Outline

- Get to know the numbers
- Filesystems overview
- Distributed file systems
  - Basic (example: NFS)
  - Shared storage (example: Global FS)
  - Wide-area (example: AFS)
  - Fault-tolerant (example: Coda)
  - Parallel (example: Lustre)
  - Fault-tolerant and Parallel (example: dCache)
- The Google File System
- Homework

Numbers real world engineers should know

L1 cache reference	0.5	ns
Branch mispredict	5	ns
L2 cache reference	7	ns
Mutex lock/unlock	100	ns
Main memory reference	100	ns
Compress 1 KB with Zippy	10,000	ns
Send 2 KB through 1 Gbps network	20,000	ns
Read 1 MB sequentially from memory	250,000	ns
Round trip within the same data center	500,000	ns
Disk seek	10,000,000	ns
Read 1 MB sequentially from network	10,000,000	ns
Read 1 MB sequentially from disk	30,000,000	ns
Round trip between California and Netherlands	150,000,000	ns

The Joys of Real Hardware

Typical first year for a new cluster:

~0.5 overheating (power down most machines in <5 mins, ~1-2 days to recover)
~1 PDU failure (~500-1000 machines suddenly disappear, ~6 hours to come back)
~1 rack-move (plenty of warning, ~500-1000 machines powered down, ~6 hours)
~1 network rewiring (rolling ~5% of machines down over 2-day span)
~20 rack failures (40-80 machines instantly disappear, 1-6 hours to get back)
~5 racks go wonky (40-80 machines see 50% packetloss)
~8 network maintenances (4 might cause ~30-minute random connectivity losses)
~12 router reloads (takes out DNS and external vips for a couple minutes)
~3 router failures (have to immediately pull traffic for an hour)
~dozens of minor 30-second blips for dns
~1000 individual machine failures
~thousands of hard drive failures

slow disks, bad memory, misconfigured machines, flaky machines, etc.

Motivation

- Google needed a good distributed file system
  - Redundant storage of massive amounts of data on cheap and unreliable computers (no RAIDs)
- Why not use an existing file system?
  - Google's problems are different from anyone else's
    - Different workload and design priorities
  - GFS is designed for Google apps and workloads
  - Google apps are designed for GFS

Assumptions

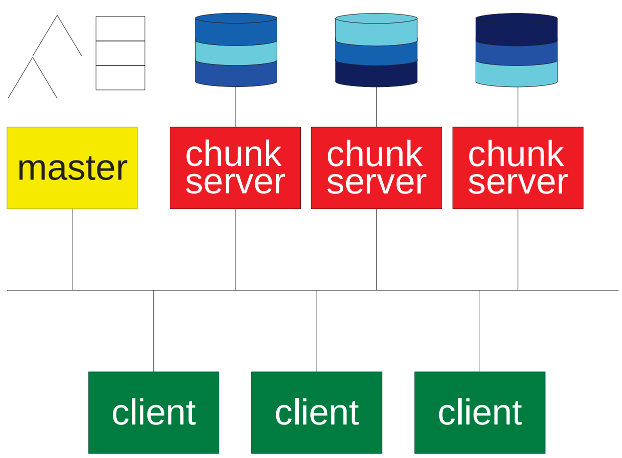
- High component failure rates
  - Inexpensive commodity components fail all the time
- “Modest” number of HUGE files
  - Just a few million
  - Each is 100MB or larger; multi-GB files typical
- Read: large streaming reads
- Write: write-once, mostly appended to (perhaps concurrently)
- *High sustained throughput* favored over *low latency*
  - We want to build **trucks**, not **scooters**

GFS Interface

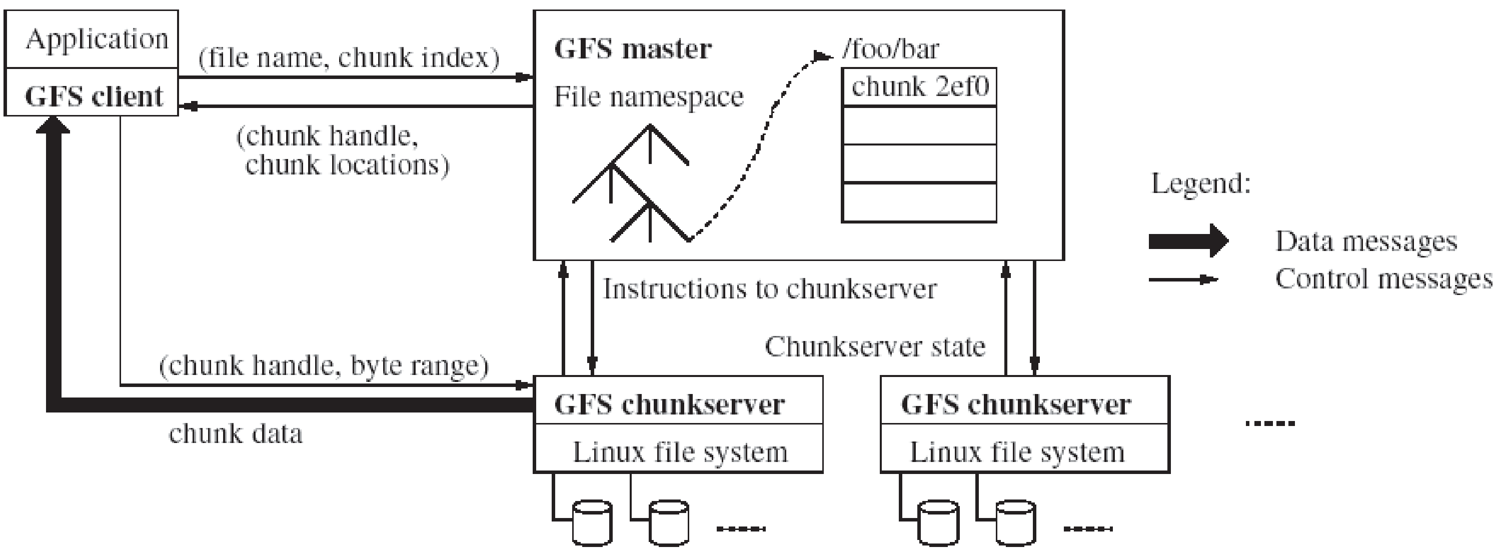
- Files organized into a directory hierarchy, identified with path names
- Supported operations:
  - create, delete, open, close, read, write a file.
  - directory
  - *snapshot*: make a copy of a file or a directory tree.
  - *record append*: atomic concurrent append.
- No POSIX APIs
  - don't have to implement Linux virtual file system interface

Roles in GFS

- Roles: master, chunkserver, client
  – Commodity Linux box, user level server processes
  – Client and chunkserver can run on the same box
- Master holds metadata, chunkservers hold data, client produces/consumes data.

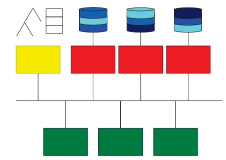


GFS Architecture



GFS Design Decisions

- Reliability through replication
- Single master to coordinate access, keep metadata
  - Simple centralized management
- No data caching
  - Little benefit on client: large data sets / streaming reads
  - No need on chunkserver: rely on existing file buffers
  - Simplifies the system by eliminating cache coherence issues
- Familiar interface, but customize the API
  - No Posix: simplify the problem; focus on Google apps
  - Add *snapshot* and *record append* operations



GFS Data

- Data organized in files and directories
  - Manipulation through file handles
- Files stored in chunks (c.f. “blocks” in disk file systems)
  - A chunk is a Linux file on local disk of a chunkserver
  - Unique 64 bit chunk handles, assigned by master at creation time
  - Fixed chunk size of 64MB
  - Read/write by (chunk handle, byte range)
  - Each chunk is replicated across 3+ chunkservers

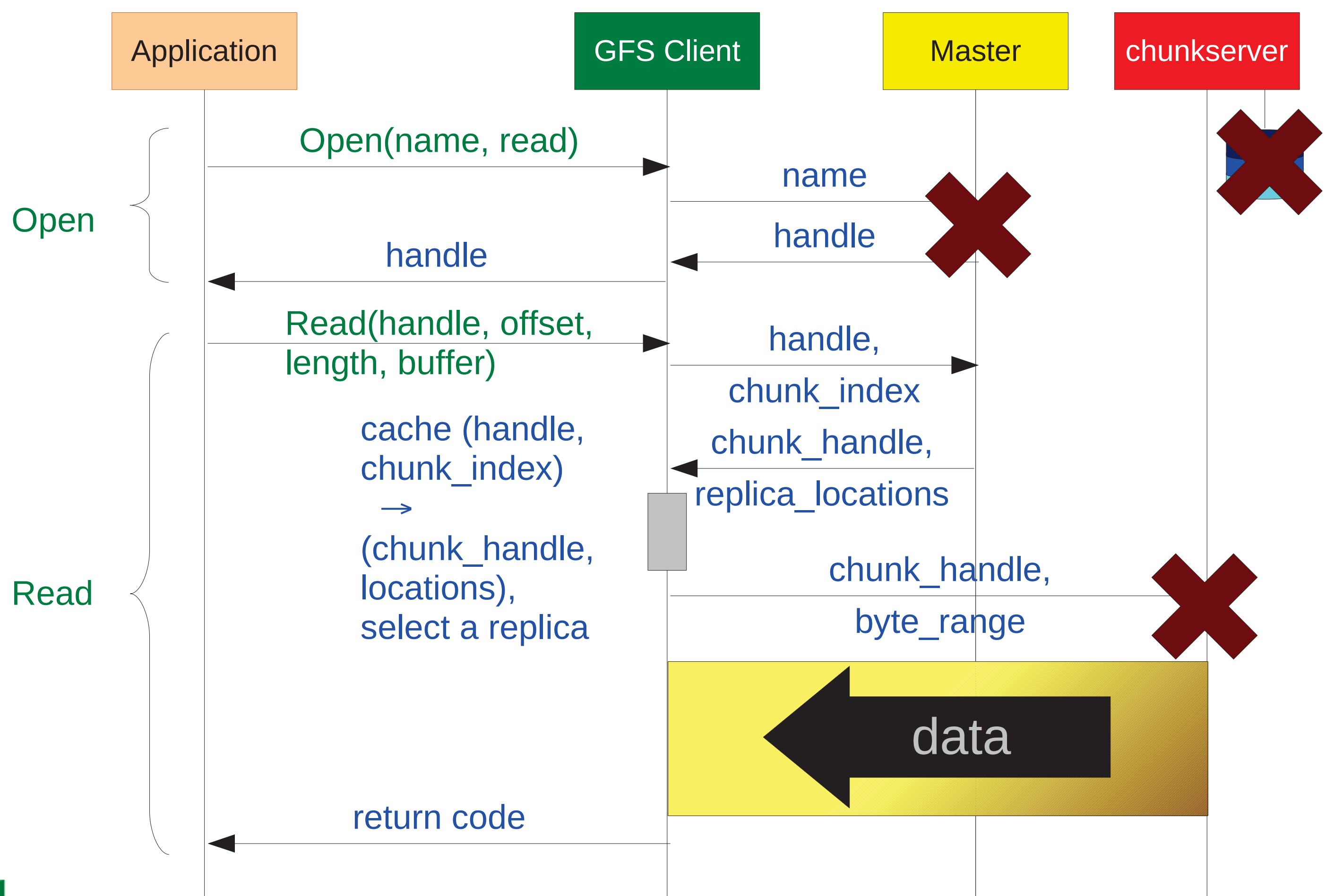
Chunk Size

- 64 MB is much larger than typical file system block sizes
- A large chunk size has the following advantages when stream reading/writing:
  - less communication between client and master
  - less memory space needed for metadata in master
  - less network overhead between client and chunkserver (one TCP connection for larger amount of data)
- … and disadvantages:
  - popular small files (one or few chunks) can become hot spots
    - How to prevent this from happening within GFS or with other architectures?
  - easier to create fragmentation (TODO: illustrate why)

GFS Metadata

- Maintained by the master
  - File namespace and chunk namespace
  - access control information.
  - file-to-chunk mapping and chunk locations
- All in memory during operation
  - about 64 bytes / chunk with 3 replicas
    - Implication on the total storage space in a cluster?
  - Fast and easily accessible
    - How long to fetch the metadata of a chunk?
    - How long if it is on disk?

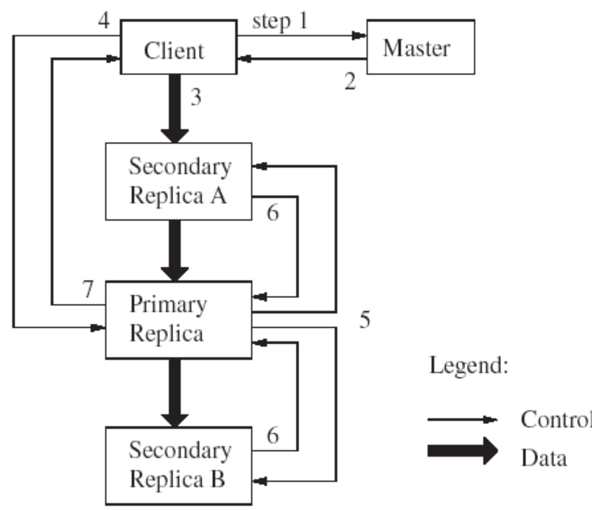
Reading a File



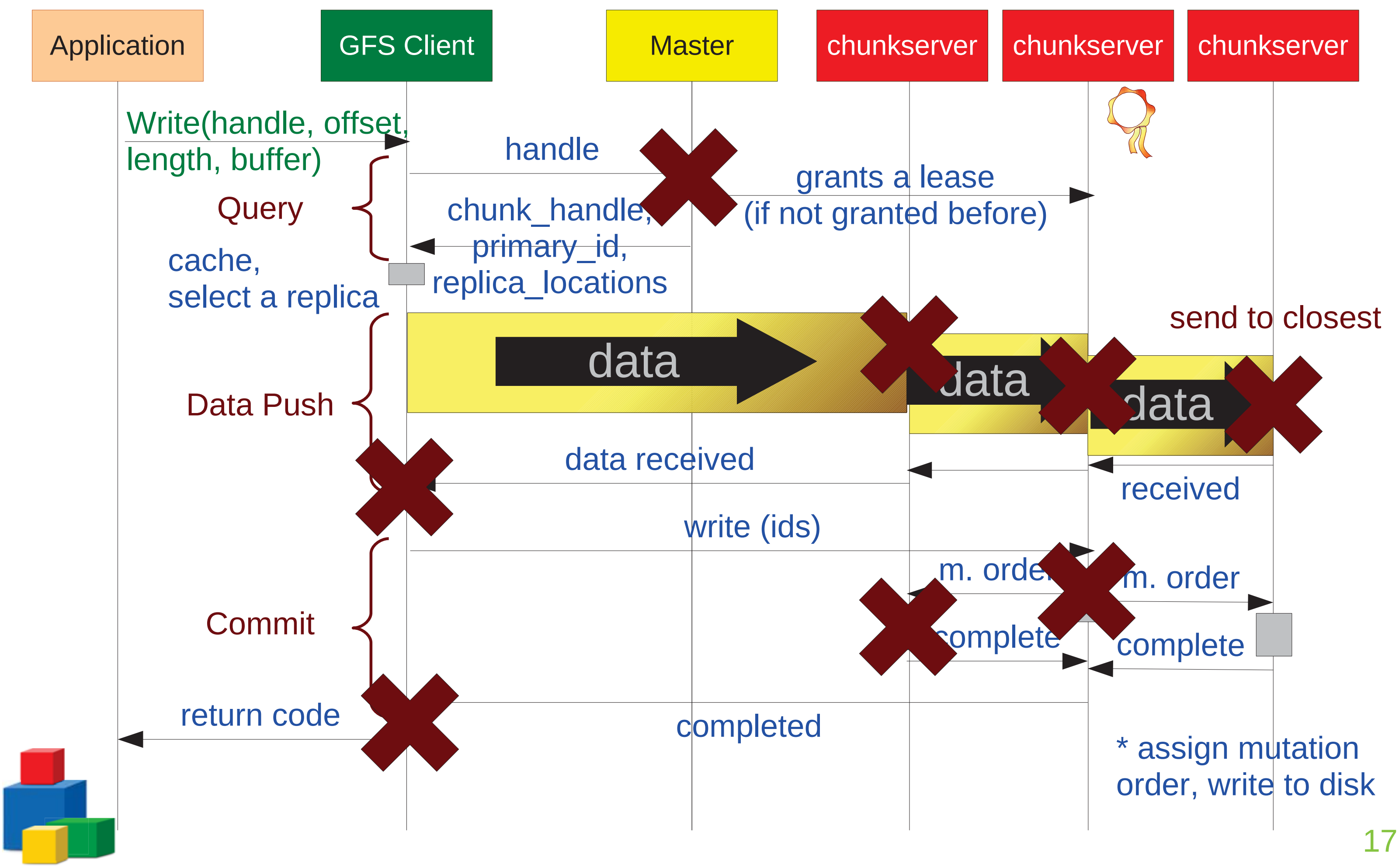
Saving data with GFS: Mutations, Record Appends, and Snapshots.

Mutations

- Mutation = write or append
  - must be done for all replicas
- Goal: minimize master involvement
- Lease mechanism for consistency
  - master picks one replica as primary; gives it a “lease” for mutations
  - a lease = a lock that has an expiration time
  - primary defines a serial order of mutations
  - all replicas follow this order
- Data flow is decoupled from control flow



Writing to a File by One Client



Concurrent Write

- If two clients concurrently write to the same region of a file, any of the following may happen to the overlapping portion:
  - Eventually the overlapping region may contain data from exactly one of the two writes.
  - Eventually the overlapping region may contain a mixture of data from the two writes.
- Furthermore, if a read is executed concurrently with a write, the read operation may see either all of the write, none of the write, or just a portion of the write.

Consistency Model of GFS

- GFS has a relaxed consistency model
- File namespace mutations are atomic and consistent
  - handled exclusively by the master
  - order defined by the operation logs
  - namespace lock guarantees atomicity and correctness
- File region mutations: complicated by replicas
  - “Consistent” = all replicas have the same data
  - “Defined” = consistent + replica reflects the mutation entirely
  - A relaxed consistency model: not always consistent, not always defined, either.

The Consistency Model of GFS

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

Trade-offs

- Some properties:
  - concurrent writes leave region consistent, but possibly undefined
  - failed writes leave the region inconsistent

- Some work has moved into the applications:
  - e.g., self-validating, self-identifying records

Record Append

- Client specifies data, but not the offset
- GFS guarantees that the data is *appended* to the file *atomically at least once*
  - GFS picks the offset, and returns the offset to client
  - works for concurrent writers
- Used heavily by Google apps
  - e.g., for files that serve as multiple-producer/single-consumer queues

How Record Append Works

- Query and Data Push are similar to write operation
- Client send write request to primary
- If appending would exceed chunk boundary
  - Primary pads the current chunk, tells other replicas to do the same, replies to client asking to retry on the next chunk.
  - Question: (fragmentation and space utilization efficiency) v.s. max record size
- Else: commit the write in all replicas
- Any replica failure: client retries.

Snapshot

- Makes a copy of a file or a directory tree almost instantaneously (“snapshot /home/foo /save/foo”)
  - minimize interruptions of ongoing mutations
  - copy-on-write with reference counts on chunks
- Steps:
  1. a client issues a snapshot request for source files
  2. master revokes all leases of affected chunks (what happens to ongoing writes?)
  3. master logs the operation to disk
  4. master duplicates metadata of source files, pointing to the same chunks, increasing the reference count of the chunks

Writing to a snapshotted chunk

- A client issues request to find the lease holder(s) of the byte range of a file
- Master find that reference count > 1 for chunk C
  - Master picks a new chunk handle C'
  - Master asks all chunservers having C to create chunk C' and copy contents from C
    - Why ask the same chunkservers? Pros? Cons?
  - Master grants lease to a chunkserver for chunk C'
  - Master replies chunk handle C' to client
- Q: where should the reference counts be stored?

Single Master

- The master have global knowledge of chunks
  - easy to make decisions on placement and replication
- From distributed systems we know this is a:
  - Single point of failure
  - Scalability bottleneck
- GFS solutions:
  - Shadow masters
  - Minimize master involvement
    - never move data through it, use only for metadata
    - cache metadata at clients
    - large chunk size
    - master delegates authority to primary replicas in data mutations (chunk leases)

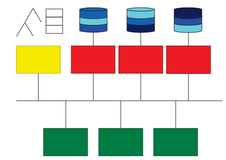
Master's responsibilities

- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
- Chunk creation, re-replication, rebalancing
- Garbage Collection
- Stale replica deletion
- A "director" that moves no data by itself

Heatbeat

- Master issues HeartBeat messages to chunkservers regularly
  - “ping”: too many strikes, and you're out.
  - give instructions: revoke/extend lease, delete chunk, etc.
  - collect chunk status: corrupt, possessed, etc.
- A primary sends lease extension request in HeartBeat reply, and get extension in the next HeartBeat if granted.
- A chunkserver sends some chunk IDs that it has, and get orphaned chunks in reply.
- A chunkserver sends corrupt chunk ID
- How often should a master send HeatBeats to a chunkserver?

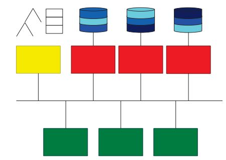
Lease Management



- A crucial part of concurrent write/append operation
  - Designed to minimize master's management overhead by authorizing chunkservers to make decisions
- One lease per chunk
  - Granted to a chunkserver, which becomes the primary
  - Granting a lease increases the version number of the chunk
  - Reminder: the primary decides the mutation order
- The primary can renew the lease before it expires (default 60 seconds)
  - Piggybacked on the regular heartbeat message
- The master can revoke a lease (e.g., for snapshot)
- The master can grant the lease to another replica if the current lease expires (primary crashed, etc)

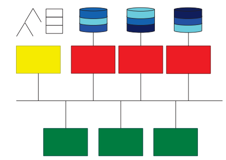
Namespace Management and Locking

- GFS have no per-directory data structure, no links
  - Flat lookup table of path to metadata
  - Use prefix compression in memory
  - A map of path to read-write locks
- Manipulating path /d1/d2/.../dn/leaf acquires read-lock on /d1, /d1/d2, ..., /d1/d2/.../dn and read or write lock on the path
  - “snapshot /user/home /save/home” acquires read lock for /user and /save, write lock on /user/home and /save/home.
    - Why write-lock on /user/home?
  - allows concurrent mutations in the same directory



Replica Placement

- Traffic between racks is slower than within the same rack
- A replica is created for 3 reasons
  - chunk creation
  - chunk re-replication
  - chunk rebalancing
- Master has a replica placement policy
  - Maximize data reliability and availability
  - Maximize network bandwidth utilization
    - per host, per rack switch
  - Conclusion: must spread replica across racks



Chunk Creation

- Where to put the initial replicas?
  - servers with below-average disk utilization
    - for equalization of space
  - but not too many recent creations on a server
    - why?
  - and must have servers across racks

Chunk Re-replication

- Master clones a chunk as soon as # of replicas is below the *goal*, default goal = 3.
  - Chunkserver dies, is removed, etc. Disk fails, is disabled, etc. Chunk is corrupt. Goal is increased.
- Factors affecting which chunk is cloned first:
  - how far is it from the goal
  - live files vs. deleted files
  - blocking client
- Placement policy is similar to chunk creation
- Master limits number of cloning per chunkserver and cluster-wide to minimize impact to client traffic
- Chunkserver throttles cloning read.

Chunk Rebalance

- Master rebalances replicas periodically
  - Moves chunks for better disk space balance and load balance
  - Fills up new chunkserver
    - why not fill it up immediately so space is balanced sooner?
- Master prefers to move chunks out of crowded chunkserver

How to Delete a File?

- Replica deletion messages may be lost
  - need detection and handshake mechanisms
  - Should a client wait for deletion to complete?
- Chunk creation may fail on some chunkservers, leaving chunks unknown to master (storage leak)
  - Need a way to claim the storage space
- How to undelete a file?

Garbage Collection

- Chunks of deleted files are not reclaimed immediately
- Mechanism:
  - Client issues a request to delete a file
  - Master logs the operation immediately, renames the file to a hidden name with timestamp, and replies.
  - Master scans file namespace regularly
    - Master removes metadata of hidden files older than 3 days
  - Master scans chunk namespace regularly
    - Master removes metadata of orphaned chunks
  - Chunkserver sends master a list of chunk handles it has in regular HeartBeat message
    - Master replies the chunks not in namespace
    - Chunkserver is free to delete the chunks

Discussion on Garbage Collection

- Requests to delete a file is returned promptly
  - Master's regular scan of file and chunk namespaces are done when it is relatively free, so master can be more responsive to clients
- Storage reclamation is done in regular background processes of the master with regular HeartBeats
- renaming + 3 day grace period gives an easy way to undelete
- Disadvantage: quota

Chunk Version and Stale Replica Deletion

- Stale replica = a replica that misses mutation(s) while the chunkserver is down
  - Server reports its chunks to master after booting. Oops!
- Solution: chunk version number
  - Master and chunkservers keep chunk version numbers persistently.
  - Master creates new chunk version number when granting a lease to primary, and notifies all replicas, then store the new version persistently.
    - Unreachable chunservers miss the new version number
    - Question: why not store before granting the lease?
- Master sends version number along to client / replicator for verification at read time.

Metadata Operations

Metadata: Persistence

- Namespace and file-to-chunk mapping are kept persistent
  - *operation logs + checkpoints*
- Operation logs = historical record of mutations
  - represents the timeline of changes to metadata in concurrent operations.
  - stored on master's local disk
  - replicated remotely
- A mutation is not done or visible until the operation log is stored locally and remotely.
  - master may group operation logs for batch flush.

Metadata: Chunk Locations

- Chunk locations are not stored in master's disks
  - The master asks chunkservers what they have during master startup or when a new chunkserver joins the cluster.
  - It decides chunk placements thereafter.
  - It monitors chunkservers with regular heartbeat messages.
- Rationale:
  - Disks fail.
  - Chunkservers die, (re)appear, get renamed, etc.
  - Eliminate synchronization problem between the master and all chunkservers.
  - Chunkservers have the final say anyway.

Metadata: Recovery

- Recover the file system = replay the operation logs.
  - "fsck" of GFS after, e.g., a master crash.
- Use checkpoints to speed up
  - memory-mappable, no parsing.
  - Recovery = read in the latest checkpoint + replay logs taken after the checkpoint.
  - Incomplete checkpoints are ignored.
  - Old checkpoints and operation logs can be deleted.
- Creating a checkpoint: must not delay new mutations
  1. Switch to a new log file for new operation logs: all operation logs up to now are now "frozen".
  2. Build the checkpoint in a separate thread.
  3. Write locally and remotely.

Fault Tolerance

Fault Tolerance

- The system must function through component failures
- High availability
  - fast recovery
  - chunk replication: 3 replicas by default, with cloning mechanism in the background.
  - master replication
- Data integrity
- Diagnostic tools

High availability: Fast Recovery

- Master and chunkserver can start and restore to previous state in seconds
  - metadata is stored in binary format, no parsing
  - 50MB – 100 MB of metadata per server
  - normal startup and startup after abnormal termination is the same
  - can kill the process anytime

High availability: master replication

- Master's operation logs and checkpoints are replicated on multiple machines.
  - A mutation is complete only when all replicas are updated
- If the master dies, cluster monitoring software starts another master with checkpoints and operation logs.
  - Clients see the new master as soon as the DNS alias is updated
- Shadow masters provide read-only access
  - Reads a replica operation log to update the metadata
  - Typically behind by less than a second
  - No interaction with the busy master except replica location updates (cloning)

Data Integrity

- A responsibility of chunkservers, not master
  - Disks failure is norm, chunkserver must know
  - GFS doesn't guarantee identical replica, independent verification is necessary.
- 32 bit checksum for every 64 KB block of data
  - available in memory, persistent with logging
  - separate from user data
- Read: verify checksum before returning data
  - mismatch: return error to client, report to master
  - client reads from another replica
  - master clones a replica, tells chunkserver to delete the chunk

Diagnostic Tools

- Logs on each server
  - Significant events (server up, down)
  - RPC requests/replies
- Combining logs on all servers to reconstruct the full interaction history, to identify source of problems.
- Logs can be used on performance analysis and load testing, too.

Performance

Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Cluster	A	B
Read rate (last minute)	583 MB/s	380 MB/s
Read rate (last hour)	562 MB/s	384 MB/s
Read rate (since restart)	589 MB/s	49 MB/s
Write rate (last minute)	1 MB/s	101 MB/s
Write rate (last hour)	2 MB/s	117 MB/s
Write rate (since restart)	25 MB/s	13 MB/s
Master ops (last minute)	325 Ops/s	533 Ops/s
Master ops (last hour)	381 Ops/s	518 Ops/s
Master ops (since restart)	202 Ops/s	347 Ops/s

Deployment in Google

- Many GFS clusters
- hundreds/thousands of storage nodes each
- Managing petabytes of data
- GFS is the foundation of BigTable and MapReduce, etc.

Conclusion

- GFS demonstrates how to support large-scale processing workloads on commodity hardware
  - designed to tolerate frequent component failures
  - uniform logical namespace
  - optimize for huge files that are mostly appended and read
  - feel free to relax and extend FS interface as required
  - relaxed consistency model
  - go for simple solutions (e.g., single master, garbage collection)

- GFS has met Google’s storage needs.

Discussion

- How many sys-admins does it take to run a system like this?
  - much of management is built in
- Is GFS useful as a general-purpose commercial product?
  - small write performance not good enough?
  - relaxed consistency model: can your app live with it?

Homework

- Implement the metadata operation of snapshot and write.
  - There is no chunk structure for simplicity
  - Load TA's metadata into memory
  - Accept commands from port 9000: “write filename” and “snapshot filename1 filename2” (no directory snapshots)
  - “write filename”, return the file handle of the file, taking into account whether the file has been snapshotted
  - “snapshot”, make necessary changes to metadata, return 0 as success
- Bonus points: prefix compression of metadata in memory
- Super bonus points: read-write lock so snapshot revokes current writes and blocks future writes until it's done.