Google File System (GFS)

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

Authors: Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung

Why Build GFS?

- Need a scalable, distributed file system that targets Google's workloads
  - Need to support much bigger (GB+) files
  - On 100/1000s of commodity servers that fail regularly
- Workloads process bulk multi-GB/TB datasets
  - High throughput more important than low latency accesses
  - Mostly sequential reads
    - Read sizes are bimodal, between 1-64K, or larger than 512KB
  - Most writes are file appends
    - Multiple clients perform concurrent file appends, e.g., producer-consumer queues, many-way merge operations, etc.
    - Overwrites are practically non-existent

GFS Interface

- Google co-designs its applications
  - Applications don’t require POSIX compliance
  - Weaker consistency for higher throughput is acceptable
- Supports typical file system operations
  - E.g., create, delete, open, close, read, and write
- record append: allows multiple concurrent clients to append data to the same file
  - At-least once semantics
- snapshot: create copy of file/directory tree at low cost
  - Enables backup, experimentation
  - Similar to some modern file systems, e.g., btrfs, zfs

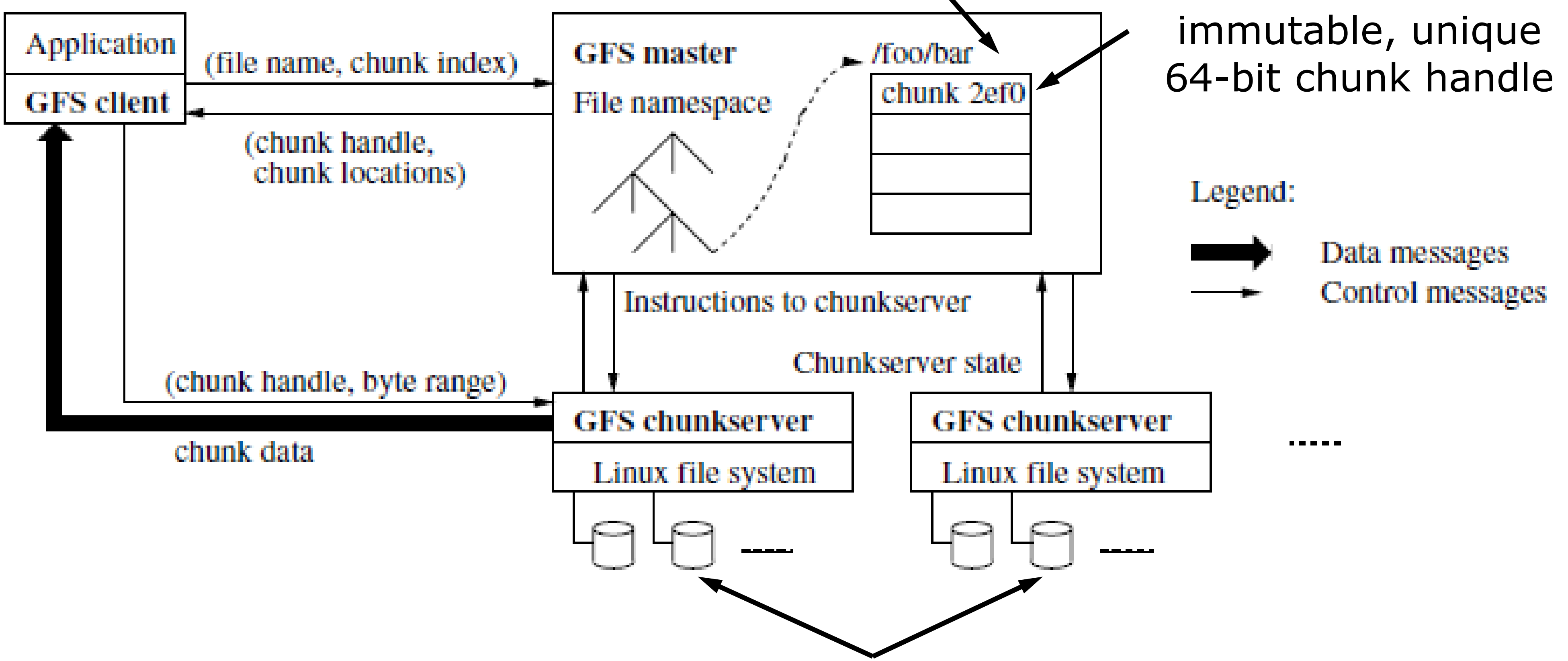
Key Design Ideas

- Use a cluster of inexpensive, commodity machines
  - Separate metadata and data operations for scalability
  - Node failures are common, so need fault tolerance
- Single metadata server
  - Simplifies design of overall system
  - Serializes metadata operations using a metadata log
  - Replicates metadata log for fault tolerance
  - Manages data replication
    - Data consistency, replica placement, load balancing, etc.
  - Avoids performing any data operations
- Data servers …

Key Design Ideas

- Use a cluster of inexpensive, commodity machines
  - Separate metadata and data operations for scalability
  - Node failures are common, so need fault tolerance
- Single metadata server …
- Data servers
  - Store replicas of chunks (fixed-size partitions) of files
  - Chunk size is relatively large (64MB)
    - Allows efficiently accessing large files
  - Support file appends efficiently

GFS Architecture



Master

- Maintains file system metadata in memory:
  - Chunk namespace, i.e., all chunk handles in the system
    - For each chunk: reference count (for copy-on-write snapshots), version number (for detecting stale chunk replicas)
  - File namespace, i.e., all file paths
    - For each file path: acl, file->chunk_handle mappings
- This metadata is stored persistently for failure recovery
  - All metadata changes are ordered and logged to disk
  - Log is replicated to backup master nodes
  - Then changes are applied to in-memory structures
  - In-memory structures are periodically checkpointed to reduce recovery time

Master

- Caches chunk locations, i.e., chunkservers on which a chunk is stored
- Maintains per-filepath read-write locks for ensuring that metadata operations are performed atomically
- This data is not stored persistently

Chunkserver

- A chunkserver stores
  - Chunks as Linux files on local disks
  - Chunk handles is Linux filename
  - Checksums for each 64KB block within chunk
- Each chunk is replicated across three chunkservers
- Application may read chunk from any replica
- Chunkservers report chunks they store to master
  - Master controls chunk placement but chunkservers serve as authorities for chunks

Master <-> Chunkserver

- Master periodically communicates with each chunkserver using HeartBeat messages
- Enables master to:
  - Know about chunk locations
  - Perform lease management, i.e., maintain primary for a chunk
  - Determine stale chunk servers
  - Garbage collect orphaned and stale chunks, etc.

Weak Consistency Model

- Definitions:
  - consistent: for a file region, all replicas store the same data
  - defined: after a write to a file region, the region is consistent and has the entire write (same as linearizable write)
- Complicated guarantees
  - Serial write: defined regions
  - Failed write: inconsistent regions
  - Concurrent writes within a chunk: defined regions
  - Concurrent writes that cross chunks: consistent but not defined regions
  - Record append: defined region, possibly interspersed with inconsistent regions

Understanding Consistency Model

- Serial write
- key idea: write to each chunk is applied to all replicas, so writes are linearizable, i.e., defined regions

write(chunk A, A1,
chunk B, B1)

R1	R2	R3	R4
A1	A1	A1	
	B1	B1	B1

Understanding Consistency Model

- Failed write
- All replicas must respond with success or else a write is considered failed
- In this case, the region may have different data at the different replicas, i.e., inconsistent region

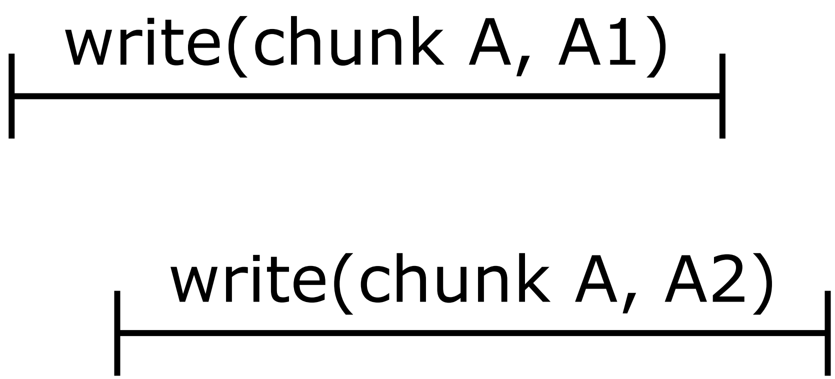
write(chunk A, A1)

R1	R2	R3
A1	A1	

- Application needs to handle failure by retrying write

Understanding Consistency Model

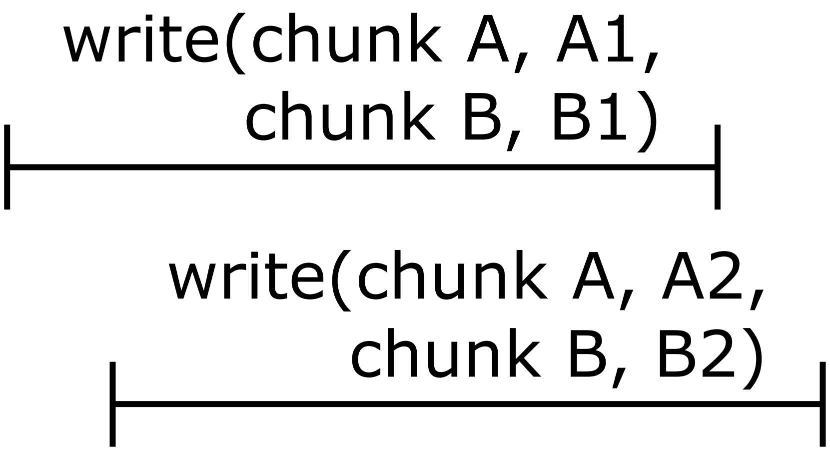
- Concurrent writes within a chunk
- key idea: writes to each chunk are applied in the same order to all replicas, so writes are linearizable, i.e., defined regions



R1	R2	R3
A1	A1	A1
A2	A2	A2

Understanding Consistency Model

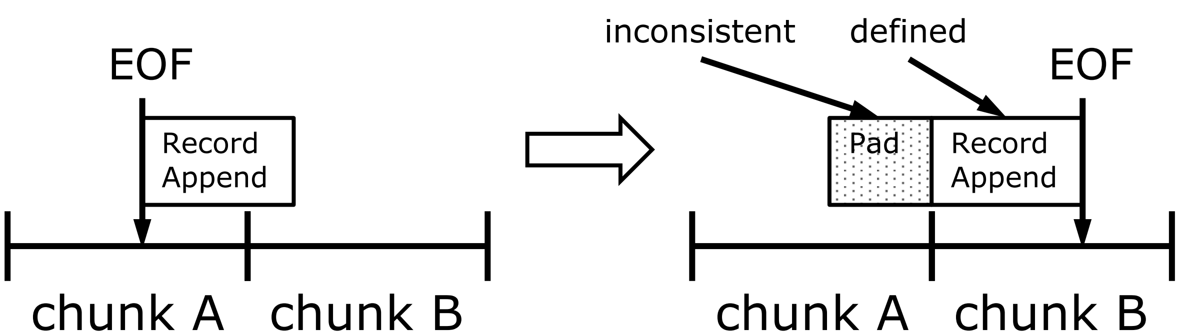
- Concurrent writes that cross chunks
- Problem: writes to different chunks may be serialized in different order, so writes are not linearizable
  - e.g., final state is consistent (A=A2, B=B1), but not defined



R1	R2	R3	R4
A1	A1	A1	
A2	B2	A2	B2
	A2	B2	B1
	B1	B1	

Understanding Consistency Model

- Record append
- Need to ensure that record append yields a defined region, i.e., these operations are linearizable
- Key idea: force append to lie within chunk with padding

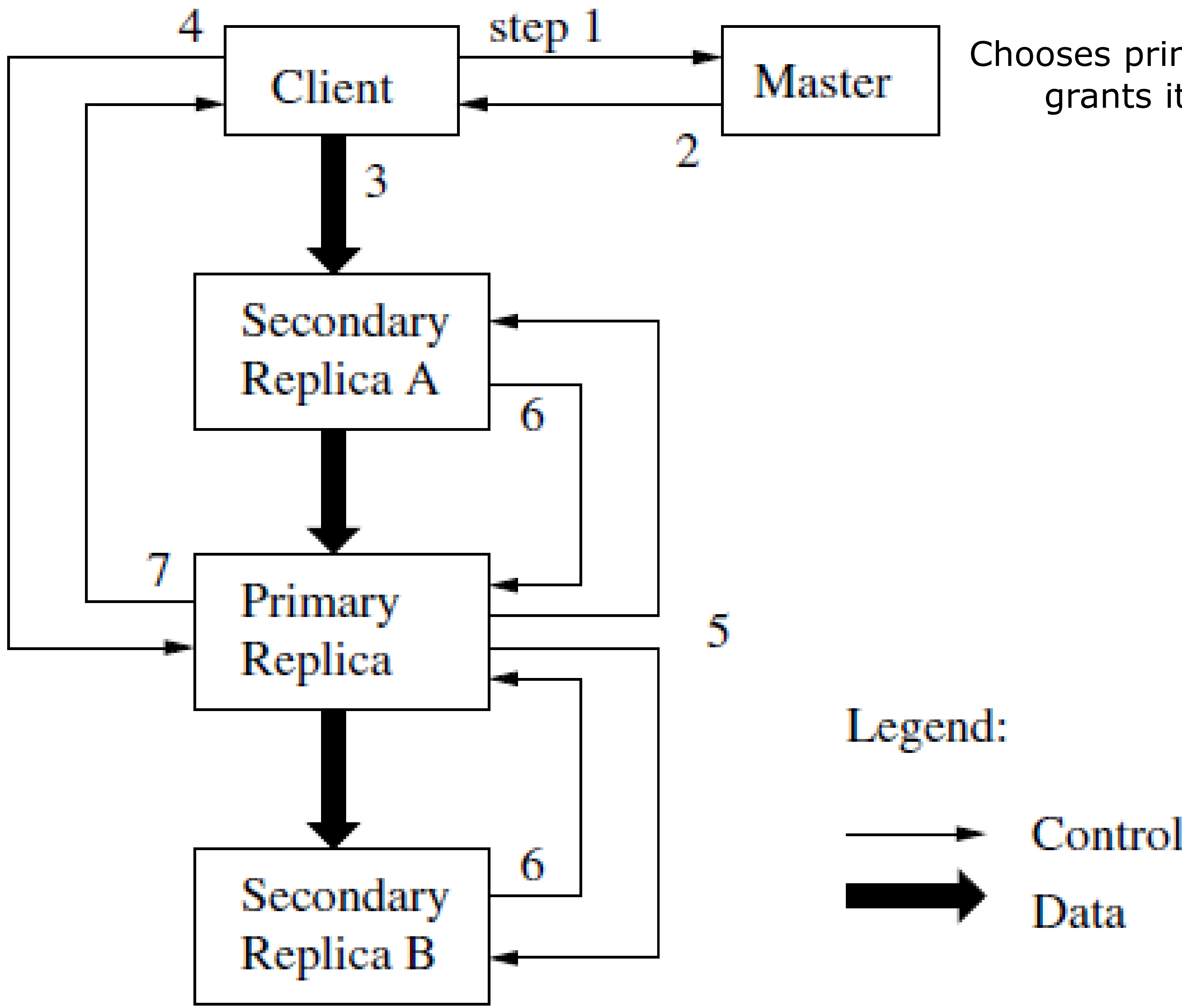


- Padding is an inconsistent region
- If record append fails, it creates an inconsistent region
  - A retry may lead to duplicate record appends

Implementing Consistency Model

- Use lease mechanism to ensure consistency
  - Master grants a chunk lease to one of replicas (primary)
  - Primary picks a serial order for all mutations to the chunk
  - All replicas follow this order when applying mutations
  - Global mutation order defined by
    - Lease grant order chosen by the master
    - Serial numbers assigned by the primary within lease
- If master doesn't hear from primary, it grant lease to another replica after lease expires

Chunk Write Implementation



Chooses primary replica, grants it a lease

Stale Replicas

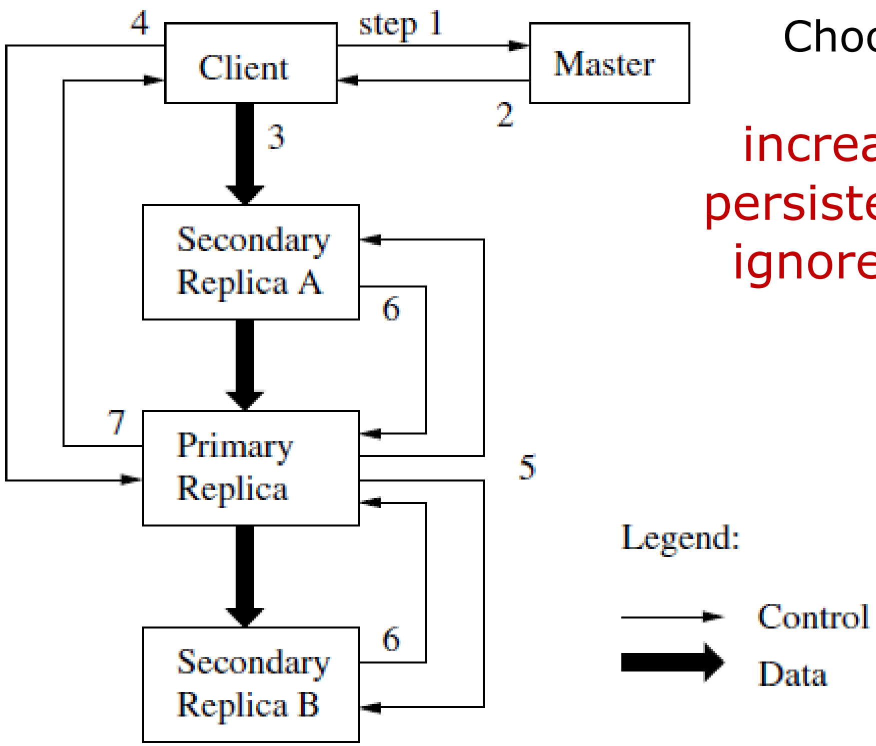
- What would happen if a replica is stale, i.e., doesn’t have recent writes, and a write is attempted?

write(chunk A, A2)

R1	R2	R3
A1	A1	

- Assume A1 and A2 do not overlap within Chunk A
  - Writing A2 to the three replicas will make them inconsistent
  - Reading from R3 will not return the A1 update

Detecting Stale Replicas with Chunk Versions



Chooses primary replica, grants it a lease,

increases chunk version number, stores it persistently, tells all replicas to do the same, ignores replicas with stale chunk versions.

write(chunk A, A2)

R1	R2	R3
A1	A1	
A2	A2	

Master Failures

- Master replicates metadata operation log and checkpoints to backup masters
- An external service detects master failure and promotes a backup to primary
  - Updates DNS so clients can access new master
- Backup applies operation log to its most recent checkpoint before starting operation

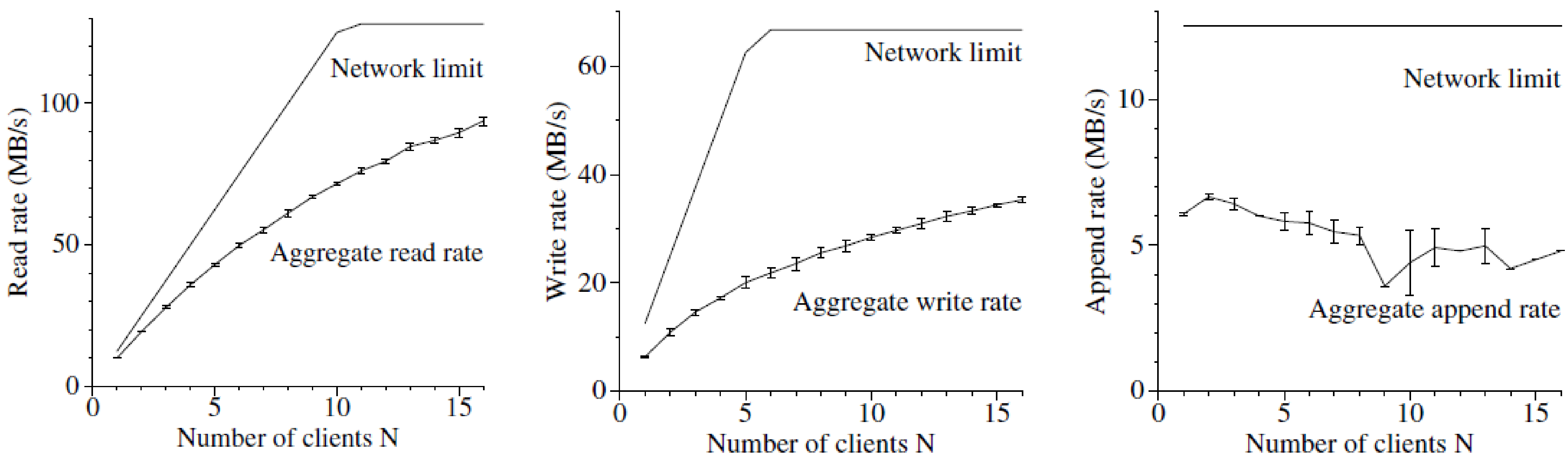
Chunkserver Failures

- Master uses HeartBeat messages to determine chunkserver status, enables master to:
  - Learn about failed chunkserver
  - Switch primary for chunks stored on chunkserver
  - Clone chunks that have fewer than 3 replicas to other chunkservers

Evaluation

- Performance measured on test cluster with:
  - 1 master, 2 master backups
  - 16 chunkservers (store 3 replicas for each chunk)
  - 16 client machines
- Server machines connected to 100 Mbps switch
- Client machines connected to second 100 Mbps switch
- Switches connected with 1 Gbps link

Performance



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

Conclusions

- Single master can dramatically simplify design
  - However, need to carefully design it to ensure it doesn't become a CPU, memory, disk, etc., bottleneck
- Decoupling metadata and data operations in file systems enables optimizing for them separately
- Targeting important use cases (e.g., concurrent appends) allows focusing on correct abstractions
  - Enables scaling with weaker consistency guarantees
- Very influential
  - Apache HDFS based on GFS design

GFS: Pros, Cons

- Pros
  - Can handle massive data and massive objects scalably
  - Works well for large sequential reads, appends
  - Simple, robust reliability model
- Cons
  - Metadata server can be bottleneck, single point of failure
    - However, sharding the namespace or replicating the server is feasible
  - Weak consistency guarantees
    - Linearizability for single chunk writes (not for cross-chunk writes)
    - Stale chunk reads possible
    - Duplicate and inconsistent data can be read
  - Small reads, overwrites are expensive

Q1

- Traditional file systems use small block sizes, e.g., 4KB, while GFS uses a large chunk size (64MB). Why does it use such a large chunk size? What are the tradeoffs?

Q2

- What are the most important differences between GFS and Zookeeper in terms of functionality?

Q3

- Zookeeper and RAFT ensures linearizable writes using a quorum-based protocol. How does GFS ensure linearizable data writes and metadata operations without using a similar protocol?

Q4

- When a chunk write fails at any chunkserver, GFS 1) exposes the failure to the client, and 1) may update the chunk at some chunkservers (inconsistent region). Why is this done? How does this approach compare with writes to Zookeeper?

Q5

- Why are stale reads possible in GFS? Why does GFS allow them?