Advanced Topics in Computer Systems, CS262B
Prof Eric A. Brewer

Andrew File System (AFS)
Google File System

February 5, 2004

I. AFS

Goal: large-scale campus wide file system (5000 nodes)
- must be scalable, limit work of core servers
- good performance
- meet FS consistency requirements (?)
- managable system admin (despite scale)

400 users in the “prototype” -- a great reality check (makes the conclusions meaningful)
- most applications work w/o relinking or recompiling

Clients:
- user-level process, Venus, that handles local caching, + FS interposition to catch all requests
- interaction with servers only on file open/close (implies whole-file caching)
- always check cache copy on open() (in prototype)

Vice (servers):
- Server core is trusted; called “Vice”
- servers have one process per active client
- shared data among processes only via file system (!)
- lock process serializes and manages all lock/unlock requests
- read-only replication of namespace (centralized updates with slow propagation)
- prototype supported about 20 active clients per server, goal was >50

Revised client cache:
- keep data cache on disk, metadata cache in memory
- still whole file caching, changes written back only on close
- directory updates are write through, but cached locally for reads
- instead of check on open(), assume valid unless you get an invalidation callback (server must invalidate all copies before committing an update)
- allows name translation to be local (since you can now avoid round-trip for each step of the path)

Revised servers:

- move to file IDs for servers rather than pathnames (just like DDS, chunkservers in GFS)
- FIDs are globally unique -- can move files without changing FIDs (within one volume)
- Volumes map FIDs to locations (volume location db replicated on each server)
- moved to threads from processes
- added direct i-node interface to BSD (new API) to avoid have to use filenames on the servers; new index to map FIDs to i-nodes; some optimization made to Venus on the client side, which uses a local directory as the cache

Consistency:

- writes are visible locally immediately, but only globally at close()
- however this global write is not visible to others that already have an open copy
- metadata changes are visible immediately and globally
- no implicit locking; it is up to the app to do it right (just as in Unix)

Issues:

- must be able fit a file locally to open it at all (this was later relaxed)
- better semantics than NSF (with its 30 second windows)

Performance:

- defined the "Andrew Benchmark" for file systems, which is still used some
- almost 2x NFS performance, but 50% slower than stand-alone workstation

Management:

- introduced Volumes -- a group of files that form a partial subtree in the namespace
- a volume lives on one server at a time, but may be moved; a server has many volumes
- can move a volume easily: old server will forward requests to the new server until all servers know the location of the new server (eventual consistency of the volume location info)
- optimize for read-only files (no callbacks needed)
- backup: make a read-only copy of the whole volume and then move to archive (this uses copy on write)
- file system hierarchy is orthogonal to volume management; volumes are in a flat name sapce (volume ids) and are managed independently

II. Google File System

Key background:

- new workload => new filesystem (why?)
- extreme scale: 100 TB over 1000s of disks on >1000 machines
- new API as well

Four problems to solve:

- 1) Fault tolerance: this many components ensures regular failures. Must have automatic recovery.
- 2) huge files (LFS was optimized for small files!) -- but this is because they group files into large extents (multi GB). This seems weak...
- 3) Most common operation is append, not random writes
  - Most files are write once, and read-only after that
  - web snapshots, intermediate files in a pipeline, archival files
  - implies that streaming is much more important than block caching (and LRU would be a bad choice)
- 4) customize the API to enable optimization and flexibility (more below)
  - relaxed consistency model
  - atomic append

Operations:

- few million large files, rather than billions of small files
- large streaming reads, random reads
- large streaming writes, very rare random reads
- concurrency for append is critical (files act as a shared queue); also producer/consumer concurrency
- focus on throughput not latency (lots of parallel operations)

Architecture

- single master, multiple chunkservers, multiple clients
- fixed-size chunks (giant blocks) (how big? 64MB)
  - 64-bit ids for each chunk
  - clients read/write chunks directly from chunkservers
  - chunks are the unit of replication
- master maintains all metadata
  - namespace and access control
  - map from filenames to chunk ids
  - current locations for each chunk
- no caching for chunks (simplifies coherence; very different from xFS)
- metadata is cached at clients (coherence?)

Single master:

- claim: simple, but good enough
- enables good chuck placement (centralized decision)
- scalability is a concern, so never move data through it, only metadata
- clients cache (file name -> chunk id, replica locations), this expires eventually
- large chunk size reduces master RPC interaction and space overhead for metadata

- large chunks can become hot spots (but not for target workload)
- all metadata is in memory (limits the size of the filesystem, how much? metadata is 64B per chunk)

Durability:

- master logs changes to the namespace or to file->chunk mappings, these are reloaded on recovery
    - each log write is 2PC to multiple remote machines that put it on disk before committing
    - this is a replicated transactional redo log
    - group commit to reduce the overhead
    - checkpoint all (log) state periodically, so that we can truncate the log and reduce recovery time; checkpoint data is essentially an mmap file to avoid reading/parsing the data
    - checkpoint works by switching to new log, and copying snapshot in the background; this means that some updates in the new log will also be in the checkpoint, so log entries must be idempotent!
    - crash during checkpoint, will simply recover using the previous checkpoint (like version vector for latest checkpoint)
- chunk->replicas mapping is not logged, but reread from the chunkservers on recovery
- chunks are essentially two-phase commit to the replicas (just like DDS)

Periodic metadata scan:

- implements GC (how?)
- implements rereplication for chunks without enough replicas (this affects the window of vulnerability!)
- implements chunk migration for load balancing
- also monitors chunkservers with periodic heartbeat; this must verify checksums to detect bad replicas (and replace them)

chunk->replica data is fundamentally inconsistent!

- stale data is OK, is will just cause extra traffic to the master.
- Not clear what happens if a chunkserver silently drops a chunk: client will detect, or perhaps master during a periodic check, and then client to go to another replica (and should tell the master to update its advice)

Consistency model:

- namespace changes are atomic and serializable (easy since they go through one place)
- replicas: “defined” if it reflects a mutation and “consistent”; “consistent” -> all replicas have the same value
- concurrent writes will leave region consistent, but not necessarily defined; some updates may be lost
- a failed write leaves the region inconsistent
- record append: atomic append at least once; GFS decides the actual location
- to ensure definition, GFS must apply writes in the same order at all replicas
- consistency ensured via version numbers (and don’t use replicas with stale versions;

they get GC'd)

- client chunk caching may contain stale data! this is OK for append only files, but not for random writes. A new open() flushes the cache.
- primary replica decides the write order (it has lease for this right from the master)
  - lease extensions piggybacked on the heartbeat messages
  - master can revoke a lease but only with the replicas agreement; otherwise it has to wait for expiration
  - client pushes data out to all replicas using a data flow tree
  - after all data received, client notifies the primary to issue the write, which then gets ordered and can now be executed (fancy 2PC)
  - client retries if the write fails
  - writes across chunks are not transactional! ("consistent" but not "defined")
  - Append is easier to retry, since there is no risk of modfications to the same "place" that could fail repeatedly

Snapshots:

- a lot like AFS
- master revokes leases to ensure a clean snapshot (for those files)
- copy-on-write for metadata, so that snapshot uses old version, updates use new version
- replicas copy the chunk as well; the old copy becomes the snapshot archive

Replcation:

- important to replicate across racks to avoid correlated failures based on the rack
- used to re-replicate and re-balance load

GC:

- relatively easy: master has *all* references
- chunkservers know about all existing chunks; those not ref'd by master are garbage
- deletion is just rename to a magic name, that is later GC in periodic sweep.
- delayed collection is very robust! users make mistakes; eventually consistency also helped by delay
- delay also batches GC cost and thus reduces the overhead

Master replication:

- hard to make it truly HA
- fast recovery is primary step
- shadow read-only second master also helps
- note that logs are replicated so updates are not lost