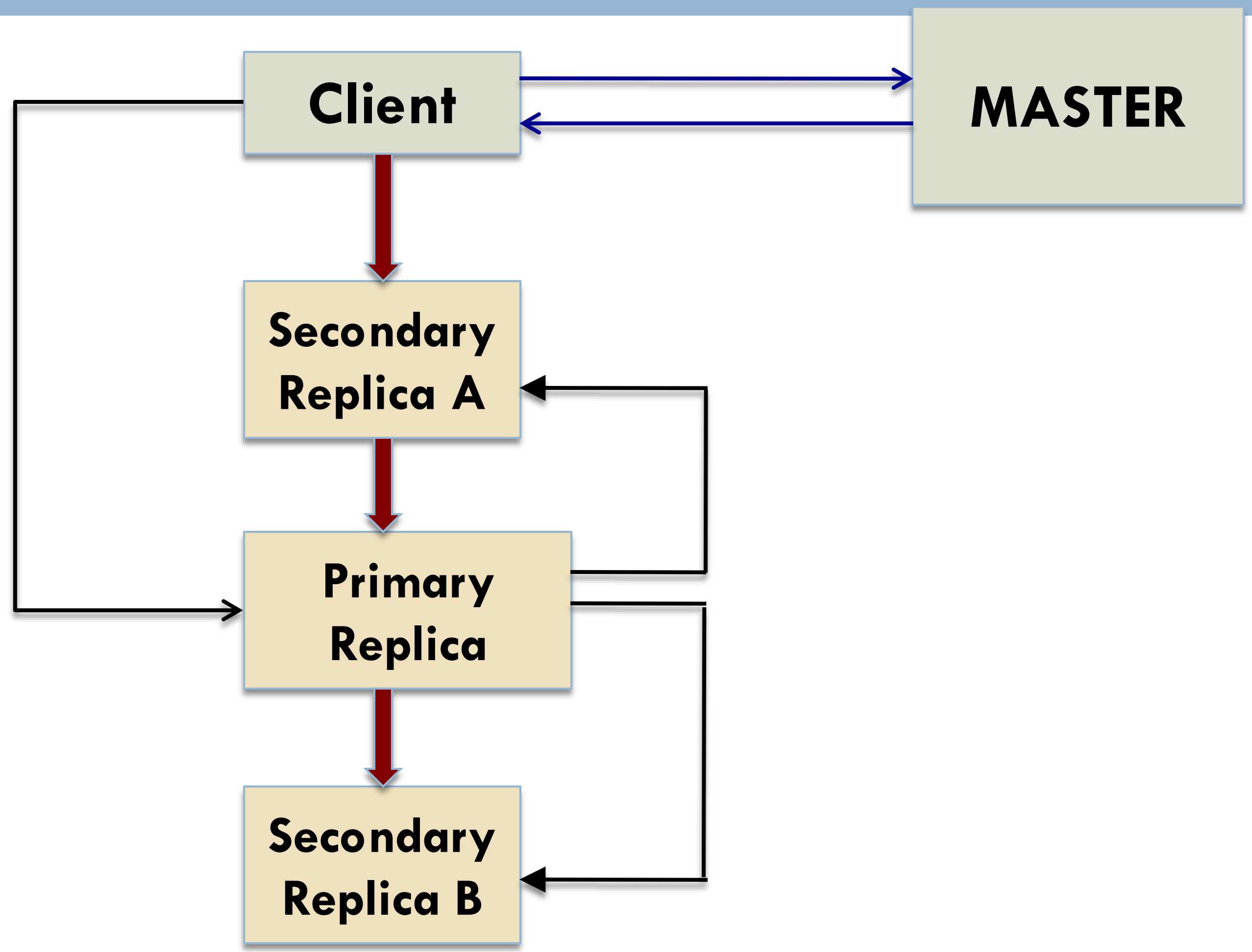
- Lease has initial *timeout* of 60 seconds
- As long as chunk is being mutated
  - Primary can request and receive **extensions**
- Extension requests/grants piggybacked over heart-beat messages

Revocation and transfer of leases

- Master may **revoke** a lease before it expires

- If communications lost with primary
  - Master can safely give lease to another replica
    - **ONLY AFTER** the lease period for old primary *elapses*

How a write is actually performed



Client orchestrates writing data to the replicas [1/2]

- Each chunk server stores data in an **LRU buffer** until
  - Data is used
  - Aged out

Client orchestrates writing data to the replicas [2/2]

- When chunk servers acknowledge receipt of data
  - Client sends a *write request to primary*
- Primary assigns *consecutive serial numbers* to mutations
  - Forwards to replicas

Data flow is **decoupled** from the control flow to utilize network efficiently

- Utilize each machine’s network bandwidth
- Avoid network bottlenecks
- Avoid high-latency links
- **Leverage** network topology
  - Estimate distances from IP addresses

What if the secondary replicas could not finish the write operation?

- Client request is considered **failed**
- Modified region is **inconsistent**
  - *No attempt* to delete this from the chunk
  - Client must handle this inconsistency
- Client **retries** the failed mutation

GFS client code implements the file system API

- Communications with master and chunk servers done *transparently*
  - On behalf of apps that read or write data
- Interact with master for **metadata**
- **Data-bearing** communications directly to chunk servers

Traditional writes

- Client specifies **offset** at which data needs to be written
- Concurrent writes to same region
  - Not serializable
  - Region ends up containing **data fragments from multiple clients**

Atomic record appends

- Client specifies **only** the data **not** the offset
- GFS **appends** it to the file
  - **At least once atomically**
  - At an offset of GFS' choosing
- No need for a distributed lock manger

The control flow for record appends is similar to that of writes

- Client pushes data to replicas of the ***last*** chunk of file
- Primary replica checks if the record **fits** in this chunk

Primary replica checks if the record append will breach the size (64MB) threshold

- If chunk size would be breached
  - **Pad** the chunk to maximum size
  - Tell client, that operation should be retried on *next chunk*

- If the record fits, the primary
  - Appends data to its replica
  - Notifies secondaries to **write at the exact offset**

Record sizes and fragmentation

- Size is restricted to ¼ the chunk size
- Minimizes worst-case fragmentation
  - Internal fragmentation in each chunk …

What if record append fails at one of the replicas

- Client **must retry** the operation
- Replicas of same chunk may contain
  - Different data
  - **Duplicates** of the same record
    - In whole or in part
- Replicas of chunks are **not bit-wise identical**!
  - In most systems, replicas are identical

GFS only guarantees that the data will be written **at least once** as an atomic unit

- For an operation to return *success*
  - Data must be **written at the same offset** on **all** the replicas
- After the write, all replicas are *as long as* the end of the record
  - Any future record will be assigned a higher offset or a different chunk

REPLICATION

Reasons why chunk replicas are created

- Chunk creation
- Re-replication
- Rebalancing

Chunk replica creation

- Place replicas on chunk servers with below average **disk space** utilization
- **Limit** the number of **recent creations** on a chunk server
  - Predictor of imminent heavy traffic
- Spread replicas **across racks**

Re-replicate chunks when replication level drops

- How **far** is it from replication goal?
- Preference for chunks of **live** files
- Boost priority of chunks **blocking client progress**

Rebalancing replicas

- **Examine** current replica distribution and **move** replicas
  - Better disk space
  - Load balancing

- **Removal** of existing replicas
  - Chunk servers with below-average disk space

Incorporating a new chunk server

- **Do not swamp** new server with lots of chunks
  - Concomitant traffic will bog down the machine
- **Gradually** fill up new server with chunks

So, so you think you can tell
Heaven from hell?
Blue skies from pain?
Can you tell a green field
From a cold steel rail?
A smile from a veil?
Do you think you can tell?

Wish You Were Here; Gilmour/Waters; Pink Floyd

CREATING SNAPSHOTS

Snapshots allow you to make a copy of a file very fast

- Master **revoke**s outstanding leases for any chunks of the file (source) to be snapshot
- **Log** the operation to disk
- Update in-memory state
  - Duplicate metadata of the source file
- Newly created file points to the same chunks as the source

When a client wants to write to a chunk C after the snapshot operation

- Master sees the *reference count* to **C > 1**
- Pick **new chunk-handle C'**
- Ask chunk-server with current replica of **C**
  - Create new chunk **C'**
  - Data is *copied locally, not over the network*
- From this point chunk handling of **C'** is no different

GFS does not have a per-directory structure that lists files in the directory

- Name spaces represented as a **lookup** table
  - Maps **full pathnames** to metadata

- File creation does not require a lock on the directory structure
  - No inode needs to be protected from modification

Each master operation acquires a set of locks before it runs

- If operation involves `/d1/d2/…/dn/leaf`
  - Acquire read locks on directory names
    - `/d1, /d1/d2, …, /d1/d2/…/dn`
  - Read or write lock on full pathname
    - `/d1/d2/…/dn/leaf`
- Used to **prevent** operations during snapshots

Locks are used to prevent operations during snapshots

- For e.g., cannot create `/home/user/foo`
  - While `/home/user` is being snapshotted to `/save/user`
- Read locks on `/home` and `/save`
  - **Read lock prevents a directory from being deleted**

- Write lock on **`/home/user`** and `/save/user`
- File creation does not require write lock on parent directory … there is no "directory"
  - Read locks on `/home` and **`/home/user`**
  - Write lock on `/home/user/foo`

The contents of this slide-set are based on the following references

- Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung: The Google file system. Proceedings of SOSP 2003: 29-43.