Google File System

By Dinesh Amatya

Google File System (GFS)

Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung

- designed and implemented to meet rapidly growing demand of Google's data processing need
- a scalable distributed file system for large distributed data-intensive applications
- provides fault tolerance while running on inexpensive commodity hardware
- Performance , Reliability , Availability , scalability

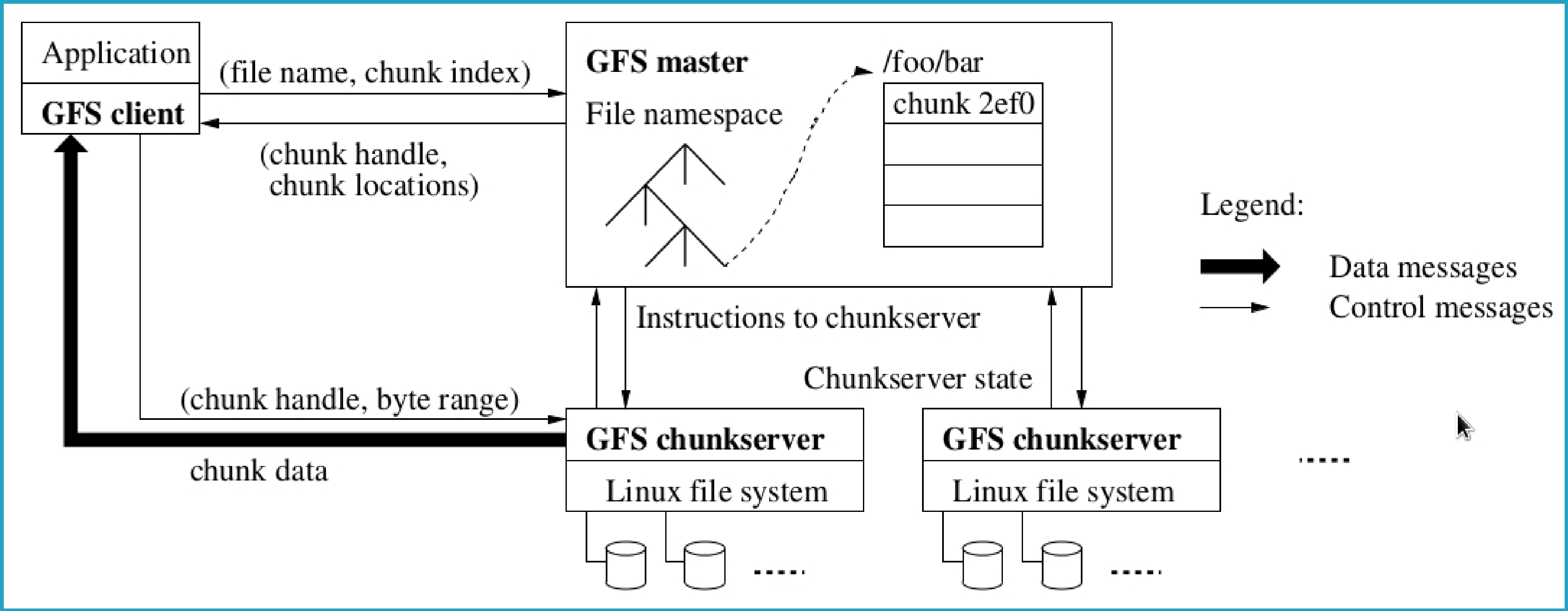
GFS Design Overview: Assumptions

- Component failure is norm : System components such as disks, machines and links are ought to fail. So, the system should be able to detect and recover from the failure by constantly monitoring
- File are huge : System stores millions of files most of which are multi-GB. So, large files must be stored efficiently.
- Most modifications are appends: Random writes are practically non-existent. Once written, files are seldom modified again and are read sequentially.
- Two types of read: Large streaming reads, Small random reads
- must efficiently implement well-defined semantics for multiple clients that concurrently append to the same file
- Sustained bandwidth more important than low latency

GFS Design Overview: Interface

- Provides familiar file system interface
  - Create, Delete, Open, Close, Read, Write
- Files are organized hierarchically in directories and identified by pathnames
- GFS has snapshot and record append operations
- Snapshot creates a copy of a file or a directory tree at low cost
- Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client's append

GFS Design Overview: Architecture



GFS Design Overview: Architecture

→ Single Master

→ Multiple Chunkserver

→ Multiple Clients

→ Files are divided into Chunks

→ Chunk handle

→ Master

- maintains Metadata (namespace, access control information,mapping from file to chunks, current location of chunks)
- controls activities such as chunk lease management, garbage collection of orphaned chunks, and chunk migration between chunkservers.
- communicates with each chunkserver in HeartBeat messages to give it instructions and collect its state

GFS Design Overview: Architecture

→ GFS client code linked into each application implements the file system API and communicates with the master and chunkservers to read or write data on behalf of the application

→Clients interact with the master for metadata operations, but all data-bearing communication goes directly to the chunkservers

→ Neither the client nor the chunkserver caches file data

GFS Design Overview: Single Master

→ vastly simplifies the design

→ enables the master to make sophisticated chunk placement and replication decisions using global knowledge

→ However, one must minimize its involvement in reads and writes so that it does not become a bottleneck

→ Clients never read and write file data through the master. Instead, a client asks the master which chunkservers it should contact. It caches this information for a limited time and interacts with the chunkservers directly for many subsequent operations.

GFS Design Overview: Chunk Size

→ 64 MB – much larger than ordinary,

Why ?

- Advantages
  - . Reduce client-master interaction
  - . Reduce network overhead
  - . Reduce the size of metadata
- Disadvantage
  - . Hot spots – many clients accessing a 1-chunk file

GFS Design Overview: Metadata

→ Three major types

- File and chunk namespaces
- File to chunk mapping
- Location of chunk replicas

→ All kept in memory

- Fast
- Quick global scan (Garbage collection , Reorganization)
- 64 bytes per 64 MB of data

GFS Design Overview: Metadata

→ Chunk location

- obtained in master's memory by polling chunkservers at startup
- updated using heartbeat messages

→ Operation Log

- contains historical record of metadata changes
- only persistent record of metadata
- master recovers its file system by replaying this log
- is critical, hence replicated

GFS Design Overview: Consistency Model

→ A file region can be:

+ consistent: if clients see same data regardless of which replica they read from
+ defined: consistent, when a mutation succeeds without interference from concurrent writers
+ undifined :Concurrent successful Mutations
+ inconsistent: failed mutation

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

Table 1: File Region State After Mutation

GFS Design Overview: Consistency Model

→ After a sequence of successful mutations, the mutated file region is guaranteed to be defined

→ GFS achieves this by

- applying mutations to a chunk in the same order on all its replicas
- using chunk version numbers to detect any replica that has become stale because it has missed mutations while its chunkserver was down

GFS Design Overview: System Interaction

- → Leases and mutation order
- → Data flow
- → Atomic record appends
- → Snapshot

System Interaction : Lease

- → Mutation: operation that changes the contents or metadata
- → Leases to maintain a consistent mutation order across replicas
- → Designed to minimize management overhead of master
- → Master grants lease to one replica which is called primary
- → Primary picks serial order of mutation and all replicas follow
- → 60 second timeout, can be extended
- → can be revoked

System Interaction : Data Flow

→ Decouples data flow and control flow

→ Control Flow

  - Master → primary → secondary

→ Data Flow

  - Carefully picked chain of chunk servers

    . Forward to the closest first

    . Pipelining to exploit full-duplex links

System Interaction : Data Flow

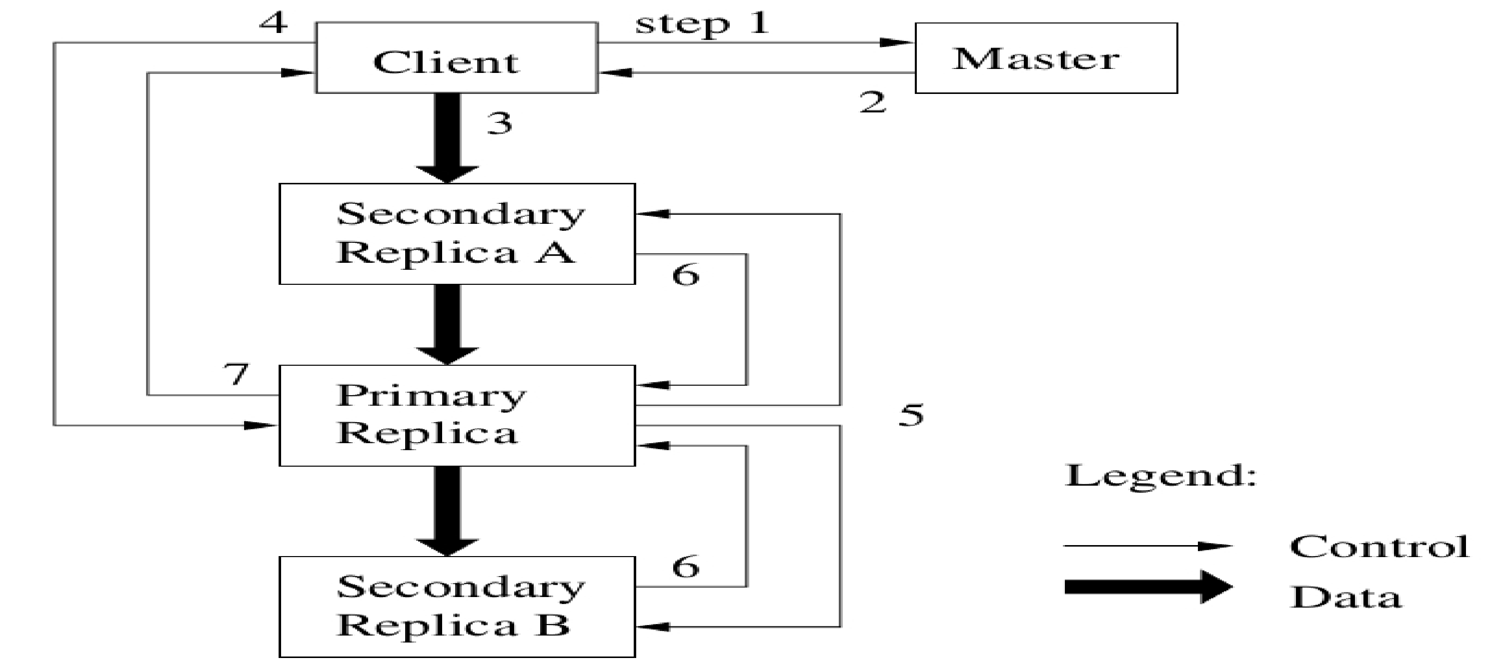


Fig: Write Control and Data Flow

System Interaction : Atomic Record Appends

→ Decouples data flow and control flow

→ Control Flow

- Master → primary → secondary

→ Data Flow

- Carefully picked chain of chunk servers
  - . Forward to the closest first
  - . Pipelining to exploit full-duplex links

→ Master grants lease to one replica which is called primary

→ Primary picks serial order of mutation and all replicas follow

→ 60 second timeout, can be extended

→ can be revoked

System Interaction : Snapshot

→ Makes a copy of file of directory tree almost instantaneously

→ Steps

- Revokes lease
- Logs operation to disk

GFS Design Overview: Master Operation

→ executes all namespace operations
→ manages chunk replicas throughout the system
→ makes placement decisions, create new chunks
→ ensures chunks are fully replicated
→ balances load across all chunkservers
→ reclaim unused storage

Master Operation:
Namespace management and locking

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

→ each node in namespace tree has a read write lock

→ to access /d1/d2/leaf , need to lock /d1 , /d1/d2 and /d1/d2/leaf

→ can modify a directory concurrently. Each thread aquires

- a read lock on directory
- a write lock on a file

Master Operation:
Replica Placement

→hundreds of chunkservers spread across many racks

→two purposes: maximize data reliability and availability, and maximize network bandwidth utilization

→ spread chunk replicas across racks

→ tradeoff ?

Master Operation: Creation, Re-replication, Re-balancing

→factors considered for creating replicas

- place new replicas on chunkserver with below-average disk utilization
- limit the number of “recent” creation on chunkservers
- spread replicas of chunk across racks

→ re-replication priority

- chunk that is blocking client progress
- chunks for live files as opposed to chunks that belongs to recently deleted files
- how far it is from its replication goal

Master Operation:
Creation, Re-replication, Re-balancing

→master picks highest priority chunk and instructs some chunkserver to copy the chunk data directly from an existing valid replica

→examines the current replica distribution and moves replicas for better disk space and load balancing

→ remove replicas on chunkservers with below-average free space

→gradually fills up a new chunkserver rather than instantly swamping it with new chunks

Master Operation:
Garbage Collection

→GFS does not reclaim the available physical space after deletion of a file

→does so lazily during regular garbage collection

→ file is renamed to a hidden name

→ can still be read under the new, special name and can be undeleted by renaming

→ removed during master's regular scan if such file existed for more than 3 days

Master Operation:
Stale Replica Detection

→replicas may become stale if a chunkserver fails and misses mutations to the chunk while it is down

→master maintains a chunk version number to distinguish between up-to-date and stale replicas

GFS Design Overview: Fault Tolerance

→ greatest challenges in designing the system is dealing with frequent component failures

Fault Tolerance:High Availability

- → Fast Recovery
- → Chunk Replication
- → Master Replication

Fault Tolerance:Data Integrity

→ uses checksumming to detect corruption of stored data

→ chunk is broken up into 64KB blocks with corresponding 32 bit checksum

→ chunkserver verifies the checksum before returning data

References

- http://google-file-system.wikispaces.asu.edu/
- http://static.googleusercontent.com/media/research.google.com/en//archive/gfs-sosp2003.p
- http://queue.acm.org/detail.cfm?id=1594206
- http://stackoverflow.com/questions/27864495/google-file-system-consistency-model
- http://pages.cs.wisc.edu/~thanhdo/qual-notes/fs/fs4-gfs.txt