CS 5450

Google File System

slides from Cristina Nita-Rotaru

Google File System

- Scalable, distributed file system for large, distributed, data-intensive applications
  - eg, MapReduce
  - Different assumptions from conventional file systems
  - Implemented as a userspace library
- Fault-tolerant storage that trades consistency for simplicity and performance

Application Characteristics

- Hundreds of clients must perform concurrent (atomic) appends with minimal synchronization
- Sustained bandwidth more important than latency
- Response time for individual read/write not important
- Non-traditional access patterns
- Files are very big, several gigs

Design Choices

- Fault tolerance: constant monitoring, error detection, automatic recovery part of the design
- Designed for big files
  - Need to revisit I/O operations and block sizes
- Non-traditional access patterns
  - Large repositories that must be scanned (archival, streams, intermediate data)
  - Most files are modified by appending rather than overwriting – focus optimization on appending
- Co-design applications and file system: looser consistency

Interface Design

- Non-standard API (not POSIX)
- Supports file/directory hierarchy and the usual {create, delete, open, close, read, write}
- Files identified by path names
- Snapshot: low cost file / directory tree copying
- Concurrent appends, no locks

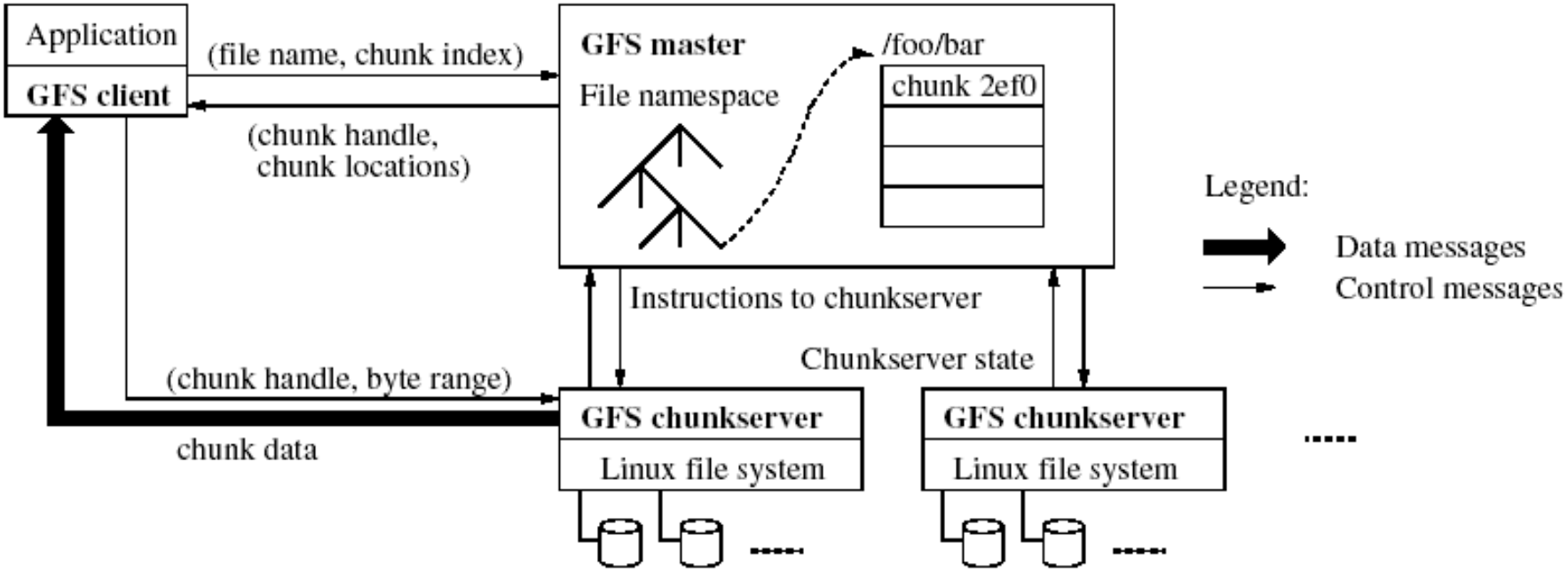
Architecture Overview

- Base unit is chunk
  - Chunk: fixed part of a file (typically 64 MB), globally identified by a unique, 64-bit "chunk handle"
    - Assigned by master server upon chunk creation
  - To read/write, need chunk handle + byte range
  - Each chunk is replicated, minimum three copies
  - Stored as a plain Linux file on a chunkserver
- Servers
  - Single master, keeps metadata
  - Multiple backups (chunkservers)
- Multiple clients

GFS Architecture

Clients cache for a limited time the **chunk handles and locations** so they could interact directly with the chunkservers

Neither clients, nor chunkservers cache file data



Metadata

- Master server keeps metadata
  - File and chunk namespaces, logged to disk with chunk version
  - Mapping from files to chunks, logged to disk with chunk version
  - Locations of chunk replicas (what chunkserver keeps what chunk): master obtains them at startup from each chunkserver, does not keep a persistent record
- … in memory, to speed things up

Why Locations Are Not Persistent

- To keep locations of chunks persistent, the master and chunkservers must be kept in sync as chunkservers join and leave
  - Difficult when there is too much churn
- Chunkserver ultimately knows what chunks it does or does not have on its own disks

Operation Log

- Persistent log of historical changes of metadata
  - Kept small for fast startup, periodic checkpointing
- Defines the order of concurrent operations on the metadata
- Replicated on several remote machines and response goes back to the client only after the corresponding log record was flushed to disk locally and remotely
- Master recovers its state by replaying the log

Consistency Model

- Namespace modifications: only by master server, atomic, in memory, and logged on disk
- Concurrent writes, atomic appends
- Possible chunk states
  - Consistent: all clients see same data, from any replica
  - Defined: consistent and known, i.e., some modification done w/o interruption
  - Undefined: concurrent modifications are successful, replicas consistent but bits mixed from different writes
  - Inconsistent: after any failed modification, inconsistent regions may be padded or contain duplicates

System-Wide Activities

Master server also in charge of ...

- chunk lease management
- garbage collection of orphaned chunks
- chunk migration between chunkservers

Read

- Client translates the filename and offset into a chunk index within a file, sends request to the master
- Master replies with a chunk handle and locations of replicas
  - Client caches this info, does not interact with master on further reads
- Client sends the chunk handle and a byte range within the chunk to a replica chunkserver

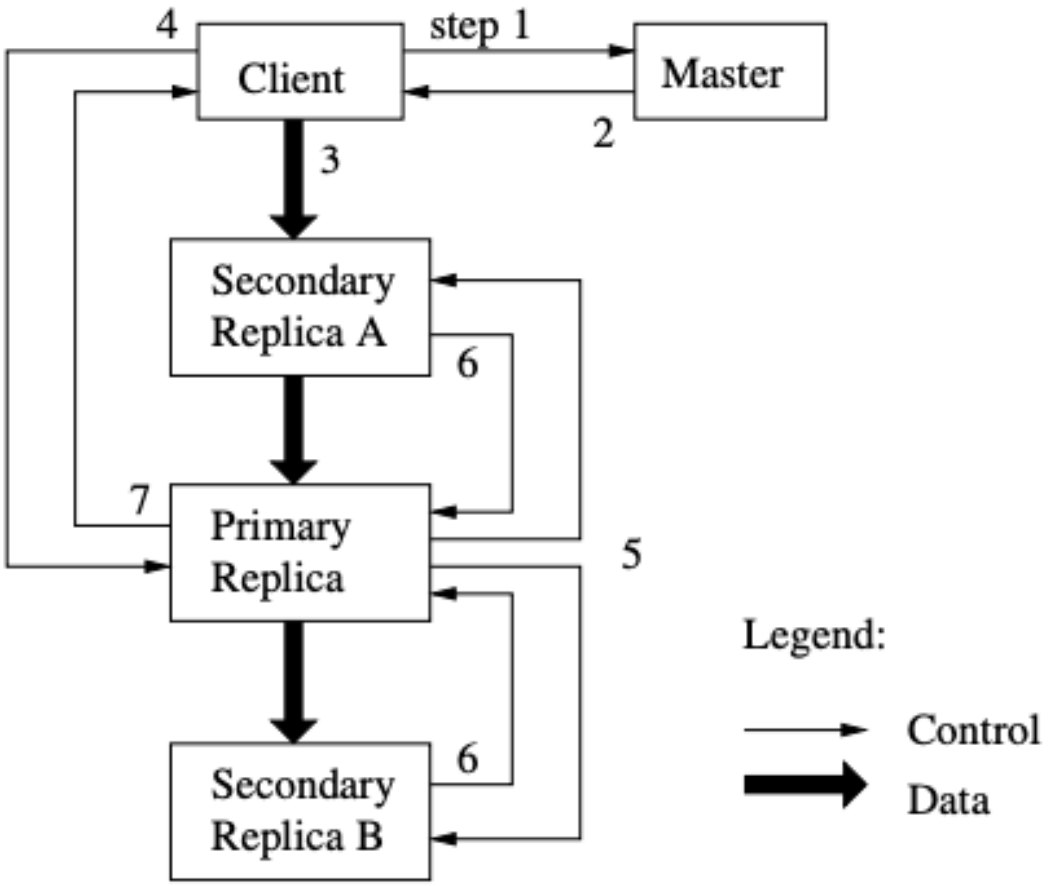
Leases

- Lease = master server grants to a chunkserver a 60-sec permission to modify the chunk
  - Can revoke earlier, e.g., if master wants to rename file
- Primary chunkserver…
  - Chooses the serial number for all changes on a chunk
  - Propagates the changes to the chunkservers with the backup copies, doesn't save until all acknowledge

Write

- Client asks master what chunkserver has the lease for the chunk it wants to write to
  - If no lease exists, master chooses a chunkserver and issues a lease
- Master replies to client with location of the primary chunkserver and replicas for that chunk
- Client pushes data to all replicas in any order, they buffer
  - Note: decoupling of data flow and control flow (why?)
- Once all replicas ack, client sends the update to primary
- Primary assigns the order and sends the update to the replicas
- Replicas apply changes in given order and ack to primary
- Primary replies to client

Write Overview



Write: What Can Go Wrong?

- Client gets the reply back: write succeeded at the primary and at some replicas
  - Any error at replicas is reported to the client
- Errors at the primary: inconsistent state
- Data larger than a chunk needs multiple writes
  - .. may overlap with operations from other clients
  - … different clients may overwrite each other's operations
  - … order is the same but fragments are from different clients – undefined state

Append

- Instead of writing new chunk, append to existing one
  - Restricted to chunk-size / 4 to minimize overflow of chunk boundaries… if overflows, create a new chunk or write into chunk at some offset and tell the replicas
- Guaranteed: record was written at least once atomically (there can be duplicates)
  - Not guaranteed: replicas are byte-wise identical
  - Failures at subset of chunkservers result in inconsistent regions
  - If a replica was offline and missed some updates, may have holes after record append

Snapshot

- Makes a copy of file or directory tree
- Used to quickly create branch copies of huge data sets, or to checkpoint current state to allow rollback
- Uses copy-on-write (like AFS)
  - Master revokes chunk leases, then snapshots
  - Any time a lease is requested, chunk is copied first on same chunkserver
  - After lease expires, commit to log + copy metadata

Snapshot Details

- After receiving a snapshot request, master revokes existing leases… any modification on those files must go to the master first
- Master logs operation on disk
- Master changes metadata, new snapshot file points to same chunks as original file
- Client wants to write chunk C after snapshot, master sees reference count for C > 1, picks new id C’, asks chunkserver that has C to create a copy with the new id C’

Namespace Management & Locking

- GFS is not a typical file system
  - No typical per-directory data structure to list files
  - Does not support aliases (i.e., hard or sym links)
  - Namespace: lookup table that maps full pathnames to metadata, fits in memory (prefix compression)
- Master ops acquire set of locks before running
  - Ex: to read "leaf", lock /d1, /d1/d2, /d1/d2/../dn/leaf
  - File creation does not lock parent dir
    - No directory or inode-like structure to modify
  - Allows concurrent modifications in same directory
  - Locks always acquired in same order, prevents deadlock

Replica Distribution

- Hundreds of chunkservers across many hardware racks, each accessed by hundreds of clients on same or different racks
- Want to maximize reliability, availability… but also network utilization
- Must spread replicas across machines and racks
  - Replicas survive switch failure or power outage of entire rack, etc.
  - Read traffic for a chunk can exploit bandwidth of multiple racks
  - Write traffic must flow through multiple racks

Chunk Creation

- Place new replicas on chunkservers with below-average disk utilization
  - Equalizes disk utilization across chunkservers
- Limit number of recently created chunks per chunkserver
  - Creation implies chunk will be heavily written soon
- Place new replicas on different racks
  - Requires IP addressing scheme to identify physical location of chunkservers

Re-replication and Rebalancing

- Chunks re-replicated as soon as the number of available replicas falls below user-specified goal
  - Prioritize chunks that are far from goal, chunks of live (vs recently deleted) files, chunks that are blocking client progress
  - "Clone" chunk data from existing replicas
- Periodically move replicas for better disk space and load balancing
  - Helps gradually fill up a new chunkserver instead of swamping it (new chunks = heavy write traffic)

Garbage Collection

- "Deleting" a file only renames it… now it's "hidden"
- Hidden files are removed after 3 days (configurable)
  - Prevents accidental-and-irreversible data loss
- "Undelete" by renaming hidden file
- Lazy garbage collection, i.e., during idle time
  - In-memory metadata erased
  - Orphaned chunks gradually erased by chunkservers
  - Also removes stale replicas (not up to current chunk version #)
- 3-day delay caused some trouble for some users
  - Temp files stick around, wasting space
  - Replication level can be user specified per part of file namespace

Fault Tolerance and Availability

- Fast recovery
  - Master, chunk servers restore state quickly
  - No distinction between normal/abnormal termination
- Chunk replication across chunkservers
- Master replication
  - State of master is replicated, external watchdog can change DNS over to replica if master fails
  - Additional “shadow” masters provide RO access during outage
    - Metadata may lag the primary master by fractions of second
    - Depends on primary master for replica location updates

Integrity of Chunks

- Chunks can get corrupted by disk failures, interruptions in r/w paths
  - Each server must checksum because chunks not byte-wise equal
  - Chunks are broken into 64KB blocks, each with 32-bit checksum – kept in memory and logged with metadata
  - Can overlap with IO since checksums all in memory
- Client code attempts to align reads to checksum block boundaries
- Can checksum inactive chunks during idle periods