The Google File System

By Ghemawat, Gobioff and Leung

Outline

- Overview
- Assumption
- Design of GFS
- System Interactions
- Master Operations
- Fault Tolerance
- Measurements

Overview

- GFS: Scalable distributed file system for large distributed data-intensive applications

Assumption

- Built from inexpensive commodity components that often fail
- System stores modest number of large files.
  - Few million files, each 100MB+. Multi-GB common. Small files supported but need not optimize for.

Assumption

- Typical workload Read pattern
  - Large streaming reads
    - Individual operations read hundreds of KBs, more commonly 1MB or more.
    - Succesive operations from same client often read through contiguous region of file.
  - Small random reads
    - Read a few KB at some random offset.
    - Performance-conscious apps often batch small reads.

Assumption

- Typical workload Write pattern
  - Many large sequential writes that append data to files.
    - Once written, files seldom modified again.
    - Small writes at arbitrary positions supported, not efficient. (not often in typical workload)

Design of GFS

- Multiple standalone and independent “clusters”
- Each cluster includes a master and multiple chunkservers
- Cluster accessed by multiple clients.
- Can run chunkserver and client on same machine.
- Files divided into fixed-size chunks.
    - Each chunk is 64MB big
- Each file has 64 bit chunk handle assigned by master at time of chunk creation.
    - The addressable range of a GFS cluster is
        - 2^{64} * 64MB = 1125899906842624 TB

Design of GFS

- Master
  - maintains all file system metadata.
    - Namespace, access control info, files → chunks mapping, locations of chunks.
  - Manages chunk lease management, garbage collection of orphaned chunks, chunk migration.
  - Periodically communicates with each chunkserver in *HeartBeat* messages to give it instructions and collect its state.
- Chunkservers
  - Repositories of chunks in GFS
  - Chunks stored on each chunkserver's local disks as LIINUX files
  - Chunks replicated (default factor of 3).

Design of GFS

- GFS client code linked into each application
- Client does not cache file data
  - Most applications stream through huge files or have working sets too large to be cached
  - No need to worry about Cache coherence issues
- Chunkservers do not cache file data.
  - Most applications stream through huge files or have working sets too large to be cached
  - Chunkservers could benefit from some LINUX buffer caches

Design of GFS

- Fault Toleance Measures
  - (Dynamic) Replication
  - Heartbeat messages
  - Logging
  - Checkpointing / Recovery

Design of GFS

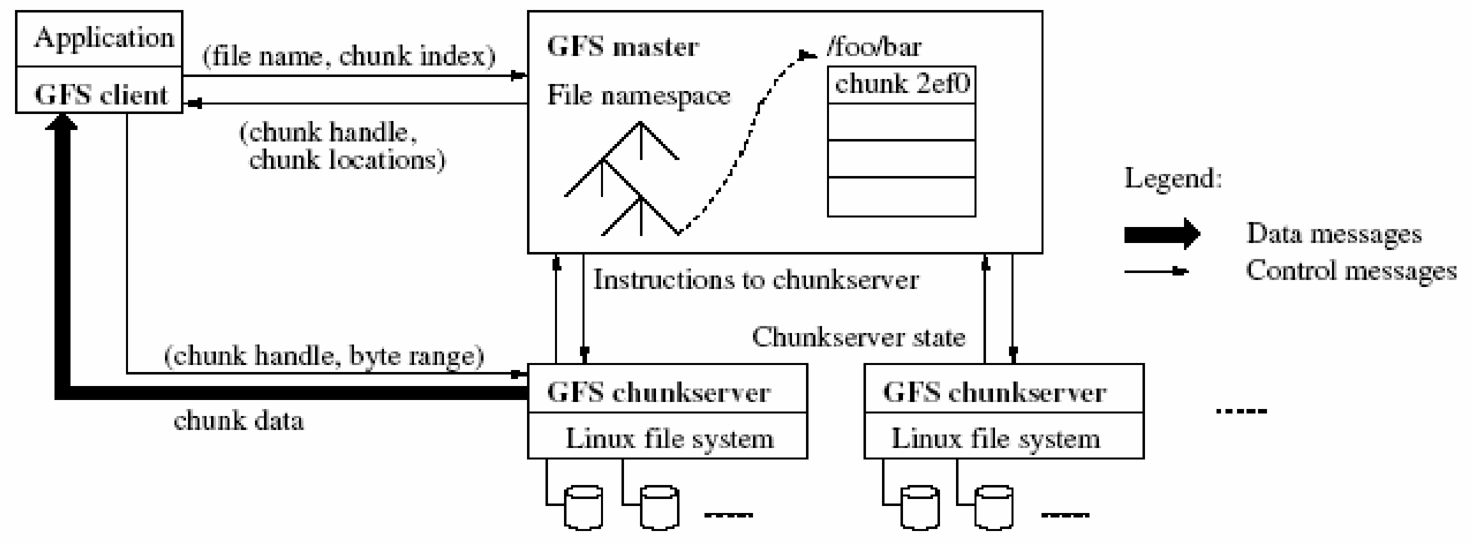


Figure 1: GFS Architecture

Design of GFS

- Consistency Model
  - File regions: “consistent”, “defined”
    - Consistent: All clients will always see same data regardless of which replicas.
    - Defined: Consistent AND clients will see what a mutation writes in its entirety
  - Guarantees by GFS
    - File namespace mutations are atomic (e.g. creation). Handled exclusively by master.
    - Successful mutation without interference → defined
    - Concurrent successful mutations → undefined but consistent
    - Failed mutation → inconsistent (and undefined)

Design of GFS

- Implications of the consistency model on the applications
  - Applications need to distinguish between defined and undefined regions
    - Rely on appends not overwrites => avoid undefined regions
    - Application-level Checkpointing
    - Write self-validating self-identifying records

System Interactions

- Interactions include
  - File Read, File Mutations, and Snapshots
- Has to ensure the consistency guarantees
- Has to ensure the throughput
  - Minimize the involvement of master in the interactions
  - Delegate operations onto chunkservers/clients

System Interactions

- Read access

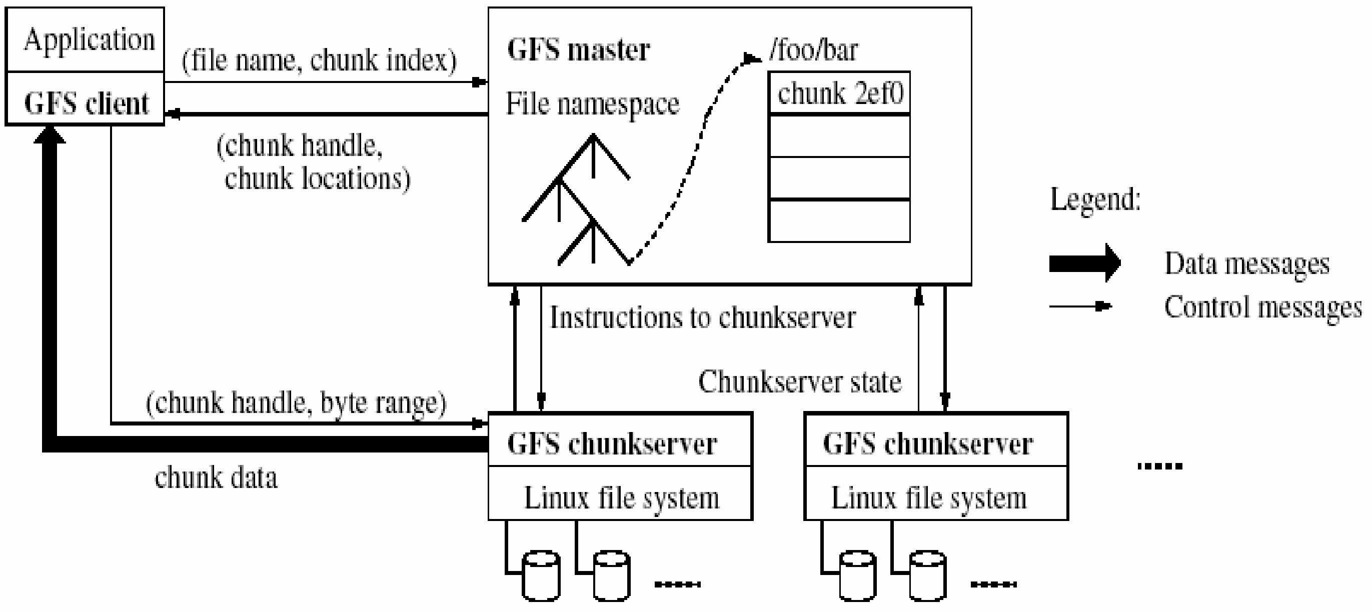


Figure 1: GFS Architecture

System Interactions

- File Mutations: Write/Record Append/Snapshots
  - Write: causes data to be written at an application-specified file offset.
  - Record append: causes data to be appended atomically at least once even in the presence of concurrent mutations.
  - Snapshot: Makes a copy of a file or directory tree
  - Need to ensure consistency model

System Interactions

- Each mutation performed at all the chunk's replicas
- Use leases to maintain a consistent mutation order across replicas.
  - Master grants chunk lease to one of the replicas – the primary.
  - Primary picks a serial order for all mutations to the "chunk". All replicas follow this order when applying mutations.
  - THUS global mutation order defined first by lease grant order chosen by master and then by serial numbers in the lease

System Interactions

- Lease
  - Delegate authorization to the primary replica in data mutations
  - Minimize management overhead at the master
  - Lease expiration time: 60 seconds. Can get extended.

System Interactions

- Write access

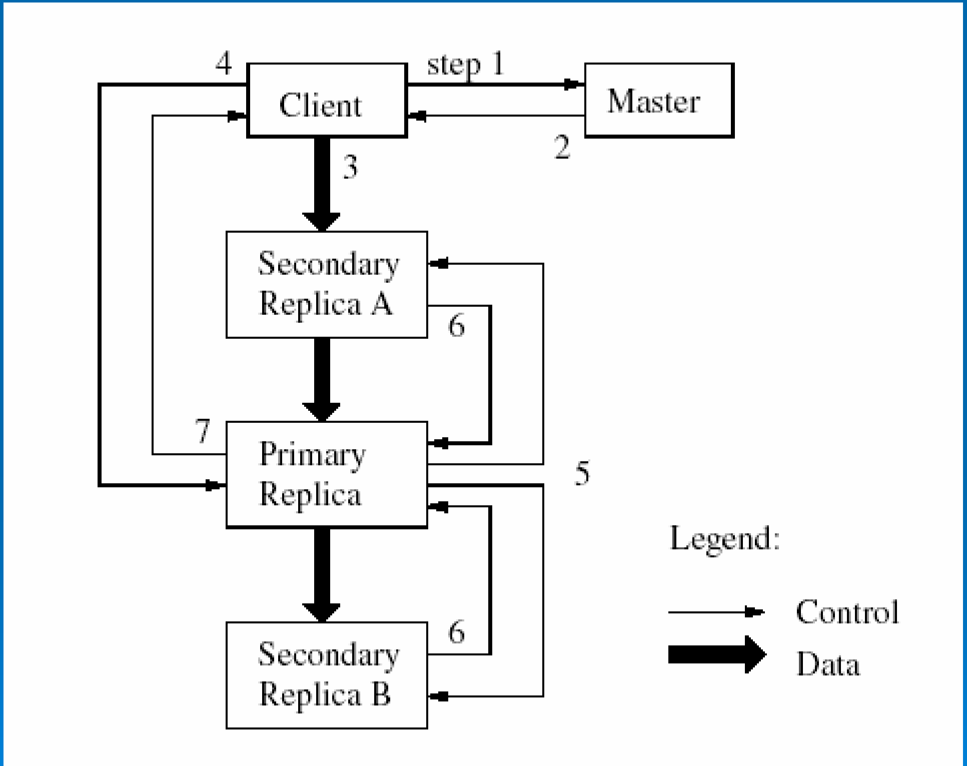


Figure 2: Write Control and Data Flow

1. Client asks master which chunkserver holds current lease and locations of other replicas. If no lease, master grants one to a replica it chooses.
2. Master replies with primary and secondaries. Client caches this for future mutations; contacts master again only when primary unreachable or does not have a lease
3. Client pushes data to all replicas. Client can do so in any order. Each chunkserver stores data in internal LRU buffer cache until data used or aged out.
4. Once all replicas have received data, client sends write to the primary. Primary assigns consecutive serial numbers to mutations it receives, possibly from multiple clients, which provides necessary serialization. Applies mutation to its own local state in serial number order

System Interactions

- Write access

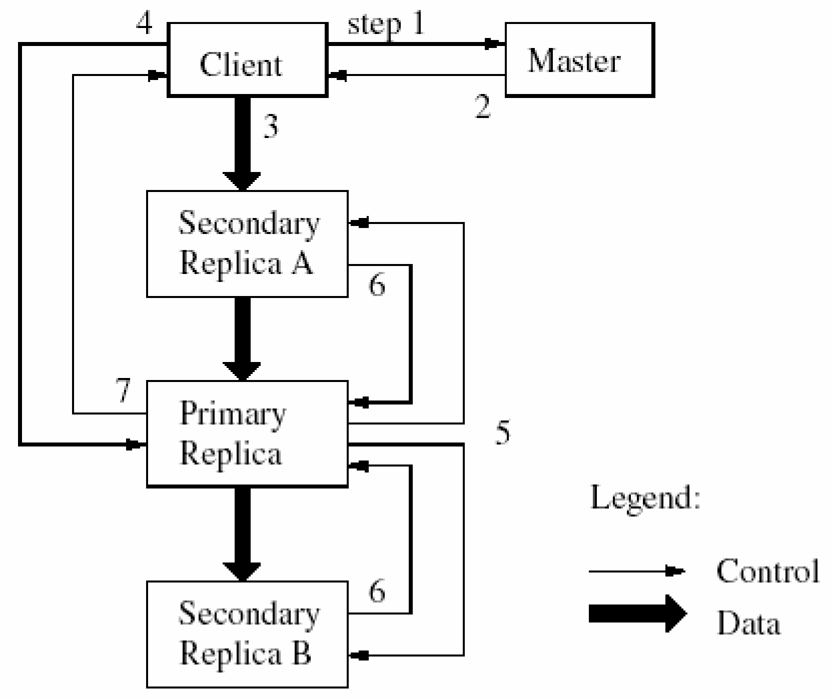


Figure 2: Write Control and Data Flow

5. Primary forwards write request to all secondary replicas. Each secondary replica applies mutations in the same serial number order assigned by the primary
6. Secondaries all reply to primary – “finished the operation”
7. Primary replies to the client
   - Success of operation
   - Operation failed

System Interactions

- Write access
  - If a write by the application is large and straddles a chunk boundary, GFS client code breaks it down into multiple write operations
    - Concurrent writes on the same chunk could result in “undefined” region
    -

System Interactions

- Record Append
  - An atomic operation which appends data to the end of a file
    - The appended data is guaranteed to be “defined” and “consistent”
    - GFS appends it to the file at least once automatically at an offset of GFS’s choosing and returns that offset to the client
    - Control flow similar to a write with some small changes
      - The length of the appended data is limited to 16 MB (1/4 of the maximum chunk size)

System Interactions

- Record Append (steps)
  - Follow similar steps as in the Write access
    - Client pushes data to all replicas
    - Client sends “Append” request to the primary replica
    - Primary checks if the data being appended will straddle the boundary of the last chunk. (Note: the appended data size <= 16MB)
      - Pads the chunk to the maximum size (on all replicas)
      - Replies to the client indicating the operation should be retired on the next chunk
      - This is a uncommon case

System Interactions

- Record append (steps)
  - If the data is within the chunk's boundary
    - The primary appends the data
    - Tells the secondaries to append the data at the same offset
    - Reply success to the client
    - This is the common case !

System Interactions

- Snapshots
  - Makes a copy of a file or directory tree (the “source”) really fast while minimizing any interruptions of ongoing mutations
  - Users use it to quickly create branch copies of huge data sets or to checkpoint current state before experimenting with changes that can later be committed or rolled back easily
  - Adopt the Copy-On-Write technique for high efficiency and low overhead

System Interactions

- When master receives snapshot request, it first revokes any outstanding leases on the chunks in the files it is about to snapshot – thus, any subsequent writes to these chunks will require an interaction with the master to find the lease holder, giving master opportunity to create a new copy of the chunk first.
- After leases revoked or expired, master logs operation to disk. Then applies log record to its in-memory state by duplicating the metadata for source file or directory tree. Newly created snapshot files point to the same chunks as the source files.
  - Reference counts to these chunks are increased by one.
- First time a client wants to write a chunk C after the snapshot operation it sends a request to the master to find the current lease holder.
  - Master notices that the reference count for a chunk C is greater than one.
  - Defers replying to client request and instead picks a new chunk handle C'.
  - Then asks each chunkserver that has C to create a new chunk called C'.
  - Request handling proceeds as normal.

Master operations

- Maintains the mapping between file names and chunks
- Handles the mutation lease
- Namespace management and locking
- Replica placement
  - Chunk creation, re-replication, rebalancing
    - Space utilization, “recent” creations, spread across racks
- Garbage collection
  - Lazy garbage collection for simplicity, background activity
- Stale replica deletion
  - Using chunk version number

Master operations

- Namespace management
  - File creation / deletion
- Locks on the directories / files are used to properly serialize various master operations
  - A snapshot of /home/user requires read-lock on /home and /home/user and write-lock on /home/user^{L1}.
  - A file creation /home/user/foo.dat requires a read-lock on /home/user^{L2} and write-lock on /home/user/foo.dat
  - L1 and L2 conflicts with each other and these two operations will be serialized.

Master operations

- Replica placement
  - Chunk creation
  - re-replication
    - If a replica corrupts or fails.
    - Create more replicas for hot chunks (chunks which are being accessed very often)
  - rebalancing
    - Equalize space utilization
    - Balance the load on each chunk server
    - Spread across racks for higher reliability

Master operations

- Garbage collections
  - Release the occupied space of deleted files
    - File deletion in GFS is just a renaming of the filename to a “hidden” filename
  - Removes orphaned chunks
    - Orphaned chunks are chunks which are not referred to by any files.
    - A result from the failures in the chunk creation/deletion process.
  - Remove stale replicas (mentioned in next slide)
  - Garbage collections are performed when the master is not too busy.

Master operations

- Stale replica deletion
  - Chunk replicas may become stale if the chunkserver fails and missies mutations to the chunk while it is down
  - Everytime the master grants a lease on a chunk it increases the chunk version number on it.
    - The version number will be passed to the replicas and the client.
    - The master keeps a copy of the chunk version number in the metadata.
    - Both the master and the chunkservers will save the version number information in their persistent states before the client is notified about the lease.
  - When version number mismatch happens, the one with the highest version number will be used. The stale replicas (one with lower version number) will be garbage collected.

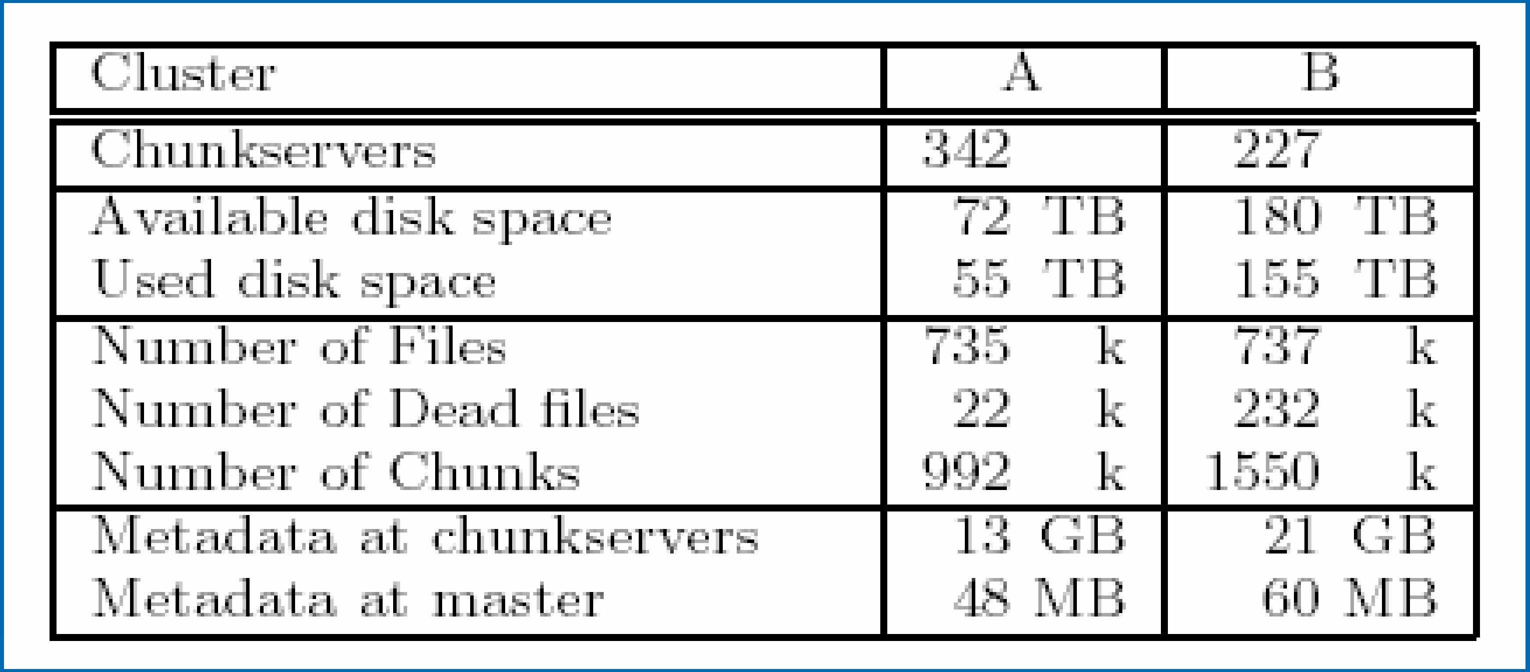
Fault Tolerance

- Fast recovery
  - Master uses operation logs and checkpoints
    - Fast recovery from the nearest checkpoint by reapplying changes recorded in the log
  - When chunkserver boots up (possibly after a crash), it reports the chunks it has to the master. Usually this takes just seconds.
- chunk replication
- master replication
  - Operation logs and checkpoints are replicated on multiple machines
  - A mutation to the state is considered committed only after its log record has been flushed to disk locally and on all master replicas
  - There's only one active master server
    - However, there can be multiple read-only shadow master replicas
  - An external monitoring infrastructure starts a new active master server based on the replicated logs and checkpoints in case of master failure

Fault Tolerance

- Data Integrity
  - Checksums
    - Each 64 KB blocks of a chunk is protected by a 32 bit checksum.
  - Checksum verification is done every time the chunk is being read.
  - During idle periods, chunkserver scans trough the chunks and verify the checksums
- Diagnosis Tools and Analysis of logs

Measurement



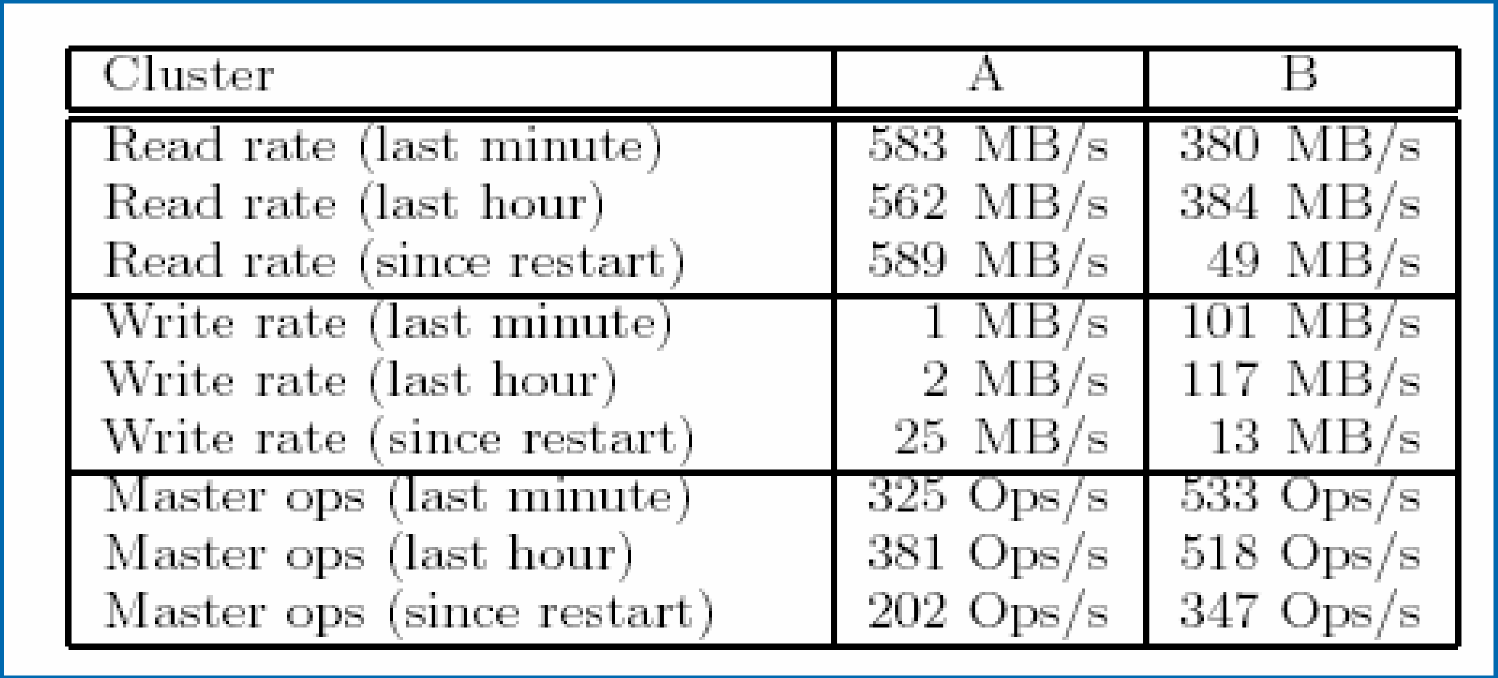
Cluster	A	B
Chunkservers	342	227
Available disk space	72 TB	180 TB
Used disk space	55 TB	155 TB
Number of Files	735 k	737 k
Number of Dead files	22 k	232 k
Number of Chunks	992 k	1550 k
Metadata at chunkservers	13 GB	21 GB
Metadata at master	48 MB	60 MB

Cluster A: used for research and develomenent by over a hundred engineers.

Cluster B: used for production data processing.

Note: average meta size at master or a chunkserver is 50~100MB

Measurement



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

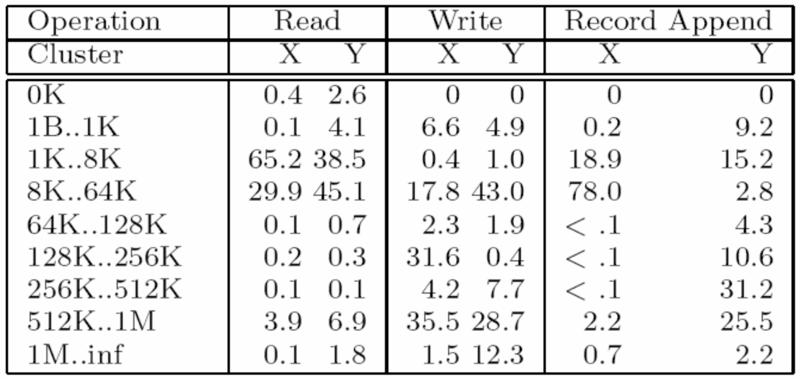
A has a sustaining read rate of 580MB/s for a week while the network configuration can support 750MB/s

Resources are being used pretty efficiently.

Measurement

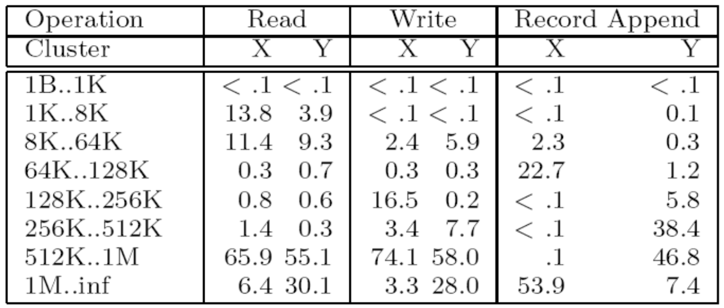
Cluster X: for R&D

Cluster Y: for production



Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
0K	0.4	2.6	0	0	0	0
1B..1K	0.1	4.1	6.6	4.9	0.2	9.2
1K..8K	65.2	38.5	0.4	1.0	18.9	15.2
8K..64K	29.9	45.1	17.8	43.0	78.0	2.8
64K..128K	0.1	0.7	2.3	1.9	< .1	4.3
128K..256K	0.2	0.3	31.6	0.4	< .1	10.6
256K..512K	0.1	0.1	4.2	7.7	< .1	31.2
512K..1M	3.9	6.9	35.5	28.7	2.2	25.5
1M..inf	0.1	1.8	1.5	12.3	0.7	2.2

Table 4: Operations Breakdown by Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested.



Operation	Read		Write		Record Append	
Cluster	X	Y	X	Y	X	Y
1B..1K	< .1	< .1	< .1	< .1	< .1	< .1
1K..8K	13.8	3.9	< .1	< .1	< .1	0.1
8K..64K	11.4	9.3	2.4	5.9	2.3	0.3
64K..128K	0.3	0.7	0.3	0.3	22.7	1.2
128K..256K	0.8	0.6	16.5	0.2	< .1	5.8
256K..512K	1.4	0.3	3.4	7.7	< .1	38.4
512K..1M	65.9	55.1	74.1	58.0	.1	46.8
1M..inf	6.4	30.1	3.3	28.0	53.9	7.4

Table 5: Bytes Transferred Breakdown by Operation Size (%). For reads, the size is the amount of data actually read and transferred, rather than the amount requested. The two may differ if the read attempts to read beyond end of file, which by design is not uncommon in our workloads.

Reviews and Conclusions

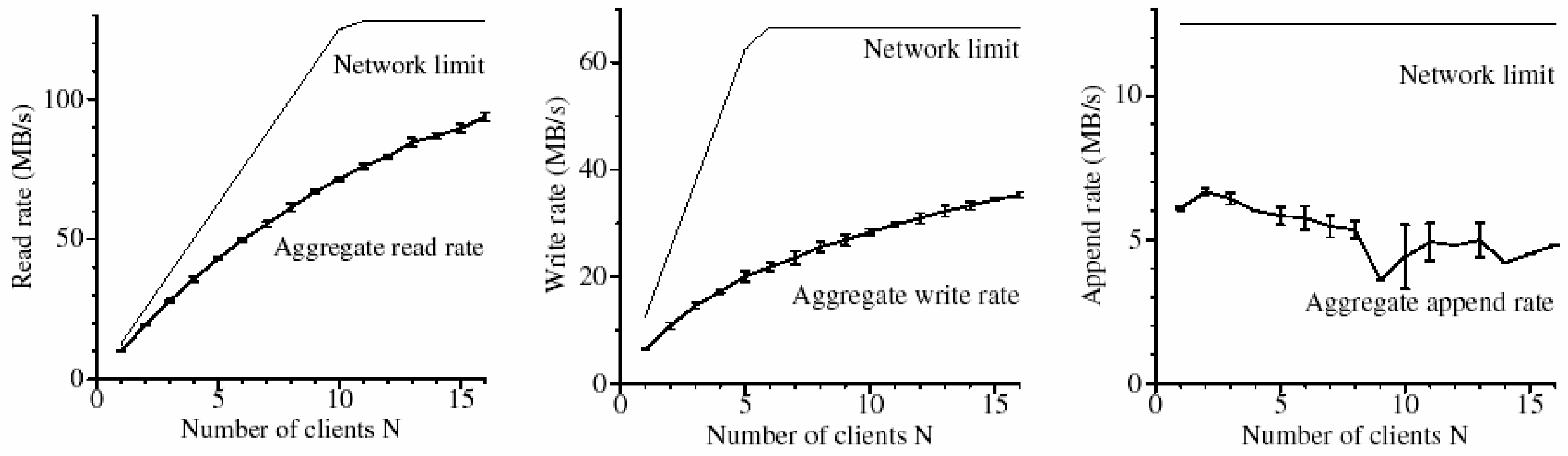
- Specials about GFS
  - No caching
    - Due to Google’s workload pattern
  - Centralized (Single) master
    - Good for Google’s file pattern (a bunch of giga or tera byte sized files)
    - Bad for zillions of (small) files
    - Simpler and more efficient in file meta data management
  - Relaxed consistency model
    - Simpler design and implementation
      - Potentially more reliable
    - Supports higher throughputs

Reviews and Conclusions

- Specials about GFS
  - Incorporate multiple fault-tolerance measures
    - Each chunk is protected through checksum and replications
    - Master server is protected through logging/checkpointing/recovery and replicaions
    - Dynamic replica reallocation
    - Replication costs ($$$) is a non-issue for Google
  - Does not use majority voting as a fault tolerance mechanism
    - Need to maintain high-throughputs
    - Corruption in the meta data at the server can be a problem
      - Detection through diagnosis tools and analysis of logs

Reviews and Conclusions

Measurement



(a) Reads (b) Writes (c) Record appends

Micro-benchmarks on a GFS cluster consisting of one master, two master replicas, 16 chunkservers, and 16 clients.