

DS256:Jan18 (3:1)

L1:Google File System

Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung
ACM SOSP, 2003

Yogesh Simmhan

9,11,16 Jan, 2018

©Department of Computational and Data Science, IISc, 2016

This work is licensed under a [Creative Commons Attribution 4.0 International License](https://creativecommons.org/licenses/by/4.0/)

Copyright for external content used with attribution is retained by their original authors



Credits

- The following slides are based on:
 - “Distributed Filesystems”, CSE 490h – Introduction to Distributed Computing, Spring 2007, © Aaron Kimball
 - <https://courses.cs.washington.edu/courses/cse490h/07sp/>
 - Google File System, Alex Moshchuk



File Systems Overview

- System that permanently stores data
- Usually layered on top of a lower-level physical storage medium
- Divided into logical units called “files”
 - Addressable by a *filename* (“foo.txt”)
 - Usually supports hierarchical nesting (directories)
- A file *path* joins file & directory names into a **relative** or **absolute** address to identify a file (“/home/aaron/foo.txt”)



What Gets Stored

- User data itself is the bulk of the file system's contents
- Also includes *meta-data* on a drive-wide and per-file basis:

Drive-wide:

Available space

Formatting info

character set

...


Per-file:

name

owner

modification date

physical layout...



High-Level Organization

- Files are organized in a “tree” structure made of nested directories
- One directory acts as the “root”
- “links” (symlinks, shortcuts, etc) provide simple means of providing multiple access paths to one file
- Other file systems can be “mounted” and dropped in as sub-hierarchies (other drives, network shares)



Low-Level Organization (1/2)

- File data and meta-data stored separately
- File descriptors + meta-data stored in *inodes*
 - Large tree or table at designated location on disk
 - Tells how to look up file contents
- Meta-data may be replicated to increase system reliability



Low-Level Organization (2/2)

- “Standard” read-write medium is a hard drive (other media: CDROM, tape, ...)
- Viewed as a sequential array of blocks
- Must address ~1 KB chunk at a time
- Tree structure is “flattened” into blocks
- Overlapping reads/writes/deletes can cause fragmentation: files are often not stored with a linear layout
 - inodes store all block numbers related to file



Fragmentation

A	B	C	(free space)
---	---	---	--------------

A	B	C	A	(free space)
---	---	---	---	--------------

A	(free space)	C	A	(free space)
---	--------------	---	---	--------------

A	D	C	A	D	(free)
---	---	---	---	---	--------



Design Considerations

- Smaller inode size reduces amount of wasted space
- Larger inode size increases speed of sequential reads (may not help random access)
- Should the file system be **faster** or **more reliable**?
- But faster at what: Large files? Small files? Lots of reading? Frequent writers, occasional readers?



Distributed Filesystems

- Support access to files on remote servers
- Must support concurrency
 - Make varying guarantees about locking, who “wins” with concurrent writes, etc...
 - Must gracefully handle dropped connections
- Can offer support for replication and local caching
- Different implementations sit in different places on complexity/feature scale



GFS



Motivation

- Google needed a good distributed file system
 - Redundant storage of massive amounts of data on cheap and unreliable computers
- 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...*seeds for the “Cloud” on commodity clusters*
- “Modest” number of **HUGE** files
 - Just a few million
 - Each is 100MB or larger; multi-GB files typical
- Files are **write-once**, mostly appended to
 - Perhaps *concurrently*
 - *Example?*
- Large streaming reads, sequential reads
- High sustained **throughput** favored over **low latency**

Web Search

■ Crawl the web

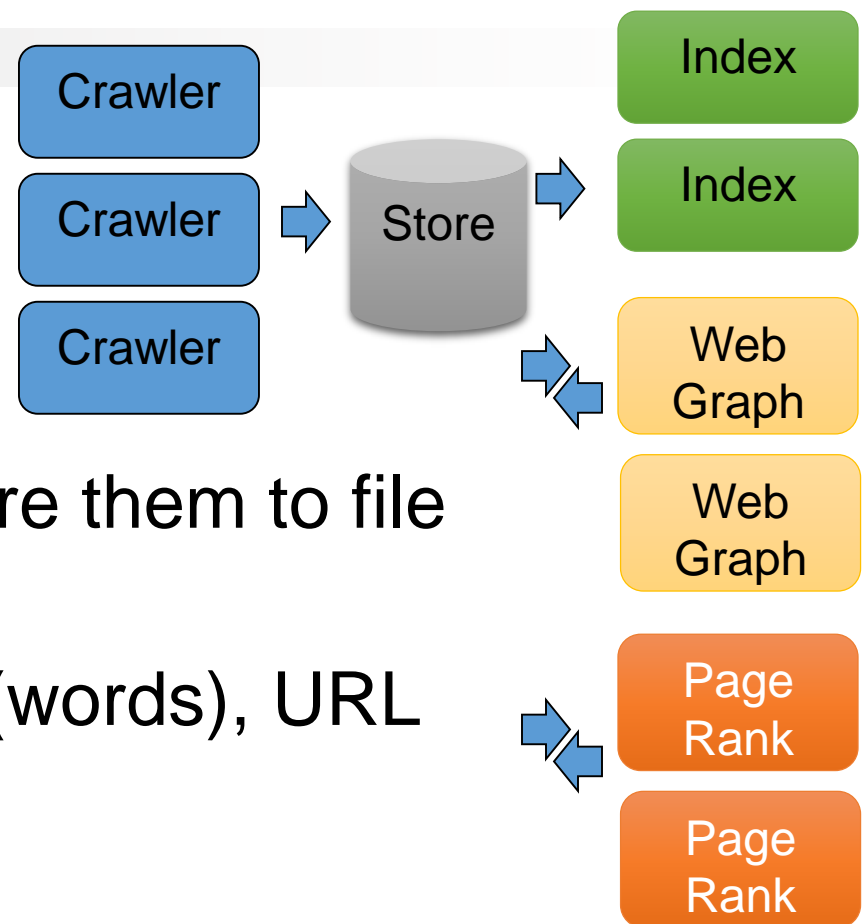
- Downloads URLs, store them to file system
- Extract page content (words), URL links
- Do BFS traversal

■ Index the text content

- Inverted index: Word->URL

■ Rank the pages

- PageRank



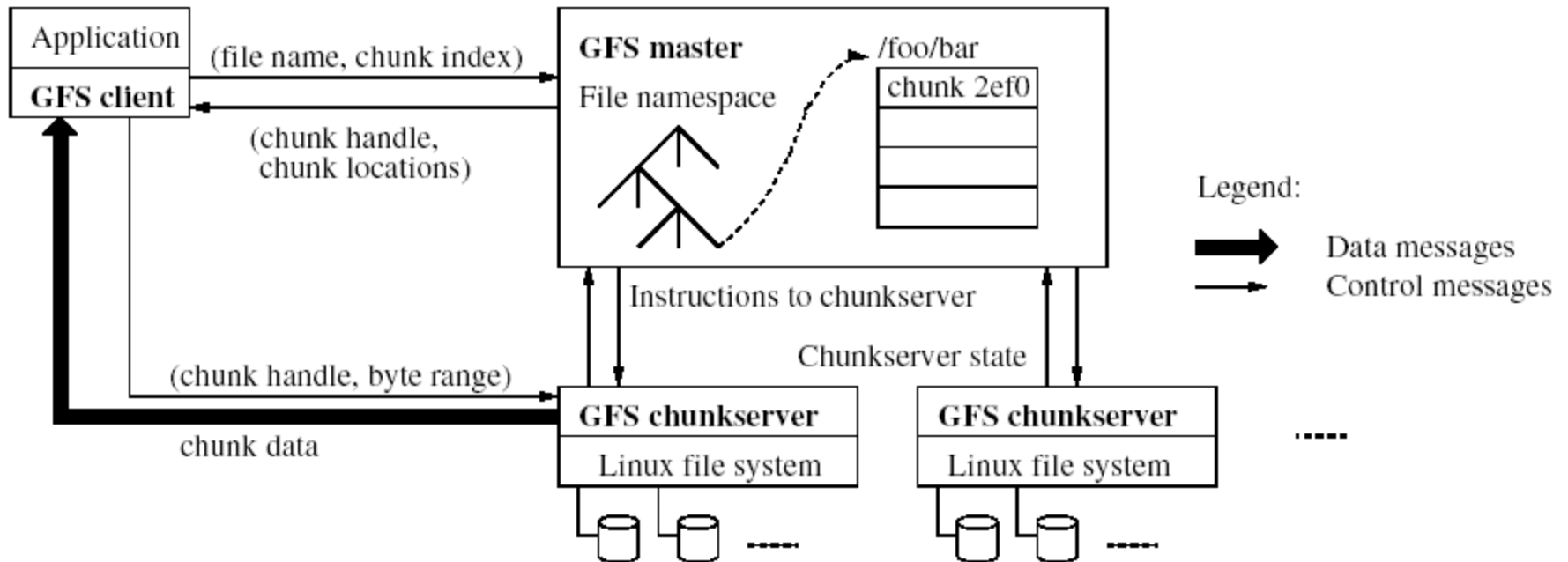


GFS Design Decisions

- Files stored as chunks
 - Fixed size (64MB)
- Reliability through replication
 - Each chunk replicated across 3+ *chunkservers*
- Single master to coordinate access, keep metadata
 - Simple centralized management, placement decisions
- No data caching
 - Little benefit due to large data sets, streaming reads
- Familiar interface, but customize the API
 - Simplify the problem; focus on Google apps
 - Add *snapshot* and *record append* operations

GFS Architecture

- Single master
- Multiple chunkservers



...Can anyone see a potential weakness in this design?



Single master

- Downsides

- Single point of failure
 - Scalability bottleneck

- GFS solutions:

- Shadow masters
 - Minimize master involvement
 - never move data through it, use only for metadata
 - and cache metadata at clients
 - large chunk size
 - master delegates authority to primary replicas in data mutations (chunk leases)



Chunks

- Stored as a Linux file on chunk servers
- Default is 64MB...larger than file system block sizes
 - Pre-allocates disk space, avoids fragmentation
 - Reduces master metadata
 - Reduces master interaction, allows metadata caching on client
 - Persistent TCP connection to chunk server
- Con: hot-spots...need to tune replication



Metadata

- Global metadata is stored on the master
 - File and chunk namespaces
 - Mapping from files to chunks
 - Locations of each chunk's replicas
- All kept **in-memory** (64 bytes / chunk)
 - Fast
 - Easily accessible
 - Cheaper to add more memory



Metadata

- Master builds *chunk-to-server* mapping on boot-up, from chunk servers
 - And on heart-beat
- Avoids sync of chunkservers & master
 - Chunkservers can come and go, fail, restart often
 - Gives flexibility to manage chunkservers independently
 - Lazy propagation to master
- Chunk server is *final authority* on whether it has chunk or not



Metadata

- Metadata should be consistent, and updates visible to clients only after stable
 - Once metadata is visible, clients can use/change it
- Master has an *operation log* for persistent logging of critical metadata updates
 - Namespace & file-chunk mapping persisted
 - Persistent on local disk as log mutations
 - **Logical time**, order of concurrent ops, single master!
 - Replicated, client ack only after replication
 - Replay log to recover metadata
 - Checkpoints for faster recovery



Chunk Servers

- Master controls chunk placement
- Chunk servers store chunks
- Interacts with clients for file operations
- Heart beat messages to master
 - liveliness, list of chunks present



Master's responsibilities

- Namespace management/locking
- Metadata storage
 - Periodic scanning of in-memory metadata
- Periodic communication with chunkservers
 - give instructions, collect state, track cluster health



Chunk Management

- Chunk creation, re-replication, rebalancing
 - balance space utilization, bandwidth utilization and access speed/performance
 - spread replicas across racks to reduce correlated failures...across servers, racks
 - re-replicate data if redundancy falls below threshold
 - rebalance data to smooth out storage and request load



Master's responsibilities

■ Garbage Collection

- simpler, more reliable than traditional file delete
- master logs the deletion, renames the file to a hidden name
- lazily garbage collects hidden files

■ Stale replica deletion

- detect “stale” replicas using chunk version numbers



Fault Tolerance

- High availability

- fast recovery

- master and chunkservers restartable in a few seconds

- chunk replication

- default: 3 replicas.

- Replication for fault tolerance vs. Replication for performance

- shadow masters

- Data integrity

- checksum every 64KB block in each chunk



Relaxed consistency model

- Namespace updates atomic and serializable
 - namespace locking guarantees this
- “Consistent” = All replicas have the same value. Any client reads same value.
- “Defined” = Consistent, and all replicas reflect the mutation performed. All clients read the *updated* value in entirety.



Relaxed consistency model

- Single successful write leaves region defined
- Concurrent writes leave region consistent, but possibly undefined
 - All clients see same content, but writes may have happened in fragments/interleaved
- Failed writes leave the region inconsistent
 - Different clients see different content at different times



Mutations

- Mutation = write or append
 - **Write** to a specific *offset*
 - **Append** data, *atomically at least once*
 - Must be done for all replicas
 - Returns start offset for defined region
 - May *pad inconsistent regions, Create duplicates*
- Mutated file guaranteed to be defined after successful mutations...
 - Apply mutations to a chunk in the same order on all its replicas
 - Use chunk version numbers to detect any stale replica that was down during a mutation
 - *Caching of metadata in client can cause stale reads!*

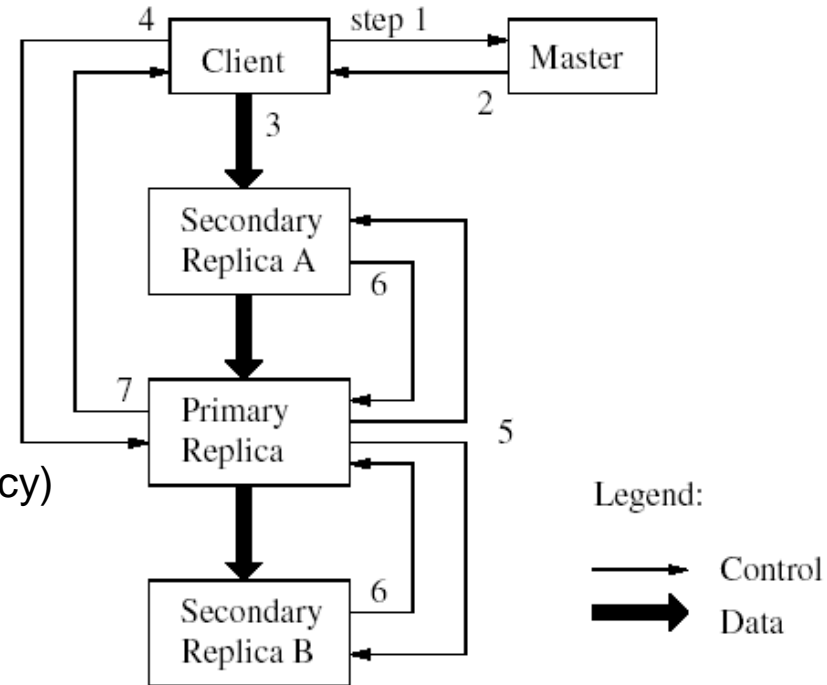


Mutations

- Goal: Minimize master involvement
- Lease mechanism for a mutation
 - master picks one replica as primary; gives it a “lease” for mutations from *any* client (default 60 secs, extensible as part of heartbeat)
 - primary defines a serial order of mutations
 - all replicas follow this order

Mutations

- Dataflow decoupled from control flow
 - Forwarding to closest server. Pipeline.
 - Fully utilize bandwidth!
 - B/T + R.L (Bytes, Thruput, Replicas, Latency)
- Send record to all replicas, any order. Maintained in buffer by c'server.
- Send “write” to primary with data IDs. Primary assigns serial numbers to records. Applies mutations in serial order.
- Forwards to secondary's. Applied on them, and acked to primary. Acked to client.
- Any failures replica leaves inconsistent state. Retry.
- Mutations across chunks are split by client.





Mutations

- Clients need to distinguish between defined and undefined regions
- Some work has moved into the applications:
 - e.g., prefer appends, use self-validating, self-identifying records, checkpoint
 - Use checksums to identify and discard padding
 - Use uids to identify duplicate records
- Simple, efficient
 - Google apps can live with it
 - what about other apps?



Atomic record append

- Concurrent writes to same region not serializable
- In Append, client specifies data. GFS appends it to the file atomically at least once
 - GFS picks the offset. Returns offset to client.
 - works for concurrent writers
- Client sends data to all replicas of last chunk of file.
- Primary applies to its end; tells secondaries to apply at that offset.
 - E.g. concurrent appends from different clients...
- Used heavily by Google apps
 - e.g., for files that serve as multiple-producer/single-consumer queues



Snapshot

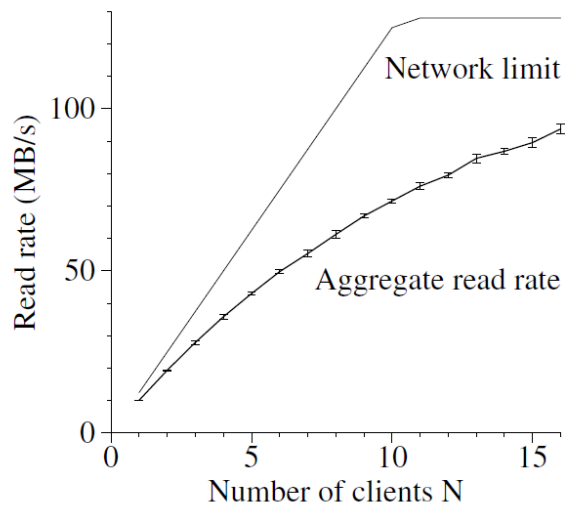
- Full copies of a file, rapidly
- On snapshot request, NameNode revokes all primary leases to chunks
- Log opn to disk. Copy metadata to new file, chunks point to old chunks with reference count +1
- Copy old chunk to new on write (copy on write to local chunkserver) if reference counter > 1

Performance

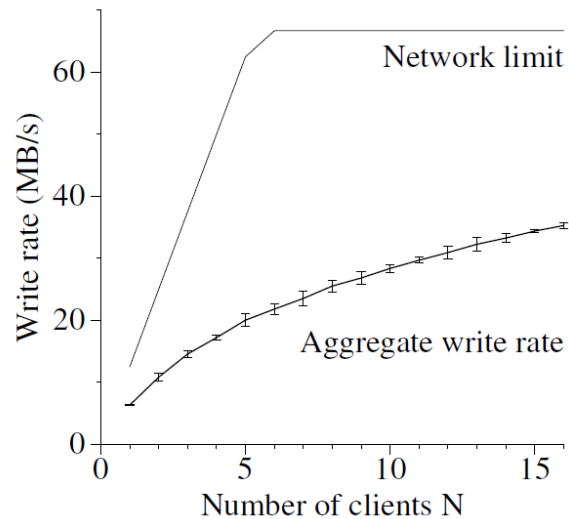
Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

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

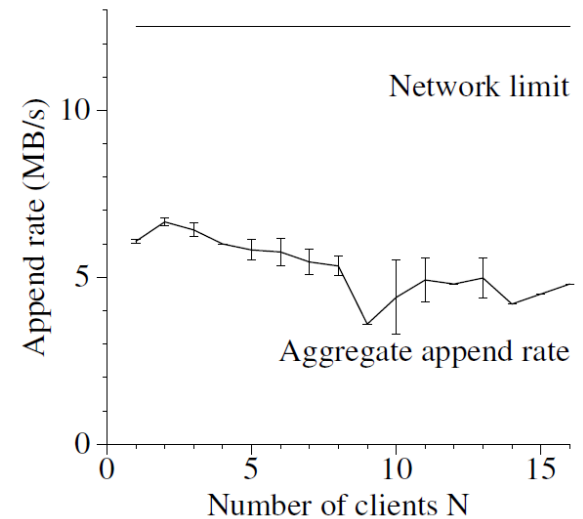
Performance



(a) Reads



(b) Writes



(c) Record appends



Deployment in Google

- 50+ GFS clusters
- Each with thousands of storage nodes
- Managing petabytes of data
- GFS is under BigTable, etc.



Conclusion

- GFS demonstrates how to support large-scale processing workloads on commodity hardware
 - design to tolerate frequent component failures
 - optimize for huge files that are mostly appended and read
 - feel free to relax and extend FS interface as required
 - go for simple solutions (e.g., single master)
- GFS has met Google's storage needs... it must be good!

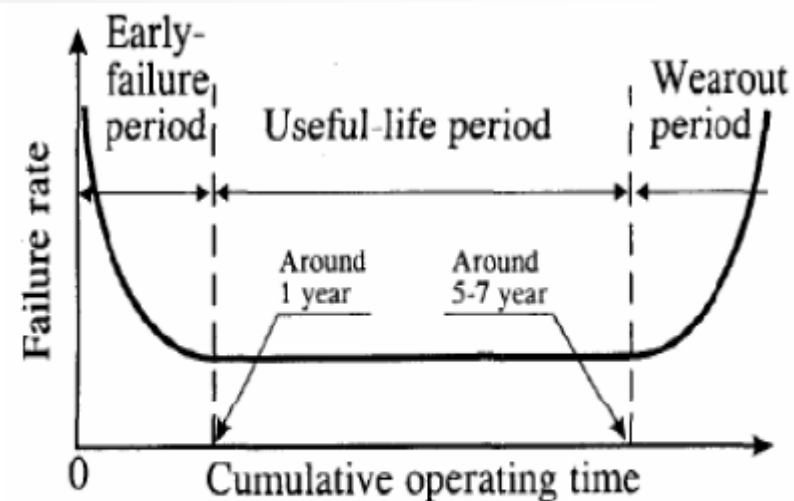
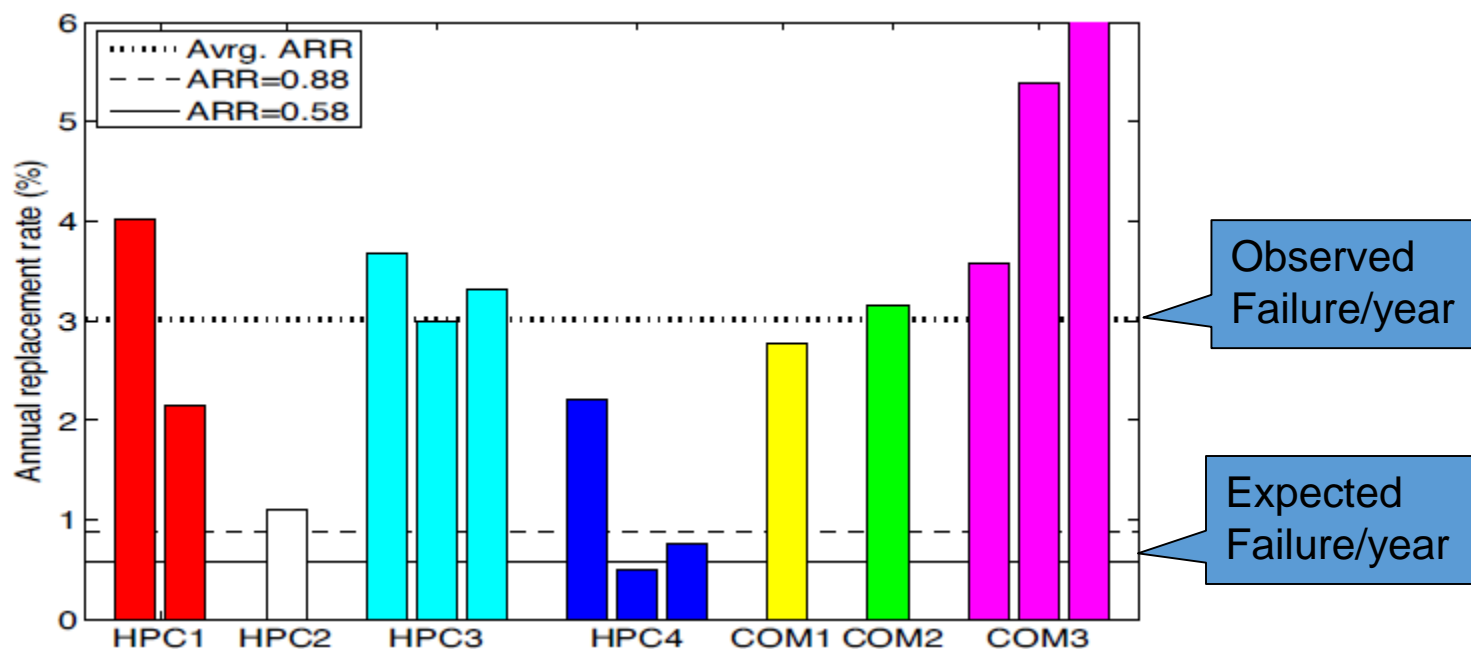


Figure 2: Lifecycle failure pattern for hard drives [33].



Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?, Bianca Schroeder
Garth A. Gibson Usenix FAST 2007



Hadoop Distributed File System

Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler,
IEEE Symposium on Mass Storage Systems and Technologies (MSST),
2010



Block Report

- DataNode identifies block replicas in its possession to the NameNode by sending a *block report*
 - block id, the generation stamp and the length
- First block report is sent immediately after the DataNode registration. Subsequent block reports are sent every hour.



Heartbeat

- DataNodes send *heartbeats* to the NameNode to confirm that it is up
- Default interval is 3 seconds. After no heartbeat in 10 minutes the NameNode considers it out of service, block replicas unavailable. Schedules replication.
- Piggyback storage capacity, fraction of storage in use, data transfers in progress
- NameNode responds to heartbeat with:
 - Replicate blocks, remove local replicas, shut down the node, send immediate block report



Checkpoint Node

- Can take hours to recreate NameNode for 1 week of journal (ops log)
- Periodically combines existing checkpoint & journal to create a new checkpoint & empty journal
- Download current checkpoint & journal from the NameNode, merges them locally, return new checkpoint to NameNode
- New checkpoint lets NameNode truncate the tail of the journal



Backup Node

- BackupNode can create periodic checkpoints.
Read-only NameNode!
- Also maintains an in-memory image of namespace, synchronized with NameNode
- Accepts the journal stream of namespace transactions from active NameNode, saves them to its local store, applies them to its own namespace image in memory
- Creating checkpoints is done locally

Block Creation Pipeline

- The DataNodes form a pipeline, the order of which
- minimizes the total network distance from the client to the last DataNode.
- Data is pushed to the pipeline as (64 KB) packet buffers
- Async, Max outstanding acks.
- Clients generates checksums for blocks, DN stores checksums for each block. Checksums verified by client while reading to detect corruption.

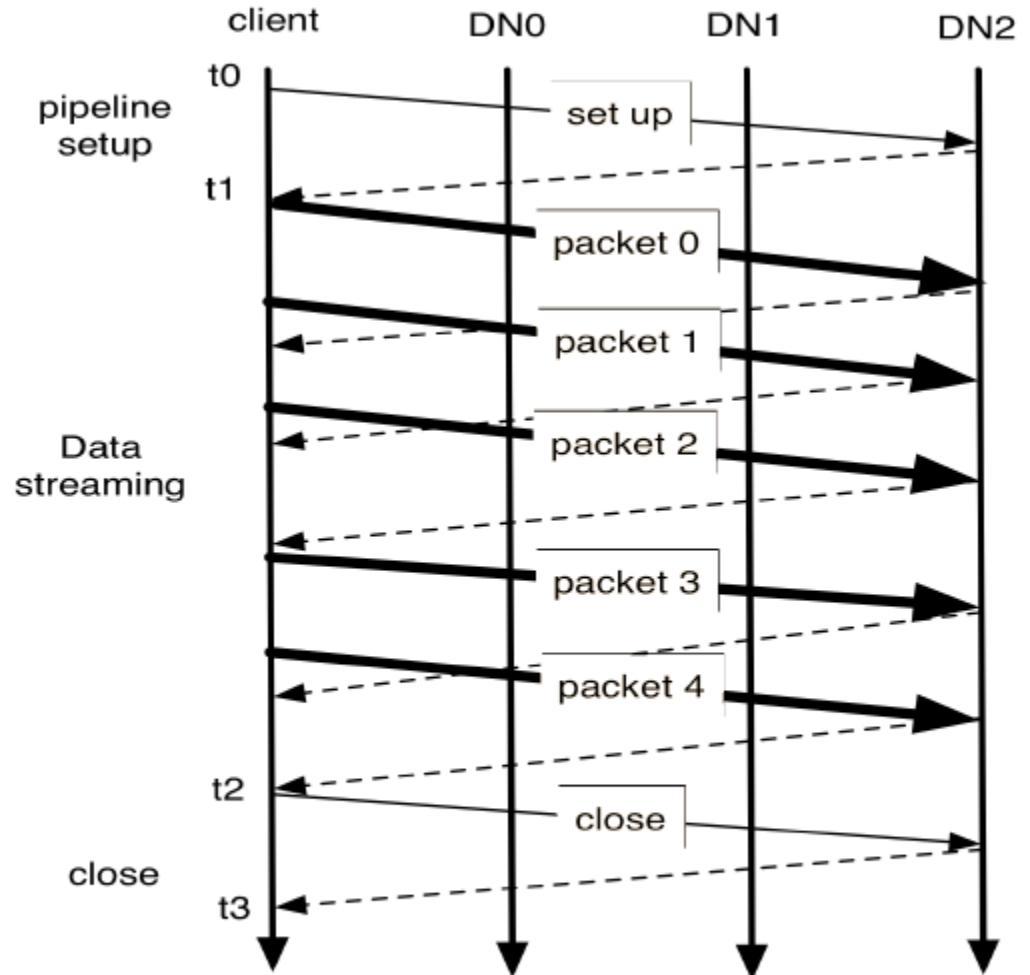
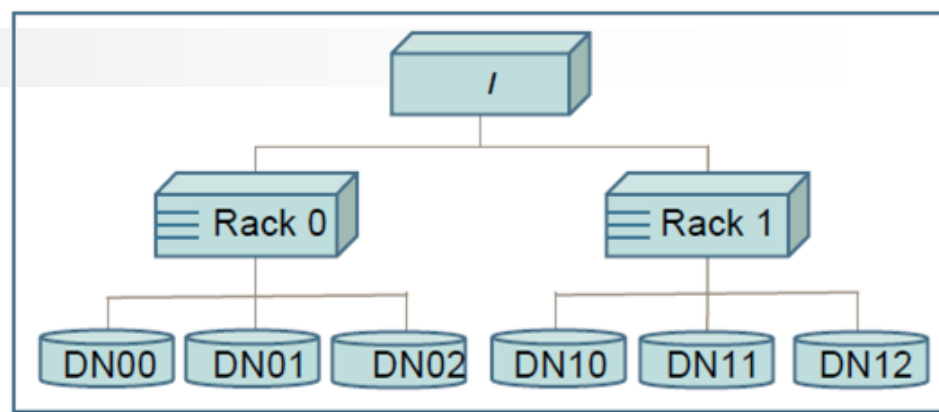


Figure 2. Data pipeline during block construction

Block Placement



- The distance from a node to its parent node is 1. Distance between nodes is sum of distances to their common ancestor.
- Tradeoff between min write cost, and max reliability, availability & agg read B/W
 - 1st replica on writer node, the 2nd & 3rd on different nodes in a different rack
 - No Datanode has more than one replica. No rack has more than two replicas of a block.
- NameNode returns replica location in the order of its closeness to the reader



Replication

- NameNode detects under- or over-replication from block report
- Remove replica without reducing the # of racks hosting replicas, prefer DataNode with least disk space
- Under-replicated blocks put in priority queue. Block with 1 replica has highest priority.
- Background thread scans the replication queue, decide where to place new replicas



Balancer

- Balances disk space usage on an HDFS cluster based on threshold
 - Utilization of a node (used%) differs from the utilization of cluster by no more than the threshold value.
- Iteratively moves replicas from nodes with higher utilization to nodes with lower.
 - Maintains data availability, minimizes inter-rack copying, limits bandwidth consumed



Discussion

- How many sys-admins does it take to run a system like this?
 - much of management is built in
- Google currently has ~450,000 machines
 - GFS: only 1/3 of these are “effective”
 - that’s a lot of extra equipment, extra cost, extra power, extra space!
 - GFS has achieved availability/performance at a very low cost, but can you do it for even less?
- Is GFS useful as a general-purpose commercial product?
 - small write performance not good enough?
 - relaxed consistency model