Cloud background

- Warehouse scale systems
  - 10K-100K nodes
  - 50MW (1 MW = 1,000 houses)
  - Power efficient
    - Located near cheap power
    - Passive cooling
    - Power Usage Effectiveness = Total facility power/IT equipment power: 1.2 or better
  - CPU, storage network, power:
    - With 10K-100K-node data center 3-5x cheaper than for 100-1K-node data center

Google File System

- Design constraints
  - Component failures are the norm
    - 1000s of components
    - Bugs, human errors, failures of memory, disk, connectors, networking, and power supplies
    - Monitoring, error detection, fault tolerance, automatic recovery
  - Files are huge by traditional standards
    - Multi-GB files are common
    - Billions of objects

Introduction

- Design constraints
  - Most modifications are appends
    - Random writes are practically nonexistent
    - Many files are written once, and read sequentially
  - Two types of reads
    - Large streaming reads
    - Small random reads (in the forward direction)
  - Sustained bandwidth more important than latency
  - File system APIs are open to changes

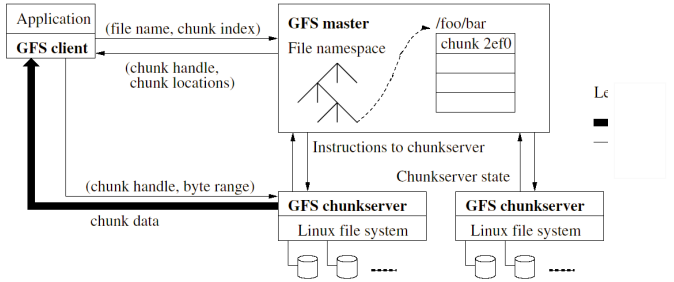
Interface Design

- POSIX-like
  - create, delete, open, close, read, write
- Additional operations
  - Snapshot
  - Record append
    - Supports concurrent append
    - Guarantees atomicity of each append

Architectural Design

- A GFS cluster
  - A single *master*
  - Multiple *chunkservers* per master
    - Accessed by multiple *clients*
  - Running on commodity Linux machines
- A file
  - Represented as fixed-sized *chunks*
    - Labeled with 64-bit unique global IDs
    - Divided in chunks (3-way replicated), stored at chunkservers

Architectural Design (2)

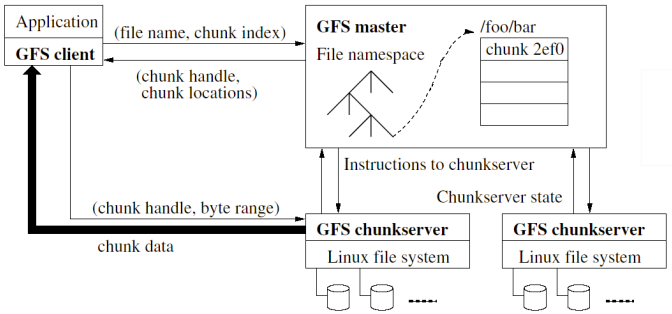


- Single master, multiple chunkservers, multiple clients.
- Files divided into fixed-size chunks.
  - Each chunk identified by immutable and globally unique chunk handle. (*How to create?*)
  - Stored by chunkservers locally as regular files.
  - Each chunk is replicated.
- Master maintains all metadata.
  - Namespaces
  - Access control (*What does this say about security?*)
  - Heartbeat
- No client side caching because streaming access. What about server side?

Single Master

- General disadvantages for distributed systems:
  - Single point of failure
  - Bottleneck (scalability)
- Solution?
  - Clients use Master only for metadata
    - Reading/writing goes directly through the chunkservers

Architectural Design (3)



- Client translates file name and byte offset to chunk index.
- Sends request to master.
- Master replies with chunk handle and location of replicas.
- Client caches this info.
- Sends request to a close replica, specifying chunk handle and byte range.
- Requests to master are typically buffered

Chunk Size

- 64 MB: much larger than a normal FS block
  - Fewer chunk location requests to master
  - Fewer metadata entries for a client (or for the Master)to cache in memory
  - Potential issue with fragmentation
- Hotspots: Some files (executable) may be accessed too much
  - Use higher replication factor.
  - Or…

Metadata

- Three major types
  - File and chunk namespaces
  - File-to-chunk mappings
  - Locations of chunk replicas
- All metadata is in memory
  - System capacity limited by memory of master
    - Memory is cheap
  - On a failure
    - Name space and mapping recovered through an operations log (kept on disk and replicated remotely)
    - Location info rebuilt by querying chunkservers upon recovery

Metadata: Chunk Location

- Not kept persistent at the Master
  - Chunkservers polled at startup
  - Info periodically refreshed using heartbeat messages to chunkservers
    - Possibly triggers re-replication
- Simplicity
  - No need to sync info at Master as nodes leave (voluntarily or because of a failure)
  - With many failures, hard to keep consistency anyway

Metadata: Operation Log

- Serves as logical timeline for when files and chunks are created
- Keeps
  - File/chunk namespaces
  - File to chunk mapping
- Replicated on remote machines
- Checkpointed periodically to truncate logs and bound startup

Consistency Model

- File namespace mutations occur atomically.
  - Relatively simple, since just a single master.
- A file region is
  - *Consistent* if all clients see the same data, independent of the replica they access.
  - *Defined* if consistent and all clients see the modification in its entirety (as if change were atomic)

	Write	Record Append
Serial success	Defined	Defined, but interspersed with inconsistent
Concurrent success	Consistent but undefined	
Failure	Inconsistent	

Consistency Model

- Two types of updates
  - Writes (at application-specified offset)
  - Record appends (at offset of GFS's choosing)
- Relaxed consistency
  - Concurrent changes are consistent but undefined
  - A record append is atomically committed at least once
    - Even during concurrent updates
    - Offset returned to the client marks beginning of defined region
    - Occasional padding/duplications
- Updates are applied in the same order at all replicas
  - Use chunk version number to detect stale replicas
    - They are garbage collected ASAP
- Client read may temporarily see stale values
  - Staleness limited by timeout on cache entry

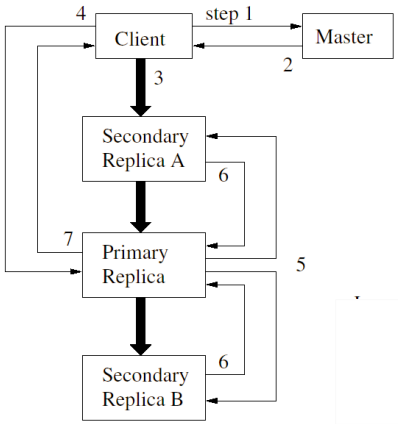
Implications for applications

- Use appends rather than overwriting
- Checkpoints to mark defined portions of files
- Self-validating/self-identifying records
  - Use checksums to throw away extra padding and record fragments
- For non-idempotent operations, use unique record identifier to handle duplicates

Leases and mutation order

- Master grants chunk lease to *primary* replica
- Primary determines order of updates to all replicas
- Lease:
  - 60 second timeouts
  - Can be extended indefinitely
  - Extension request are piggybacked on heartbeat messages
  - Can be revoked (to achieve copy-on-write for snapshots)

The leasing mechanism



1. Client asks master for all replicas and id of primary
2. Master replies. Client caches.
3. Client pre-pushes data to all replicas.
4. After all replicas acknowledge, client sends write request to primary.
5. Primary forwards write request to all replicas.
6. Secondaries signal completion.
7. Primary replies to client. Errors handled by retrying.

- If write straddles chunks, broken down into multiple writes, which causes undefined states.

Data Flow

- Separation of control and data flows
  - Control goes from client to primary to all replicas
  - Data is spread using chain replication
    - Push data to closest chunkserver
- Transfers are pipelined
  - Chunkserver forwards data as soon as it receives it

Atomic Record Appends

1. Client pushes data to all replicas.
2. Sends request to primary.
3. If record does not fit in chunk, primary pads current chunk and tells client to retry with new chunk.
4. Primary writes data, tells replicas to do the same at its chosen offset.

- If any replica fails client retries
  - Different replicas may end up storing different sequences of bits
  - On success, data written everywhere at the same offset
    - Clients need to handle inconsistencies (see before)

Snapshot

- Used to
  - Create copies of large data sets
  - Have something to go back to quickly before applying tentative updates
- Handled using copy-on-write
  - Revoke outstanding leases
  - Log operation to disk
  - Duplicate the metadata, but point to the same chunks.
  - When a client requests a write, the master allocates a new chunk handle
    - To save bandwidth, new chunk copy is located on the same chunkserver as the old one

Master Operation

- No directories
- No hard links and symbolic links
- Logically, namespace represented as a lookup table
  - Maps full path name to metadata

Locking Operations

- A lock per path
  - To access /d1/d2/leaf
  - Need to lock /d1, /d1/d2, and /d1/d2/leaf
  - Can modify a directory concurrently
    - Each thread acquires
      - A read lock on a directory
      - A write lock on a file
  - Totally ordered locking to prevent deadlocks

Replica Placement

- Goals:
  - Maximize data reliability and availability
    - Spread chunk replicas across machines and racks
  - Maximize network bandwidth
- Creation
  - Prefer underutilized chunkservers
  - Limit new chunks per chunkservers (why?)
- Re-replication
  - Prioritize chunks with lower replication factors
- Rebalancing
  - Periodically move replicas around for better disk and load balancing

Garbage Collection

- Used to implement lazy file deletion
  - Simple
  - Works despite master or replica failure, message loss, etc
- Deleted files are first “hidden” for three days
- After 3 days, in-memory metadata of hidden file is removed
- Orphaned chunks detected and deleted during regular scan

Stale replica detection

- Chunk version number
  - Increased on every new lease
  - Logged by master and replicas in persistent state
  - Relayed to client when contact Master for chunk handle
- Stale replicas detected by Master when chunkserver recovers

Fault Tolerance and Diagnosis

- Fast recovery
  - Master and chunkserver are designed to restore their states and start in seconds regardless of termination conditions
- Chunk replication
  - Moving towards erasure coding
- Master replication
  - Shadow masters provide read-only access when the primary master is down

Fault Tolerance and Diagnosis

- Data integrity
  - A chunk is divided into 64-KB blocks
  - Each with its checksum (logged persistently)
  - Verified at read and write times
  - Also background scans for rarely used data