```
6.824 2018 Lecture 3: GFS

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

Why are we reading this paper?
  the file system used for map/reduce
  main themes of 6.824 show up in this paper
    trading consistency for simplicity and performance
    motivation for subsequent designs
  good systems paper -- details from apps all the way to network
    performance, fault-tolerance, consistency
  influential
    many other systems use GFS (e.g., Bigtable, Spanner @ Google)
    HDFS (Hadoop Distributed File System) based on GFS


What is consistency?
  A correctness condition
  Important but difficult to achieve when data is replicated
    especially when application access it concurrently
    [diagram: simple example, single machine]
    if an application writes, what will a later read observe?
      what if the read is from a different application?
    but with replication, each write must also happen on other machines
    [diagram: two more machines, reads and writes go across]
    Clearly we have a problem here.
  Weak consistency
    read() may return stale data  --- not the result of the most recent write
  Strong consistency
    read() always returns the data from the most recent write()
  General tension between these:
    strong consistency is easy for application writers
    strong consistency is bad for performance
    weak consistency has good performance and is easy to scale to many servers
    weak consistency is complex to reason about
  Many trade-offs give rise to different correctness conditions
    These are called "consistency models"
    First peek today; will show up in almost every paper we read this term

"Ideal" consistency model
  Let's go back to the single-machine case
  Would be nice if a replicated FS behaved like a non-replicated file system
    [diagram: many clients on the same machine accessing files on single disk]
  If one application writes, later reads will observe that write
  What if two application concurrently write to the same file?
    Q: what happens on a single machine?
    In file systems often undefined  --- file may have some mixed content
  What if two application concurrently write to the same directory
    Q: what happens on a single machine?
    One goes first, the other goes second (use locking)

Challenges to achieving ideal consistency
  Concurrency -- as we just saw; plus there are many disks in reality
  Machine failures -- any operation can fail to complete
  Network partitions -- may not be able to reach every machine/disk
  Why are these challenges difficult to overcome?
    Requires communication between clients and servers
      May cost performance
    Protocols can become complex --- see next week
      Difficult to implement system correctly
    Many systems in 6.824 don't provide ideal
      GFS is one example

GFS goals:
  With so many machines, failures are common
    must tolerate
    assume a machine fails once per year
```

```
    w/ 1000 machines, ~3 will fail per day.
  High-performance: many concurrent readers and writers
    Map/Reduce jobs read and store final result in GFS
    Note: *not* the temporary, intermediate files
  Use network efficiently: save bandwidth
  These challenges difficult combine with "ideal" consistency

High-level design / Reads
  [Figure 1 diagram, master + chunkservers]
  Master stores directories, files, names, open/read/write
    But not POSIX
  100s of Linux chunk servers with disks
    store 64MB chunks (an ordinary Linux file for each chunk)
    each chunk replicated on three servers
    Q: Besides availability of data, what does 3x replication give us?
       load balancing for reads to hot files
       affinity
    Q: why not just store one copy of each file on a RAID'd disk?
       RAID isn't commodity
       Want fault-tolerance for whole machine; not just storage device
    Q: why are the chunks so big?
       amortizes overheads, reduces state size in the master
  GFS master server knows directory hierarchy
    for directory, wht files are in it
    for file, knows chunk servers for each 64 MB
    master keeps state in memory
      64 bytes of metadata per each chunk
    master has private recoverable database for metadata
      operation log flushed to disk
      occasional asynchronous compression info checkpoint
      N.B.: != the application checkpointing in Â§2.7.2
      master can recovery quickly from power failure
    shadow masters that lag a little behind master
      can be promoted to master
  Client read:
    send file name and chunk index to master
    master replies with set of servers that have that chunk
      response includes version # of chunk
      clients cache that information
    ask nearest chunk server
      checks version #
      if version # is wrong, re-contact master

Writes
  [Figure 2-style diagram with file offset sequence]
  Random client write to existing file
    client asks master for chunk locations + primary
    master responds with chunk servers, version #, and who is primary
      primary has (or gets) 60s lease
    client computes chain of replicas based on network topology
    client sends data to first replica, which forwards to others
      pipelines network use, distributes load
    replicas ack data receipt
    client tells primary to write
      primary assign sequence number and writes
      then tells other replicas to write
      once all done, ack to client
    what if there's another concurrent client writing to the same place?
      client 2 get sequenced after client 1, overwrites data
      now client 2 writes again, this time gets sequenced first (C1 may be slow)
      writes, but then client 1 comes and overwrites
      => all replicas have same data (= consistent), but mix parts from C1/C2
         (= NOT defined)
  Client append (not record append)
    same deal, but may put parts from C1 and C2 in any order
    consistent, but not defined
    or, if just one client writes, no problem -- both consistent and defined

Record append
```

```
  Client record append
    client asks master for chunk locations
    client pushes data to replicas, but specifies no offset
    client contacts primary when data is on all chunk servers
      primary assigns sequence number
      primary checks if append fits into chunk
        if not, pad until chunk boundary
      primary picks offset for append
      primary applies change locally
      primary forwards request to replicas
      let's saw R3 fails mid-way through applying the write
      primary detects error, tells client to try again
    client retries after contacting master
      master has perhaps brought up R4 in the meantime (or R3 came back)
      one replica now has a gap in the byte sequence, so can't just append
      pad to next available offset across all replicas
      primary and secondaries apply writes
      primary responds to client after receiving acks from all replicas

Housekeeping
  Master can appoint new primary if master doesn't refresh lease
  Master replicates chunks if number replicas drop below some number
  Master rebalances replicas

Failures
  Chunk servers are easy to replace
    failure may cause some clients to retry (& duplicate records)
  Master: down -> GFS is unavailable
    shadow master can serve read-only operations, which may return stale data
    Q: Why not write operations?
          split-brain syndrome (see next lecture)

Does GFS achieve "ideal" consistency?
  Two cases: directories and files
  Directories: yes, but...
    Yes: strong consistency (only one copy)
    But: master not always available & scalability limit
  Files: not always
    Mutations with atomic appends
          record can be duplicated at two offsets
    while other replicas may have a hole at one offset
    Mutations without atomic append
      data of several clients maybe intermingled
      if you care, use atomic append or a temporary file and atomically rename
  An "unlucky" client can read stale data for short period of time
    A failed mutation leaves chunks inconsistent
      The primary chunk server updated chunk
      But then failed and the replicas are out of date
    A client may read an not-up-to-date chunk
    When client refreshes lease it will learn about new version #

Authors claims weak consistency is not a big problems for apps
  Most file updates are append-only updates
    Application can use UID in append records to detect duplicates
    Application may just read less data (but not stale data)
  Application can use temporary files and atomic rename

Performance (Figure 3)
  huge aggregate throughput for read (3 copies, striping)
    125 MB/sec in aggregate
    Close to saturating network
  writes to different files lower than possible maximum
    authors blame their network stack
    it causes delays in propagating chunks from one replica to next
  concurrent appends to single file
    limited by the server that stores last chunk
  numbers and specifics have changed a lot in 15 years!

Summary
```

```
  case study of performance, fault-tolerance, consistency
    specialized for MapReduce applications
  what works well in GFS?
    huge sequential reads and writes
    appends
    huge throughput (3 copies, striping)
    fault tolerance of data (3 copies)
  what less well in GFS?
    fault-tolerance of master
    small files (master a bottleneck)
    clients may see stale data
    appends maybe duplicated
```

References

http://queue.acm.org/detail.cfm?id=1594206 (discussion of gfs evolution)
http://highscalability.com/blog/2010/9/11/googles-colossus-makes-search-real-time-by-dumping-mapreduce.html