Storage Systems

NPTEL Course

Jan 2012

(Lecture 39)

K. Gopinath

Indian Institute of Science

Google File System

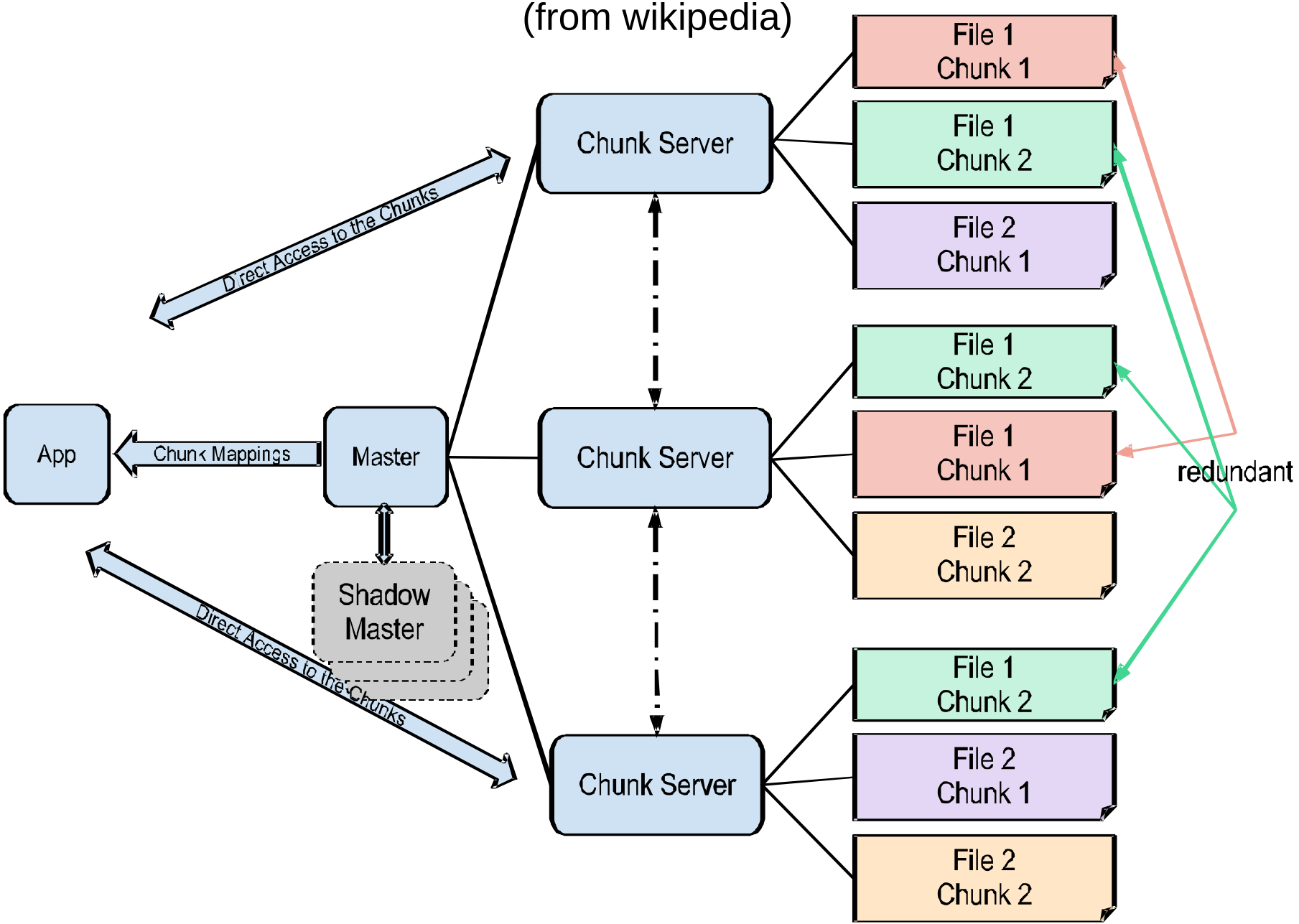
- Non-Posix scalable distr file system for large distr data-intensive applications
  - performance, scalability, reliability and availability
  - high aggregate perf to a large number of clients
    - sustained bandwidth more important than low latency
- High fault tolerance to allow inexpensive commodity HW
  - component failures norm rather than exception
  - appl/OS bugs, human errors + failures of disks, memory, connectors, networking, and power supplies.
  - constant monitoring, error detection, fault tolerance, and automatic recovery integral in the design

GFS Design

- optimized for Google's appl workloads + curr tech
  - modest number of large files (64MB): good for scans, streams, archives, shared Qs
  - append new data common, not overwriting existing data or random writes:
    - perf opts and atomicity guarantees for appends
      - atomic append so that multiple clients can append concurrently to a file without extra synchronization
    - no client caching of data blocks
  - relaxed consistency model
  - cross layer design of appls + file system API

GFS Design

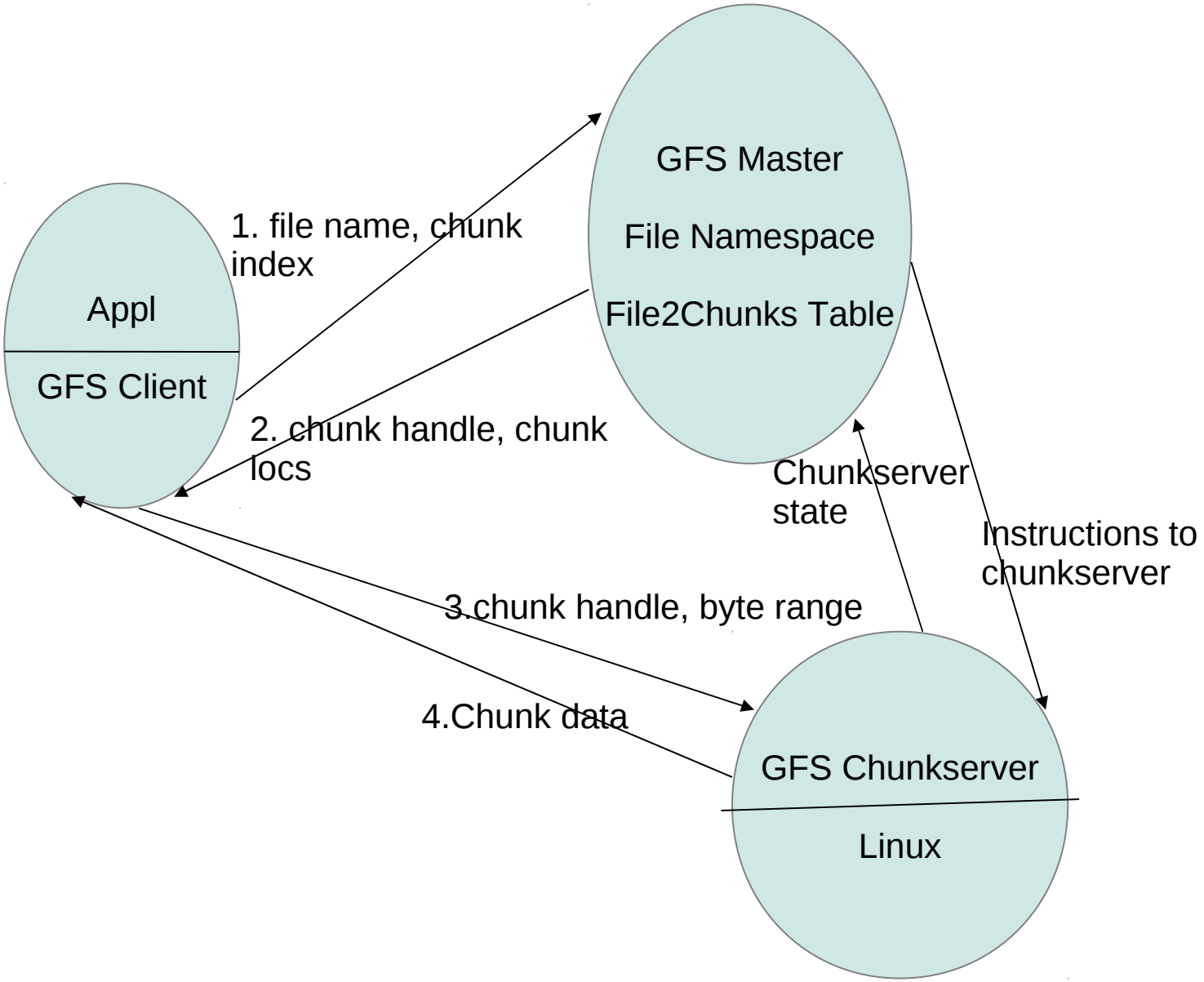
(from wikipedia)



GFS Design

- Std ops: create, delete, open, close, read, write
- Also snapshot (copy of a file or a dir tree at low cost) and record append (atomic with multiple clients)
- Single master, multiple chunkservers/clients
  - user-level server processes on Linux
- Files divided into fixed-size chunks, each identified by an immutable and globally unique 64 bit chunk handle assigned by master at chunk create
- Chunkservers store chunks on local disks as Linux files
  - read or write chunk data specified by a chunk handle and byte range
- For reliability, each chunk replicated on multiple chunkservers.
  - 3 replicas default, but can have different replication levels for different regions of file namespace

GFS


GFS Master
File Namespace
File2Chunks Table
Appl
GFS Client
1. file name, chunk index
2. chunk handle, chunk locs
Chunkserver state
Instructions to chunkserver
3.chunk handle, byte range
4.Chunk data
GFS Chunkserver
Linux

Large Chunk Size

- reduces clients' need to interact with master
  - reads and writes on same chunk require only one initial request to master for chunk loc info: appls mostly read and write large files sequentially
  - for small random reads, the client can cache all chunk loc info for a multi-TB working set.
  - with a large chunk, a client is more likely to perform many ops on a given chunk
    - reduce netw overhead by keeping a persistent TCP cnxn to chunkserver for an extended time.
  - reduces size of metadata on master: all metadata in memory!
- but hotspot if a file a popular single chunk executable
  - popular executables need higher replication factor
  - also, stagger application start times
  - (future?) allow clients to read data from other clients

GFS Design

- GFS client code linked into each appl: implements FS API
  - communicates with master and chunkservers to read/write data on behalf of appl
  - clients interact with master for metadata ops (cached for a small time), but all data-bearing comm directly to chunkservers
- No caching of file data by client or chunkserver but clients cache metadata
  - chunkservers need not cache file data: chunks may already be cached as local files in Linux's buffer cache
- Each chunk replica stored as a Linux file on a chunkserver and extended as needed
  - lazy space allocation avoids space wasting due to internal fragmentation

GFS Metadata Mgmt

- Master maintains all FS metadata: file/chunk namespaces, access control info, mapping from files to chunks, current locations of chunk's replicas
  - also controls system-wide activities such as chunk lease mgmt, GC of orphaned chunks, and chunk migration between chunkservers
  - periodically communicates with each chunkserver using heartbeats
  - single master makes design simple: better chunk placement /replication using global knowledge

GFS Master

- Master does not store chunk loc info persistently.
  - asks each chunkserver about its chunks at master startup and whenever a chunkserver joins the cluster
  - up-to-date thereafter as it controls all chunk placement and monitors chunkserver status with regular heartbeat messages.
  - earlier design problematic: difficult to maintain consistency betw master and chunkservers
    - chunkservers join and leave the cluster, change names, fail, restart
    - chunkserver has the final word over what chunks it does or does not have on its own disks as:
    - errors on a chunkserver may cause chunks to vanish (e.g., a disk may go bad and be disabled) or an operator may rename a chunkserver

Memory-based metadata in Master

- all metadata kept in master's memory
  - easy and efficient for master to periodically scan through its entire state in the background: for chunk garbage collection, re-replication on chunkserver failures, chunk migration to balance load and disk space
  - practical: < 64 bytes of metadata for each 64 MB chunk (64MB for 64TB!)
  - most chunks full: large files many chunks, only last may be partially filled
  - file namespace data typ < 64 bytes per file: file names stored compactly with prefix compression

Logs

- Namespaces and file-to-chunk mapping kept persistent by logging mutations to an operation log stored on the master's local disk and replicated on remote machines.
  - a log allows reliable upd of master state without inconsistencies if master crashes
  - serves as a logical time line that defines order of concurrent ops
    - files and chunks, as well as their versions, uniquely and eternally identified by logical times of their create
  - changes not visible to clients until metadata changes persistent
    - else can lose whole fs or recent client ops even if chunks survive
    - respond to a client op only after flushing log record to disk both locally and remotely.
    - master batches several log records before flushing to reduce impact of flushing and replication on system throughput

Recovery

- master recovers fs state by replaying the op log
  - to minimize startup time, keep log small by checkpointing master state whenever the log grows beyond a certain size
    - recover by loading latest checkpoint from local disk and replaying only limited number of later log records
    - checkpoint a compact B-tree like form directly mappable to memory
      - namespace lookup requires no extra parsing
      - speeds up recovery and improves availability
    - building a checkpoint takes time:
      - master switches to a new log file and creates new checkpoint in a separate thread
      - no delay for incoming mutations
      - new checkpoint includes all mutations before switch
      - a minute or so for a cluster with a few million files.
      - When completed, written to disk both locally and remotely.
    - recovery needs only the latest complete checkpoint and subsequent log files
      - failure during checkpointing does not affect correctness: recovery code detects and skips incomplete checkpoints