Advanced Operating Systems - Google File System

Mridul Sankar Barik

Jadavpur University

2024

Google File System

- Scalable distributed file system
  - Supports data intensive applications (hundreds of terabytes of storage)
  - Fault tolerant (runs on thousands of inexpensive commodity hardware)
  - Delivers High performance
  - Handles large no. of clients

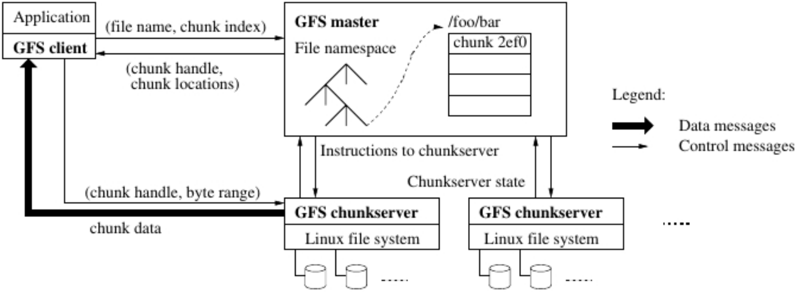
GFS Dsign: Assumptions

- System is built from many inexpensive commodity hardware
- Files are large (multi GB files are common)
- Workloads
  - Two types of read: large streaming reads (100's of KBs, 1 MB or more), small random reads (few KBs)
  - Large sequential writes (append)
- Well defined semantics for concurrent access
  - Files are often used as producer-consumer queues
  - Producers may concurrently update
  - Consumers may read later or simultaneously

GFS Dsign: Interface

- Does not implement POSIX API
- Files are organized hierarchically in directories and identified by path names
- Usual file operations: *create, delete, open, close, read, write*
- Other operations:
    - *snapshot*: create a copy of a file or a directory tree
    - *record append*: concurrent append to a file with guaranteed atomicity

GFS Design: Architecture I



- GFS cluster
  - A single *master*
  - Multiple *chunkservers*
  - Multiple *clients*

GFS Design: Architecture II

- Files are divided into fixed size *chunks*; identified by 64-bit globally unique *chunk handle* assigned by the master
- Chunckservers store chunks on local disk as Linux files
- Each chunk is replicated on multiple chunkservers
- Master maintains all file system metadata
  - namespace, access control information, mapping from files to chunks, current location of chunks
- Master periodically communicates with chunkservers via *HeartBeat* messages
- GFS client code linked into applications implement file system API
- Neither chunkserver nor client cache file data

GFS Design: Single Master

- Single master implement chunk placement and replication decisions using global knowledge
- Client read/writes do not involve master
- Example:
  - Using fixed chunk size, client translates file name and byte offset into a chunk index within the file
  - Client sends a request containing the file name and chunk index
  - Master replies with corresponding chunk handle and locations of the replicas
  - Client caches this information using file name and chunk index as the key
  - Client sends request to one of the replicas specifying chunk handle and byte range
  - Further reads of the same chunk require no more client-master interaction

GFS Design: Chunk Size

- Chunk size = 64 MB
- Advantages of large chunk size
  - Reduces client-master interactions
  - As a client is more likely to perform many operations on a given chunk, it reduces network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time
  - Reduces size of metadata on the master
- Disadvantages of large chunk size
  - Chunkservers storing chunks of a small file may become hotspots if many clients simultaneously access the same file
  - Solution: increase replication factor for such files

GFS: Meta Data I

- Three major types of metadata
  - file and chunk namespaces
  - mapping from files to chunks
  - loctaion of each chunk replicas
- All metadata is kept in master's memory
- First two types are kept persistent by logging changes to an *operation log* stored in master's local disk and replicated on remote machines

GFS: Meta Data II

- Chunk locations
  - Master never put chunk location information in persistent storage
  - Master asks each chunkserver about its chunks at startup and whenever a chunkserver joins the cluster and periodically thereafter via *HeartBeat* messages
- Operation logs
  - Contains a historical record of critical metadata changes
  - Log is replicated on multiple remote machines by flushing them in batches

GFS: Consistency Model I

- File namespace mutation
  - Atomic: handled exclusively by the master using namespace locking
  - Master's operation log defines a global total order of these operations
- Data mutation
  - A file region is *consistent* if all clients always see the same data, regardless of the replica
  - A file region is *defined* after a file data mutation if it is consistent and clients see what the mutation writes in its entirety
  - Single successful mutation (no concurrent writes) $\Rightarrow$ defined and hence consistent
  - Concurrent successful mutation $\Rightarrow$ undefined but consistent
  - Failed mutation $\Rightarrow$ inconsistent and undefined

GFS: System Interactions I

1. Leases and Mutation Order

- Mutation: operations (write/append) that change content or metadata of a chunk and must be performed at all the replicas
- Chunk lease: granted by master to one of the the replicas (primary) which picks a serial order for all mutations to the chunk

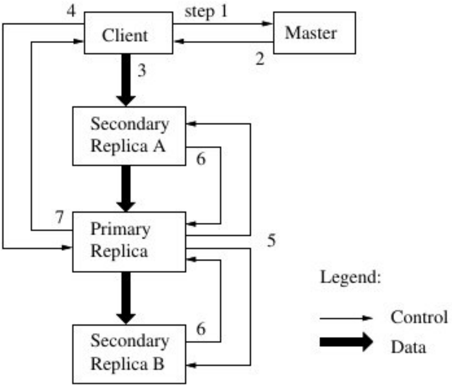


Figure 1: Write Control and Data Flow

GFS: System Interactions II

2. Data Flow

- Data is pushed linearly along a chain of chunkservers in a pipelined fashion

3. Atomic Record Appends

- GFS appends client data to the file atomically at an offset of GFS's choosing and returns the offset to the client

GFS: System Interactions III

4. Snapshot

- Copy a file or directory tree instantaneously
- Objective: to make copies of huge data sets or to checkpoint current state
- Implementation:
  - Master revokes leases on chunks in the files
  - Master logs the operation on disk; applies this log to in-memory state by duplicating metadata
  - Newly created snapshot points to the same chunks as the source
  - First write to a chunk C after snapshot $\Rightarrow$ Master creates a new chunk handle C' and asks each chunkserver that has a current replica of C to create a new chunk C' $\Rightarrow$ Master grants one of the replicas a lease on the new chunk C'

GFS: Master Operation I

- Namespace Management and Locking
  - Multiple operations may be active simultaneously by using locks over regions of name space
  - GFS does not have per-directory data structure
  - GFS logically represents its name space as a look up table mapping full path names to meta data
  - Each node in the name space tree has an associated read-write lock
  - Each master operation acquires a set of locks before it runs
  - Example: to access `/d1/d2/.../dn/leaf` it acquires read locks on directory names `/d1, /d1/d2, ..., /d1/d2/.../dn` and either a read lock or write lock on `/d1/d2/.../dn/leaf`
  - Example: Two operations; (i) create a file `/home/user/foo` (ii) snapshot of **/home/user** to `/save/user`
    - The file creation acquires read lock on `/home` and `/home/user` and write lock on `/home/user/foo`.
    - The snapshot operation acquires read locks on `/home` and `/save` and write locks on `/home/user` and `/save/user`

GFS: Master Operation II

    - Two operations will be serialized because they try to obtain conflicting locks on `/home/user`

- Replica Placement
  - Hundreds of chunk servers spread across many machine racks
  - Communication between two machines on different racks may cross one or more network switches
  - Policy: distribute chunks across machines / racks
- Creation, Re-replication, Re-balancing
  - Chunk creation: where to place when Master creates a chunk
    - on chunk servers with below average disk space utilization
    - to limit no. of recent creations on each chunk servers
    - across racks
  - Chunk re-replication: occurs when replication level falls; Policy
    - re-replicate a chunk which is farthest from its replication level
    - re-replicate chunks belonging to live file vs. recently deleted files
  - Chunk re-balancing: periodically done by master for better disk space and load balancing

GFS: Master Operation III

- Garbage Collection
  - After a file is deleted GFS reclaims physical storage lazily during periodic garbage collection
  - Mechanism:
    - Master logs deletion immediately and renames it to a hidden name with deletion time stamp
    - During master's regular scan, hidden files are removed if they have existed more than three days $\Rightarrow$ until then, they can be un-deleted
    - Orphaned chunks (not reachable from any file): removed during master's regular scan
      In regular HeartBeat messages with master a chunkserver reports a subset of chunks it has and the master replies with identity of chunks that are no longer present in master's meta data
      Reason for creation of orphaned chunks: A chunkserver might have been down when a file was deleted and then it came up
- Stale Replica Detection
  - A chunk becomes stale if chunk server misses mutation while it is down
  - Master maintains chunk version number

GFS: Fault Tolerance and Diagnosis I

- Component failures (machines/disks) may result in unavailable system or corrupted data
- High Availability
  - Fast Recovery: Master and chunk server can restore state and start very quickly
  - Chunk Replication
    - User can specify different replication level for different parts of file name space
    - Master clones existing replicas as chunk servers go offline or detect corrupted replicas
  - Master Replication
    - Master state (operation logs and checkpoints) is replicated on multiple machines
    - When Master fails, monitoring infrastructure (out side GFS) starts Master elsewhere

GFS: Fault Tolerance and Diagnosis II

- Data Integrity
  - A chunk is broken into 64 KB blocks, each has a corresponding 32-bit checksum
  - For reads, chunk server verifies the checksum of data blocks that overlap the read range and returns data
  - If mismatch, chunk server reports error to requester and to the Master $\Rightarrow$ Master clone the chunk from another replica
  - For writes (append to the end), chunk server incrementally updates checksum of last partial checksum block and compute new checksums for any new blocks
  - For writes (overwrite on range of blocks), chunk server read and verify first and last blocks of the range and then write