```
6.824 2020 Lecture 3: GFS

The Google File System
Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
SOSP 2003

Why are we reading this paper?
  distributed storage is a key abstraction
    what should the interface/semantics look like?
    how should it work internally?
  GFS paper touches on many themes of 6.824
    parallel performance, fault tolerance, replication, consistency
  good systems paper -- details from apps all the way to network
  successful real-world design

Why is distributed storage hard?
  high performance -> shard data over many servers
  many servers -> constant faults
  fault tolerance -> replication
  replication -> potential inconsistencies
  better consistency -> low performance

What would we like for consistency?
  Ideal model: same behavior as a single server
  server uses disk storage
  server executes client operations one at a time (even if concurrent)
  reads reflect previous writes
    even if server crashes and restarts
  thus:
    suppose C1 and C2 write concurrently, and after the writes have
      completed, C3 and C4 read. what can they see?
    C1: Wx1
    C2: Wx2
    C3:     Rx?
    C4:         Rx?
    answer: either 1 or 2, but both have to see the same value.
  This is a "strong" consistency model.
  But a single server has poor fault-tolerance.

Replication for fault-tolerance makes strong consistency tricky.
  a simple but broken replication scheme:
    two replica servers, S1 and S2
    clients send writes to both, in parallel
    clients send reads to either
  in our example, C1's and C2's write messages could arrive in
    different orders at the two replicas
    if C3 reads S1, it might see x=1
    if C4 reads S2, it might see x=2
  or what if S1 receives a write, but
    the client crashes before sending the write to S2?
  that's not strong consistency!
  better consistency usually requires communication to
    ensure the replicas stay in sync -- can be slow!
  lots of tradeoffs possible between performance and consistency
    we'll see one today

GFS

Context:
  Many Google services needed a big fast unified storage system
    Mapreduce, crawler/indexer, log storage/analysis, Youtube (?)
  Global (over a single data center): any client can read any file
    Allows sharing of data among applications
  Automatic "sharding" of each file over many servers/disks
```

```
    For parallel performance
    To increase space available
  Automatic recovery from failures
  Just one data center per deployment
  Just Google applications/users
  Aimed at sequential access to huge files; read or append
    I.e. not a low-latency DB for small items

What was new about this in 2003? How did they get an SOSP paper accepted?
  Not the basic ideas of distribution, sharding, fault-tolerance.
  Huge scale.
  Used in industry, real-world experience.
  Successful use of weak consistency.
  Successful use of single master.

Overall structure
  clients (library, RPC -- but not visible as a UNIX FS)
  each file split into independent 64 MB chunks
  chunk servers, each chunk replicated on 3
  every file's chunks are spread over the chunk servers
    for parallel read/write (e.g. MapReduce), and to allow huge files
  single master (!), and master replicas
  division of work: master deals w/ naming, chunkservers w/ data

Master state
  in RAM (for speed, must be smallish):
    file name -> array of chunk handles (nv)
    chunk handle -> version # (nv)
                    list of chunkservers (v)
                    primary (v)
                    lease time (v)
  on disk:
    log
    checkpoint

Why a log? and checkpoint?

Why big chunks?

What are the steps when client C wants to read a file?
  1. C sends filename and offset to master M (if not cached)
  2. M finds chunk handle for that offset
  3. M replies with list of chunkservers
     only those with latest version
  4. C caches handle + chunkserver list
  5. C sends request to nearest chunkserver
     chunk handle, offset
  6. chunk server reads from chunk file on disk, returns

How does the master know what chunkservers have a given chunk?

What are the steps when C wants to do a "record append"?
  paper's Figure 2
  1. C asks M about file's last chunk
  2. if M sees chunk has no primary (or lease expired):
     2a. if no chunkservers w/ latest version #, error
     2b. pick primary P and secondaries from those w/ latest version #
     2c. increment version #, write to log on disk
     2d. tell P and secondaries who they are, and new version #
     2e. replicas write new version # to disk
  3. M tells C the primary and secondaries
  4. C sends data to all (just temporary...), waits
  5. C tells P to append
  6. P checks that lease hasn't expired, and chunk has space
  7. P picks an offset (at end of chunk)
  8. P writes chunk file (a Linux file)
  9. P tells each secondary the offset, tells to append to chunk file
  10. P waits for all secondaries to reply, or timeout
      secondary can reply "error" e.g. out of disk space
  11. P tells C "ok" or "error"
  12. C retries from start if error

What consistency guarantees does GFS provide to clients?
  Needs to be in a form that tells applications how to use GFS.

Here's a possibility:

  If the primary tells a client that a record append succeeded, then
  any reader that subsequently opens the file and scans it will see
  the appended record somewhere.

(But not that failed appends won't be visible, or that all readers
 will see the same file content, or the same order of records.)

How can we think about how GFS fulfils this guarantee?
  Look at its handling of various failures:
    crash, crash+reboot, crash+replacement, message loss, partition.
  Ask how GFS ensures critical properties.

* What if an appending client fails at an awkward moment?
  Is there an awkward moment?

* What if the appending client has cached a stale (wrong) primary?

* What if the reading client has cached a stale secondary list?

* Could a master crash+reboot cause it to forget about the file?
  Or forget what chunkservers hold the relevant chunk?

* Two clients do record append at exactly the same time.
  Will they overwrite each others' records?

* Suppose one secondary never hears the append command from the primary.
  What if reading client reads from that secondary?

* What if the primary crashes before sending append to all secondaries?
  Could a secondary that *didn't* see the append be chosen as the new primary?

* Chunkserver S4 with an old stale copy of chunk is offline.
  Primary and all live secondaries crash.
  S4 comes back to life (before primary and secondaries).
  Will master choose S4 (with stale chunk) as primary?
  Better to have primary with stale data, or no replicas at all?

* What should a primary do if a secondary always fails writes?
  e.g. dead, or out of disk space, or disk has broken.
  Should the primary drop secondary from set of secondaries?
    And then return success to client appends?
  Or should the primary keep sending ops, and having them fail,
    and thus fail every client write request?

* What if primary S1 is alive and serving client requests,
    but network between master and S1 fails?
  "network partition"
  Will the master pick a new primary?
  Will there now be two primaries?
  So that the append goes to one primary, and the read to the other?
    Thus breaking the consistency guarantee?
    "split brain"
* If there's a partitioned primary serving client appends, and its
  lease expires, and the master picks a new primary, will the new
  primary have the latest data as updated by partitioned primary?

* What if the master fails altogether.
  Will the replacement know everything the dead master knew?
  E.g. each chunk's version number? primary? lease expiry time?

* Who/what decides the master is dead, and must be replaced?
  Could the master replicas ping the master, take over if no response?

* What happens if the entire building suffers a power failure?
  And then power is restored, and all servers reboot.

* Suppose the master wants to create a new chunk replica.
  Maybe because too few replicas.
  Suppose it's the last chunk in the file, and being appended to.
  How does the new replica ensure it doesn't miss any appends?
    After all it is not yet one of the secondaries.

* Is there *any* circumstance in which GFS will break the guarantee?
  i.e. append succeeds, but subsequent readers don't see the record.
  All master replicas permanently lose state (permanent disk failure).
    Could be worse: result will be "no answer", not "incorrect data".
    "fail-stop"
  All chunkservers holding the chunk permanently lose disk content.
    again, fail-stop; not the worse possible outcome
  CPU, RAM, network, or disk yields an incorrect value.
    checksum catches some cases, but not all
  Time is not properly synchronized, so leases don't work out.
    So multiple primaries, maybe write goes to one, read to the other.

What application-visible anomalous behavior does GFS allow?
  Will all clients see the same file content?
    Could one client see a record that another client doesn't see at all?
    Will a client see the same content if it reads a file twice?
  Will all clients see successfully appended records in the same order?

Will these anomalies cause trouble for applications?
  How about MapReduce?

What would it take to have no anomalies -- strict consistency?
  I.e. all clients see the same file content.
  Too hard to give a real answer, but here are some issues.
  * Primary should detect duplicate client write requests.
    Or client should not issue them.
  * All secondaries should complete each write, or none.
    Perhaps tentative writes until all promise to complete it?
    Don't expose writes until all have agreed to perform them!
  * If primary crashes, some replicas may be missing the last few ops.
    New primary must talk to all replicas to find all recent ops,
    and sync with secondaries.
  * To avoid client reading from stale ex-secondary, either all client
    reads must go to primary, or secondaries must also have leases.
  You'll see all this in Labs 2 and 3!

Performance (Figure 3)
  large aggregate throughput for read (3 copies, striping)
    94 MB/sec total for 16 chunkservers
      or 6 MB/second per chunkserver
      is that good?
      one disk sequential throughput was about 30 MB/s
      one NIC was about 10 MB/s
    Close to saturating network (inter-switch link)
    So: individual server performance is low
```

```
        but scalability is good
        which is more important?
    Table 3 reports 500 MB/sec for production GFS, which is a lot
  writes to different files lower than possible maximum
    authors blame their network stack (but no detail)
  concurrent appends to single file
    limited by the server that stores last chunk
  hard to interpret after 15 years, e.g. how fast were the disks?

Random issues worth considering
  What would it take to support small files well?
  What would it take to support billions of files?
  Could GFS be used as wide-area file system?
    With replicas in different cities?
    All replicas in one datacenter is not very fault tolerant!
  How long does GFS take to recover from a failure?
    Of a primary/secondary?
    Of the master?
  How well does GFS cope with slow chunkservers?

Retrospective interview with GFS engineer:
  http://queue.acm.org/detail.cfm?id=1594206
  file count was the biggest problem
    eventual numbers grew to 1000x those in Table 2 !
    hard to fit in master RAM
    master scanning of all files/chunks for GC is slow
  1000s of clients too much CPU load on master
  applications had to be designed to cope with GFS semantics
    and limitations
  master fail-over initially entirely manual, 10s of minutes
  BigTable is one answer to many-small-files problem
  and Colossus apparently shards master data over many masters

Summary
  case study of performance, fault-tolerance, consistency
    specialized for MapReduce applications
  good ideas:
    global cluster file system as universal infrastructure
    separation of naming (master) from storage (chunkserver)
    sharding for parallel throughput
    huge files/chunks to reduce overheads
    primary to sequence writes
    leases to prevent split-brain chunkserver primaries
  not so great:
    single master performance
      ran out of RAM and CPU
    chunkservers not very efficient for small files
    lack of automatic fail-over to master replica
    maybe consistency was too relaxed
```