The Google File System

Firas Abuzaid

Why build GFS?

- Node failures happen frequently
- Files are huge – multi-GB
- Most files are modified by appending at the end
    - Random writes (and overwrites) are practically non-existent
- High sustained bandwidth is more important than low latency
    - Place more priority on processing data in bulk

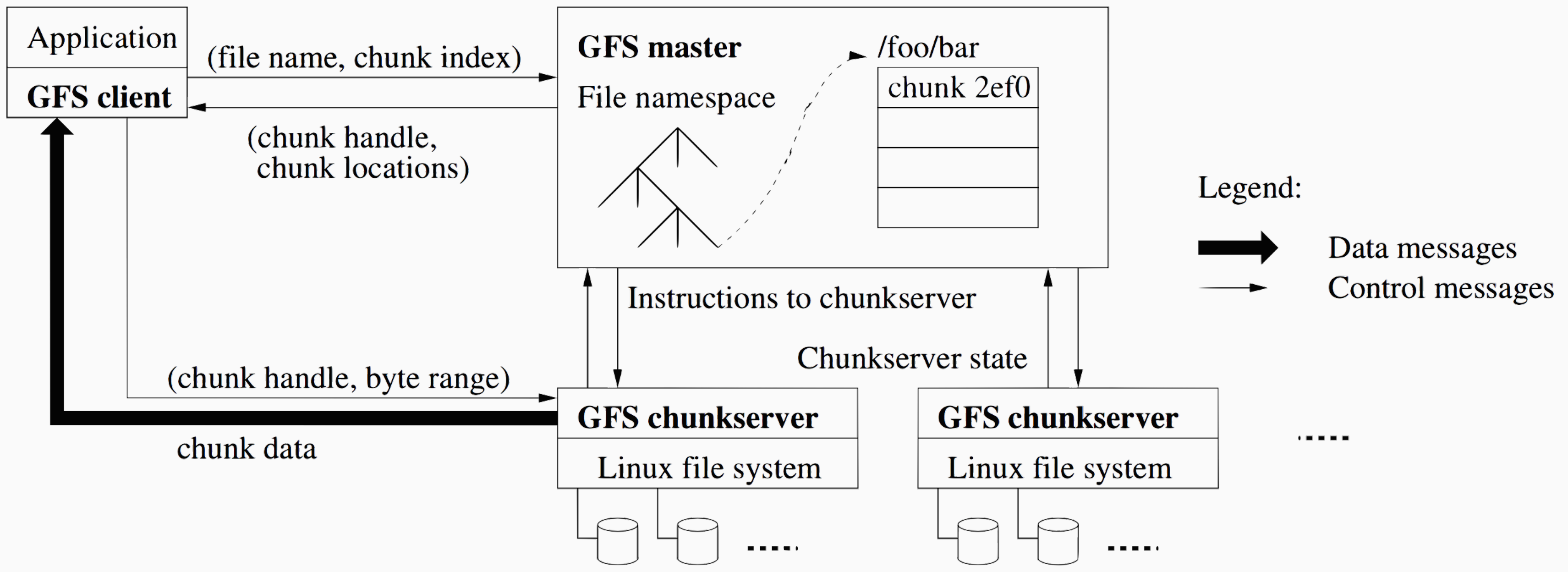
Typical workloads on GFS

- Two kinds of reads: large streaming reads & small random reads
  - Large streaming reads usually read 1MB or more
  - Oftentimes, applications read through contiguous regions in the file
  - Small random reads are usually only a few KBs at some arbitrary offset
- Also many large, sequential writes that append data to files
  - Similar operation sizes to reads
  - Once written, files are seldom modified again
  - Small writes at arbitrary offsets do not have to be efficient
- Multiple clients (e.g. ~100) concurrently appending to a single file
  - e.g. producer-consumer queues, many-way merging

Interface

- Not POSIX-compliant, but supports typical file system operations: `create`, `delete`, `open`, `close`, `read`, and `write`
- `snapshot`: creates a copy of a file or a directory tree at low cost
- `record append`: allow multiple clients to append data to the same file concurrently
    - At least the very first append is guaranteed to be atomic

Architecture



Architecture

- **Very important**: data flow is decoupled from control flow
  - Clients interact with the master for metadata operations
  - Clients interact directly with chunkservers for all files operations
  - This means performance can be improved by scheduling expensive data flow based on the network topology
- Neither the clients nor the chunkservers cache file data
  - Working sets are usually too large to be cached, chunkservers can use Linux’s buffer cache

The Master Node

- Responsible for all system-wide activities
  - managing chunk leases, reclaiming storage space, load-balancing
- Maintains all file system metadata
  - Namespaces, ACLs, mappings from files to chunks, and current locations of chunks
  - all kept in memory, namespaces and file-to-chunk mappings are also stored persistently in ***<u>operation log</u>***
- Periodically communicates with each chunkserver in `HeartBeat` messages
  - This let's master determines chunk locations and assesses state of the overall system
  - **Important**: <u>The chunkserver has the final word over what chunks it does or does not have</u> on its own disks – **not** the master

The Master Node

- For the namespace metadata, master does not use any per-directory data structures – no inodes! (No symlinks or hard links, either.)
  - Every file and directory is represented as a node in a lookup table, mapping pathnames to metadata. Stored efficiently using prefix compression (< 64 bytes per namespace entry)
- Each node in the namespace tree has a corresponding read-write lock to manage concurrency
  - Because all metadata is stored in memory, the master can efficiently scan the entire state of the system periodically in the background
  - Master's memory capacity does not limit the size of the system

The Operation Log

- Only persistent record of metadata
- Also serves as a logical timeline that defines the serialized order of concurrent operations
- Master recovers its state by replaying the operation log
    - To minimize startup time, the master checkpoints the log periodically
        - The checkpoint is represented in a B-tree like form, can be directly mapped into memory, but stored on disk
        - Checkpoints are created without delaying incoming requests to master, can be created in ~1 minute for a cluster with a few million files

Why a Single Master?

- The master now has global knowledge of the whole system, which drastically simplifies the design
- But the master is (hopefully) never the bottleneck
    - Clients never read and write file data through the master; client only requests from master which chunkservers to talk to
    - Master can also provide additional information about subsequent chunks to further reduce latency
    - Further reads of the same chunk don’t involve the master, either

Why a Single Master?

- Master state is also replicated for reliability on multiple machines, using the operation log and checkpoints
    - If master fails, GFS can start a new master process at any of these replicas and modify DNS alias accordingly
    - “Shadow” masters also provide read-only access to the file system, even when primary master is down
        - They read a replica of the operation log and apply the same sequence of changes
        - Not mirrors of master – they lag primary master by fractions of a second
        - This means we can still read up-to-date file contents while master is in recovery!

Chunks and Chunkservers

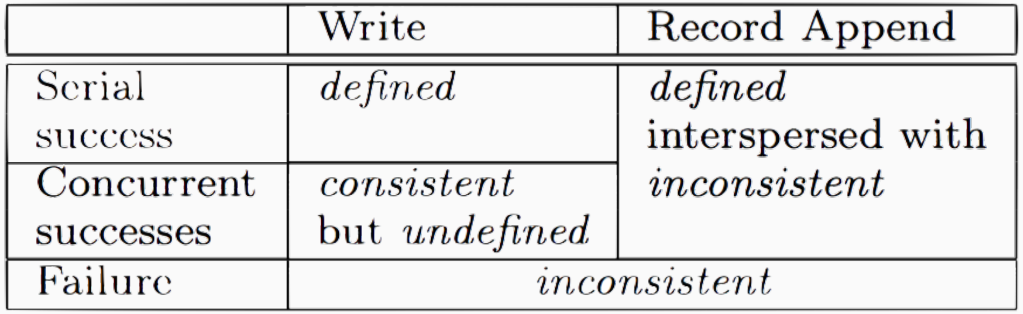
- Files are divided into fixed-size ***chunks***, which has an immutable, globally unique 64-bit ***chunk handle***
    - By default, each chunk is replicated three times across multiple chunkservers (user can modify amount of replication)
- Chunkservers store the chunks on local disks as Linux files
    - Metadata per chunk is < 64 bytes (stored in master)
        - Current replica locations
        - Reference count (useful for copy-on-write)
        - Version number (for detecting stale replicas)

Chunk Size

- 64 MB, a key design parameter (Much larger than most file systems.)
- Disadvantages:
  - Wasted space due to internal fragmentation
  - Small files consist of a few chunks, which then get lots of traffic from concurrent clients
    - This can be mitigated by increasing the replication factor
- Advantages:
  - Reduces clients' need to interact with master (reads/writes on the same chunk only require one request)
  - Since client is likely to perform many operations on a given chunk, keeping a persistent TCP connection to the chunkserver reduces network overhead
  - Reduces the size of the metadata stored in master → metadata can be entirely kept in memory

GFS's Relaxed Consistency Model

- Terminology:
  - ***consistent***: all clients will always see the same data, regardless of which replicas they read from
  - ***defined***: same as ***consistent*** and, furthermore, clients will see what the modification is in its entirety
- Guarantees:



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

Data Modifications in GFS

- After a sequence of modifications, if successful, then modified file region is guaranteed to be ***defined*** and contain data written by last modification
- GFS applies modification to a chunk in the same order on all its replicas
- A chunk is lost irreversibly **if and only if** all its replicas are lost before the master node can react, typically within minutes
  - even in this case, data is lost, not corrupted

Record Appends

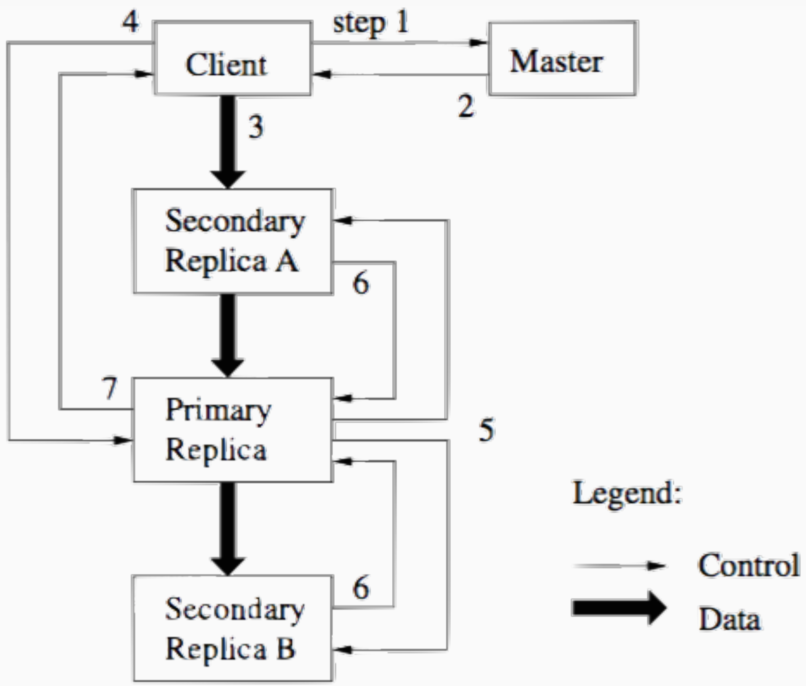
- A modification operation that guarantees that data (the “record”) will be appended atomically at least once – but at the offset of GFS’s choosing
  - The offset chosen by GFS is returned to the client so that the application is aware
- GFS may insert padding or record duplicates in between different record append operations
- Preferred that applications use this instead of `write`
  - Applications should also write self-validating records (e.g. checksumming) with unique IDs to handle padding/duplicates

System Interactions

- If the master receives a modification operation for a particular chunk:
  - Master finds the chunkservers that have the chunk and grants a ***chunk lease*** to one of them
    - This server is called the ***primary***, the other servers are called ***secondaries***
    - The primary determines the serialization order for all of the chunk's modifications, and the secondaries follow that order
  - After the lease expires (~60 seconds), master may grant primary status to a different server for that chunk
    - The master can, at times, revoke a lease (e.g. to disable modifications when file is being renamed)
    - As long as chunk is being modified, the primary can request an extension indefinitely
  - If master loses contact with primary, that's okay: just grant a new lease after the old one expires

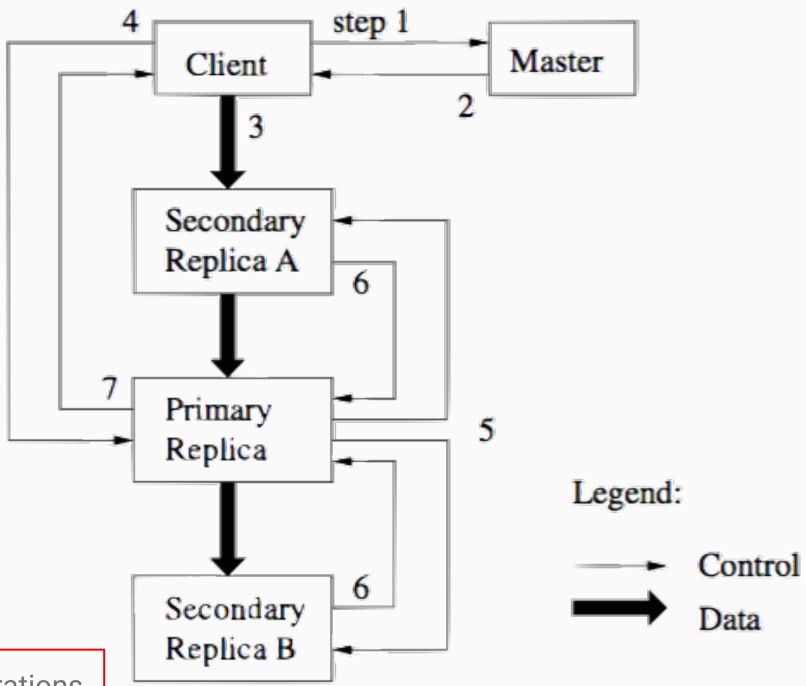
System Interactions

1. Client asks master for all chunkservers (including all secondaries)
2. Master grants a new lease on chunk, increases the chunk version number, tells all replicas to do the same. Replies to client. Client no longer has to talk to master
3. Client pushes data to all servers, not necessarily to primary first
4. Once data is acked, client sends write request to primary. Primary decides serialization order for all incoming modifications and applies them to the chunk



System Interactions

5. After finishing the modification, primary forwards write request and serialization order to secondaries, so they can apply modifications in same order. (If primary fails, this step is never reached.)
6. All secondaries reply back to the primary once they finish the modifications
7. Primary replies back to the client, either with success or error
   - If write succeeds at primary but fails at any of the secondaries, then we have inconsistent state → error returned to client
   - Client can retry steps (3) through (7)



Note: If a `write` straddles chunk boundary, GFS splits this into multiple `write` operations

System Interactions for Record Appends

- Same as before, but with the following extra steps:
  - In step (4), the primary checks to see if appending record to current chunk would exceed max size (64 MB)
    - If so, pads the chunk, notifies secondaries to do the same, and tells client to retry request on next chunk
    - Record append is restricted to ¼th max chunk size → at most, padding will be 16 MB
- If record append fails at any of the replicas, the client must retry
  - This means that replicas of the same chunk may contain duplicates
- A successful record append? That means the data must have been written at <u>the same offset</u> on all replicas of the chunk
  - Hence, GFS guarantees that record append will be ***<u>defined</u>*** interspersed with ***<u>inconsistent</u>***

Conclusions

- De-coupling of data flow vs. control flow is super-important
- Single-master design can be, in certain circumstances, quite advantageous
- Focusing on the core use cases of the file system (e.g. atomic appends) can lead you to the right abstractions

Questions?