- Yogesh Simmhan
- simmhan@iisc.ac.in

- Department of Computational and Data Sciences
- Indian Institute of Science, Bangalore

DS256 (3:1)

Scalable Systems for Data Science

Module 1

Introduction to Big Data & Distributed Storage

Data Mutations

- Three types of **data mutations**: *write, append* and *record append*
- **Write**: Data is written at a *client-specified* file offset.
  - write(fileId, offset, bytes[])
- **Append**: Data is written at a client-specified file offset, where offset is *client's perception of the EOF*.
  - s = getFileSize(fileId)
  - write(fileId, s, bytes[])
- **Record Append**: Causes data (the "record") to be appended *atomically at least once* even in the presence of *concurrent mutations*, but at an *offset of GFS's choosing*.

Consistency Model

- Consistency is defined for “regions” within a block
  - Byte-ranges in the block that are written to by a client
- Regions may be:
  - **Consistent**: All replicas have the same byte contents for that region
  - **Defined**: It is consistent, and the regions reflects the *complete update* performed by a single write client
  - **Inconsistent**: The byte region is different for the different replicas
- Block replicas can have a mix of these three region types
  - *They are still a replica even if all blocks are not byte-wise identical in their entirety*

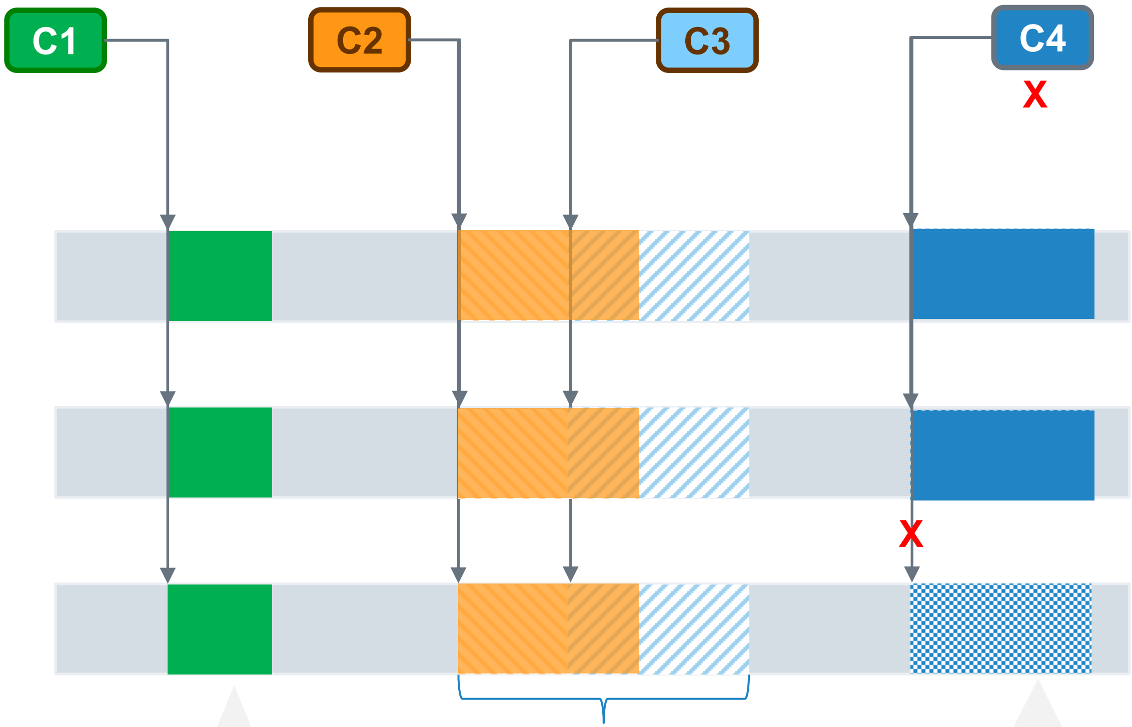
Consistency Model

- If only one client is mutating a block, and it succeeds, the affected region is **defined (and consistent)**
  - All replicas have the **same and complete** content for that region
- If multiple concurrent clients mutate a block, and all succeed, the affected region is **consistent (but may be undefined)**
  - But all mutations from any one client may not be present in the region…*"mingled fragments from multiple mutations"*…same but incomplete content
- Failed mutations → **Inconsistent** region
  - Different clients may see different data at different times

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

Table 1: File Region State After Mutation

Consistency Model



Defined

Consistent, but undefined

Inconsistent

Dataflow in different orders from different clients. Buffered in memory.

Control flow (commit to disk) is done in serial order. FCFS.

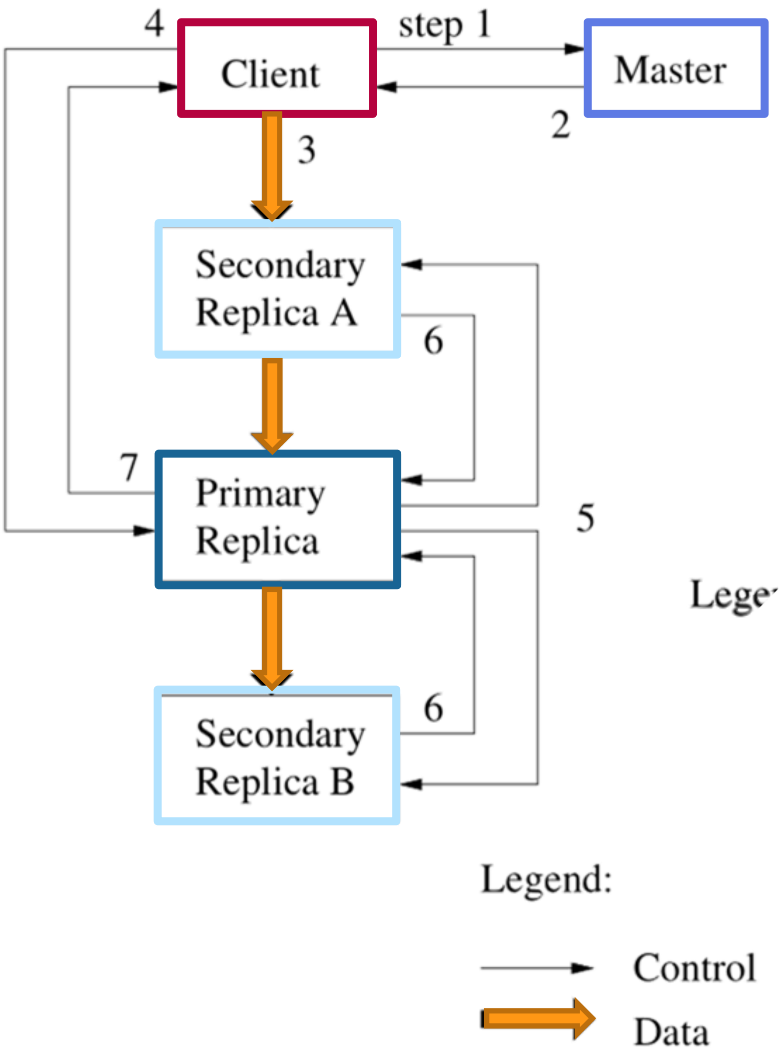
No Rollback.

Leases and Mutation

- Mutation performed at **all** the chunk's replicas
- *Leases* to maintain a consistent mutation order across replicas
- Master grants chunk lease to one of the replicas: **primary**
  - **Primary** picks a serial order for all mutations to a chunk
- Global mutation order
  - *Lease grant order* for picking a primary
  - Within a lease, the *serial numbers* assigned by primary

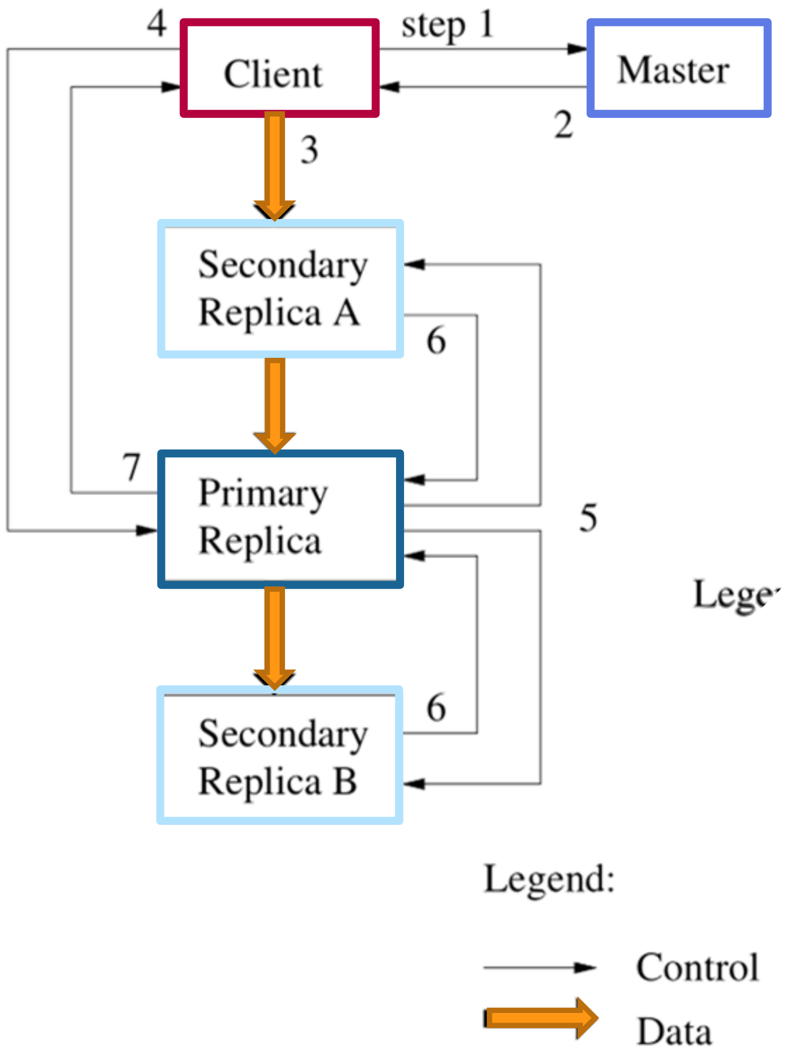
Write Data Flow

- Client gets *primary* and *secondary chunk servers* for a block from Master
  - Caches this info
- Client sends block data to all replicas, in any order
  - Worker stores in LRU buffer
- When all workers *ack*, client sends write request to *primary*
- Primary assigns serial # to mutation
  - Applies mutation locally
  - Forwards write request to all secondaries
- Secondary writes mutation in serial # order
  - *Acks* to primary after writing mutation
- Primary *acks* to client after acks from all secondaries



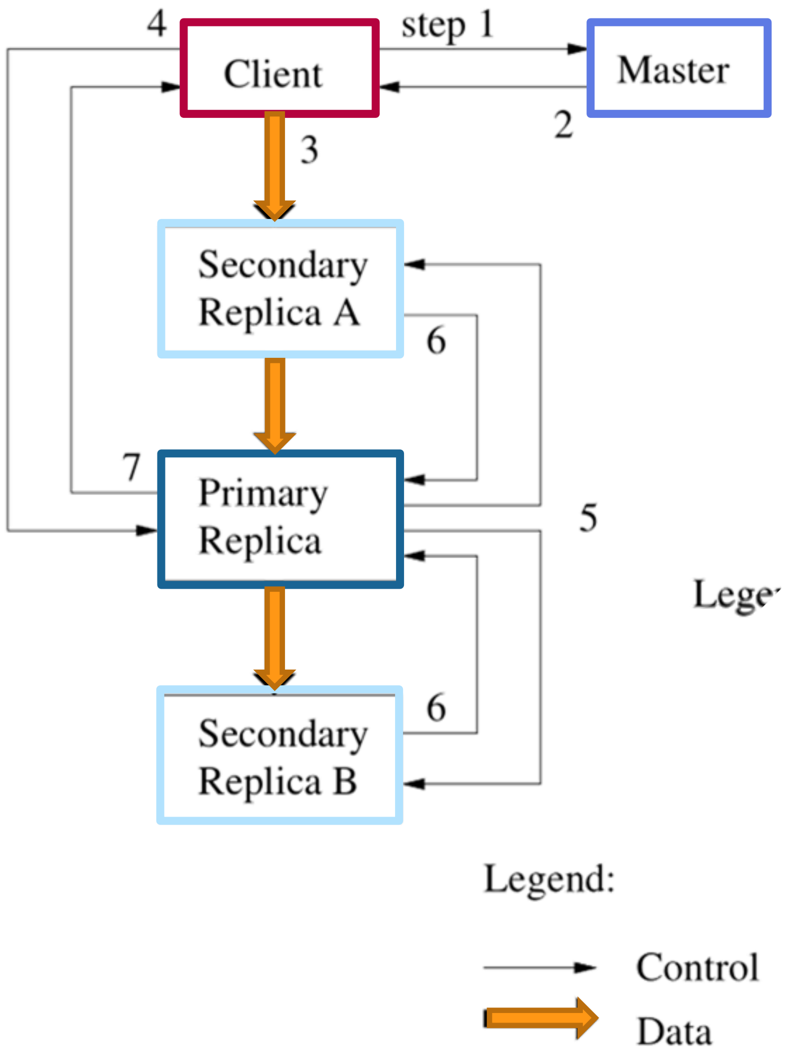
Write Data Flow

- ▷ On concurrent client writes
  - Mutations from different clients can be interleaved
  - But single serial order for writes to a chunk → chunk remains consistent
- ▷ On failure
  - If primary fails, no writes were done to chunk. Client retries.
- ▷ Primary has written, one or more secondary fails → Write has failed. Region is left inconsistent. On failure
  - If primary fails, no writes were done to chunk. Client retries.
  - Primary has written, one or more secondary fails → Write has failed. Region is left inconsistent.



Write Data Flow

- Decoupling data flow from control flow
- Control flow from Client to primary to secondary
  - Ensures serial order of writes, consistency
- Data flow can be intelligent
  - **Pipelined**, allowing input *and output bandwidths* to be used fully
  - Take path of **maximum bandwidth**, with knowledge of N/W topology/distances



Atomic Record Append

- Similar to write dataflow/control flow
- Client sends write request to primary
- Primary checks for chunk overflow
  - If so, **pads** the chunk till end of capacity, informs secondaries to do the same, asks client to retry
- Else, appends record to primary's replica
  - Asks secondaries to write record at the **same offset** as primary
  - Acks to client with offset location
- If record append **fails**, client retries
  - Replicas can contain duplicate records. But there will be *at least one copy* of the record on all replicas at a specific offset, if successful, i.e., a defined and consistent region

Snapshot

- Makes copy of a file or a directory tree *instantaneously*, minimizing interruptions to ongoing mutations
- Master revokes pending leases on chunks of the file
  - Else, waits for leases to expire
- Duplicates metadata of source file to snapshot file
- **Copy on write technique**
  - Retains the same chunks for snapshot file as source file
  - Increments *reference counter* for chunk
- Client writes to chunk
  - Master checks if reference counter > 1
  - Copies old chunk to new chunk ID, and replicates content
  - Write is performed on new chunk ID

Implications for Applications

- Relying on appends rather than overwrites
- Checkpointing
- Writing self-validating, self-identifying records
- Example #1
  - A **writer** generates a file from beginning to end
    - It atomically renames the file to a permanent name after writing all the data *(or)*
    - It periodically checkpoints how much has been successfully written, including application-level checksums
  - **Readers** verify and process only the file region up to the last checkpoint, which is in the defined state
  - Checkpointing lets writers to restart incrementally and prevents readers from processing incomplete writes

Implications for Applications

- Example #2
- Multiple append Writers and record Readers as a producer-consumer queue
  - Semantics of record append preserves each writer’s output
- Readers deal with the occasional padding and duplicates
  - Writers write extra information like checksums per record so that it can be verified
  - A reader identifies and discards extra padding and record-fragments using the checksums
  - An application specific UUID per record can help identify the occasional duplicate records

Master: Namespace & Locking

- File identified using full path name
  - Directory hierarchy is just a logical structure
- Allow multiple concurrent operations using *coarse read/write locks on namespace regions*
  - E.g., snapshot can take time
  - Locking of all directory prefix paths, e.g., get locks on /d1, d1/d2, /d1/d2/f1 to lock f1…in that order
- Allows different operations to proceed concurrently and consistently
  - File or directory level read or write locks
  - E.g., multiple files created concurrently in the same dir
    - Each acquires a read lock on the dir and a write lock on the file
    - Dir read lock prevents it from being deleted/renamed/snapshotted
    - File write lock serialize attempts to create the same file name

Master: Replica Placement

- 100s of chunk server, 100s of clients
- Same or different racks, one or more NW switches
  - Different aggregate B/W at server, rack, switches
- Chunk placement strategy
  - Maximize reliability/availability
  - Maximize bandwidth usage
- Spread replicas **across machines** and **across racks**
  - Survives even if a rack goes offline, e.g. switch fails
  - Use aggregate BW across multiple racks
  - But...write traffic across racks costlier

High Availability

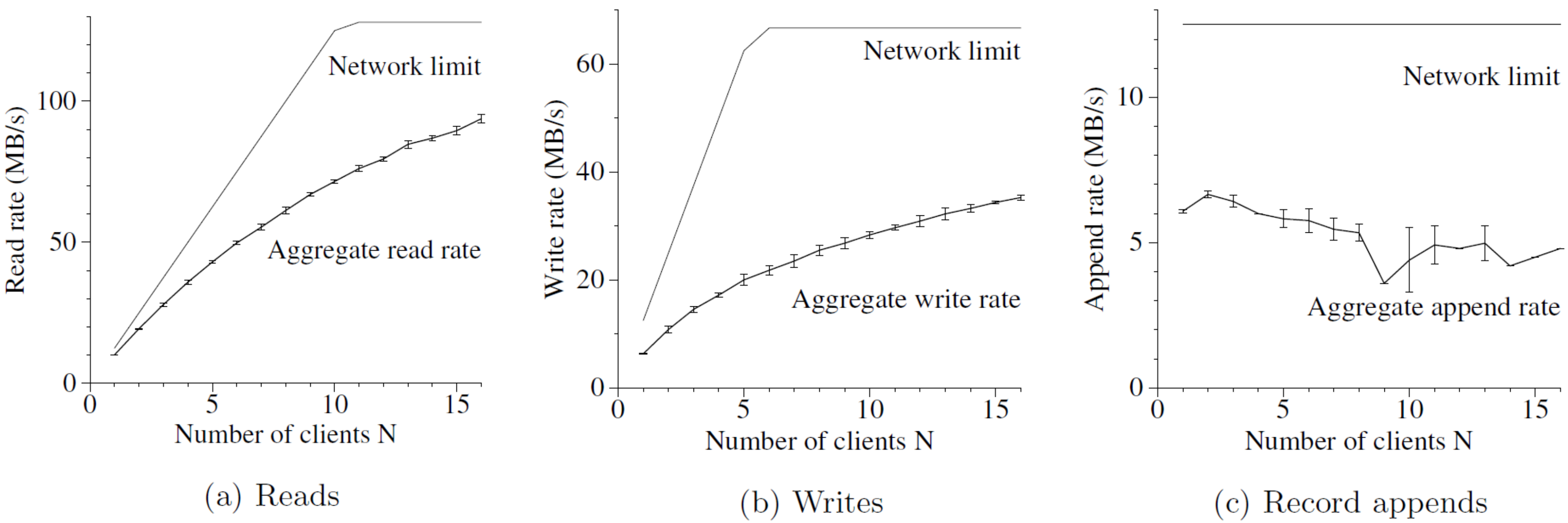
- Fast recovery: Master and Workers can restore state and restart in seconds
- Chunk replication
  - Scanning and cloning to ensure reliability level
  - Checksums to protect chunk corruption
- Master replication
  - Ops log and checkpoints replicated on other machines
- **Shadow** read-only **master** for (stale) namespace access
  - Built using ops log, Worker updates on chunk replica list

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



More Recent Results

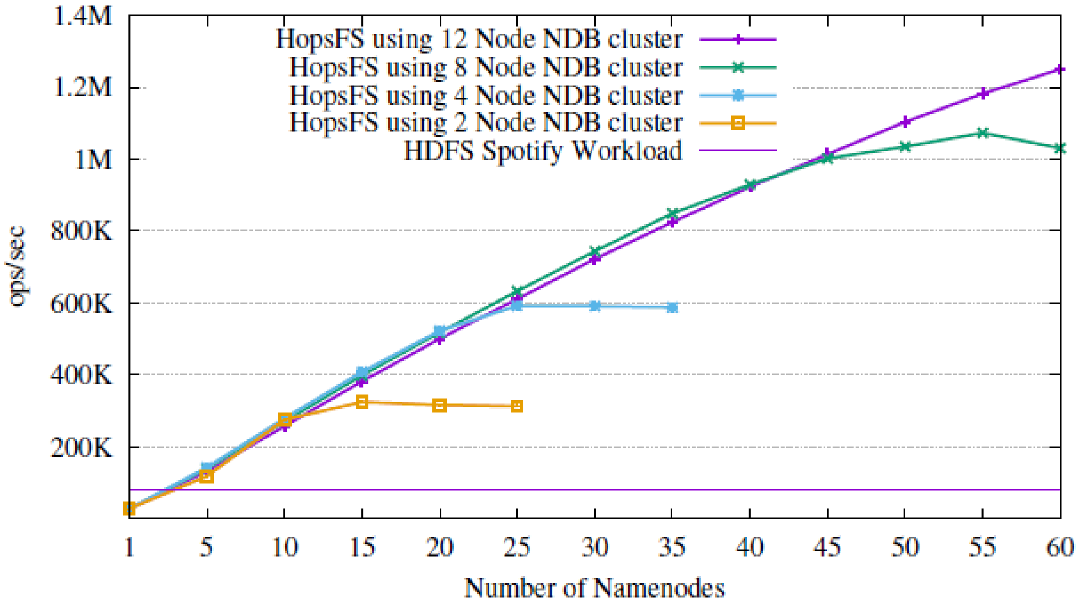


Figure 4: HopsFS and HDFS throughput for Spotify workload.

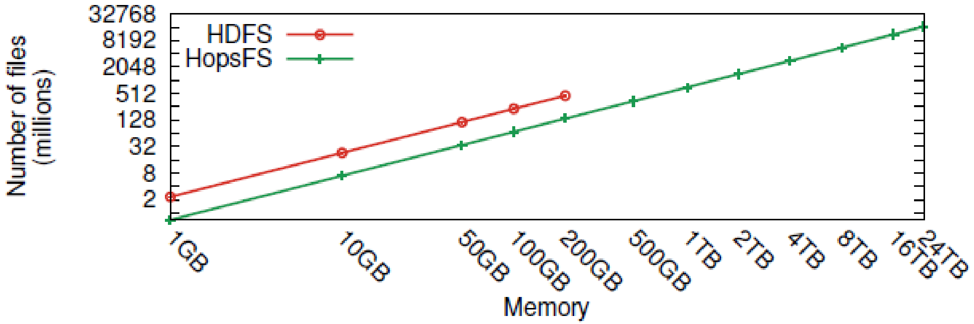


Figure 5: HopsFS and HDFS Metadata Scalability plotted in Log Scale.

Ismail, Mahmoud, et al. "Scaling HDFS to more than 1 million operations per second with HopsFS." *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017.

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!

Hadoop Distributed File System (HDFS)

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 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.

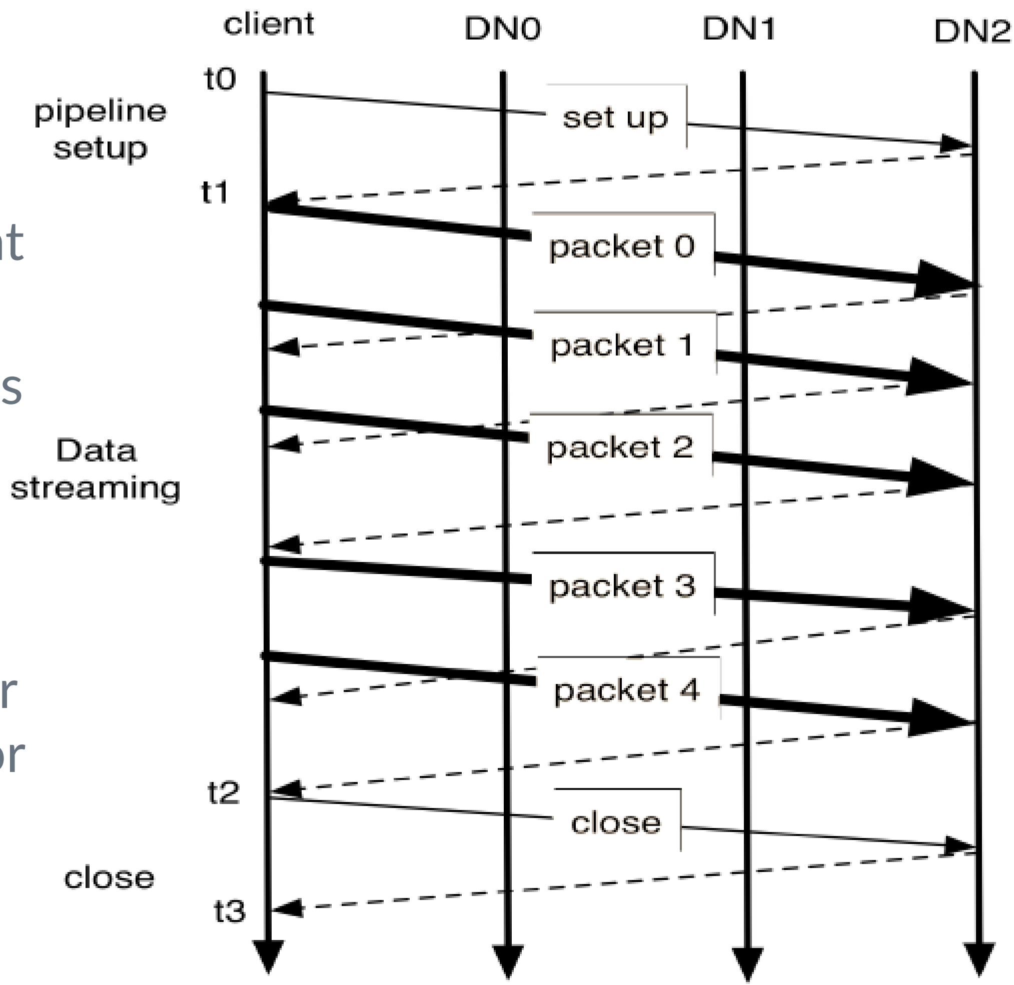
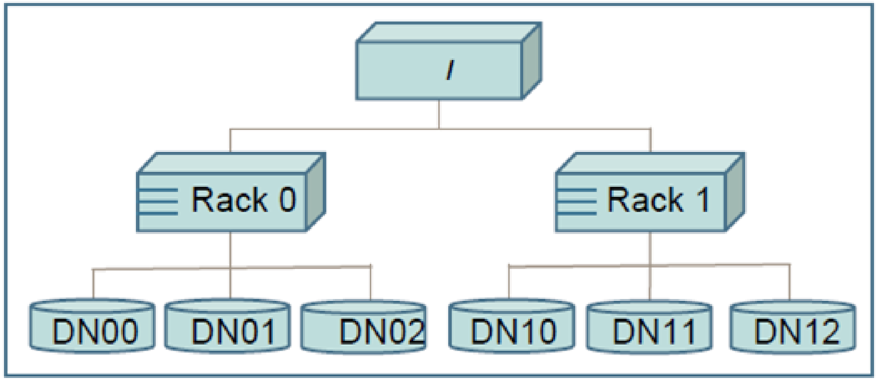


Figure 2. Data pipeline during block construction

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: **minimizing** write cost, vs. **maximizing** reliability, availability, agg read B/W
  - 1^{st} replica on writer node, the 2^{nd} & 3^{rd} 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%) should differ 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

Is failure really a concern?

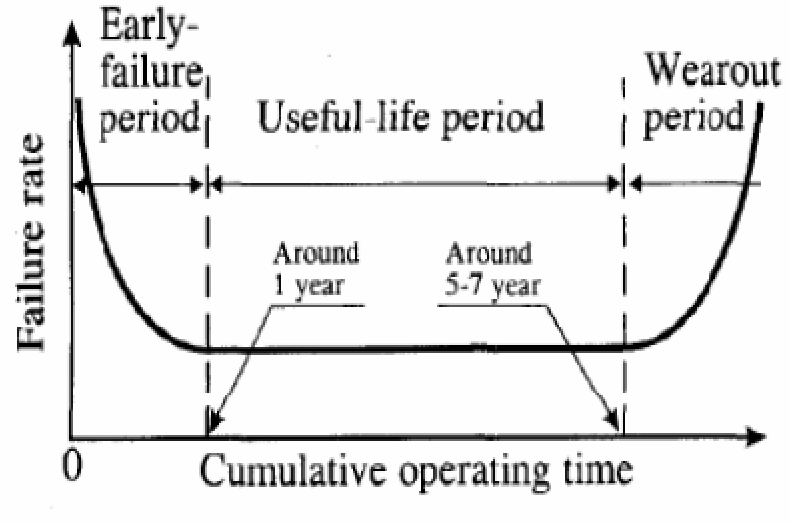
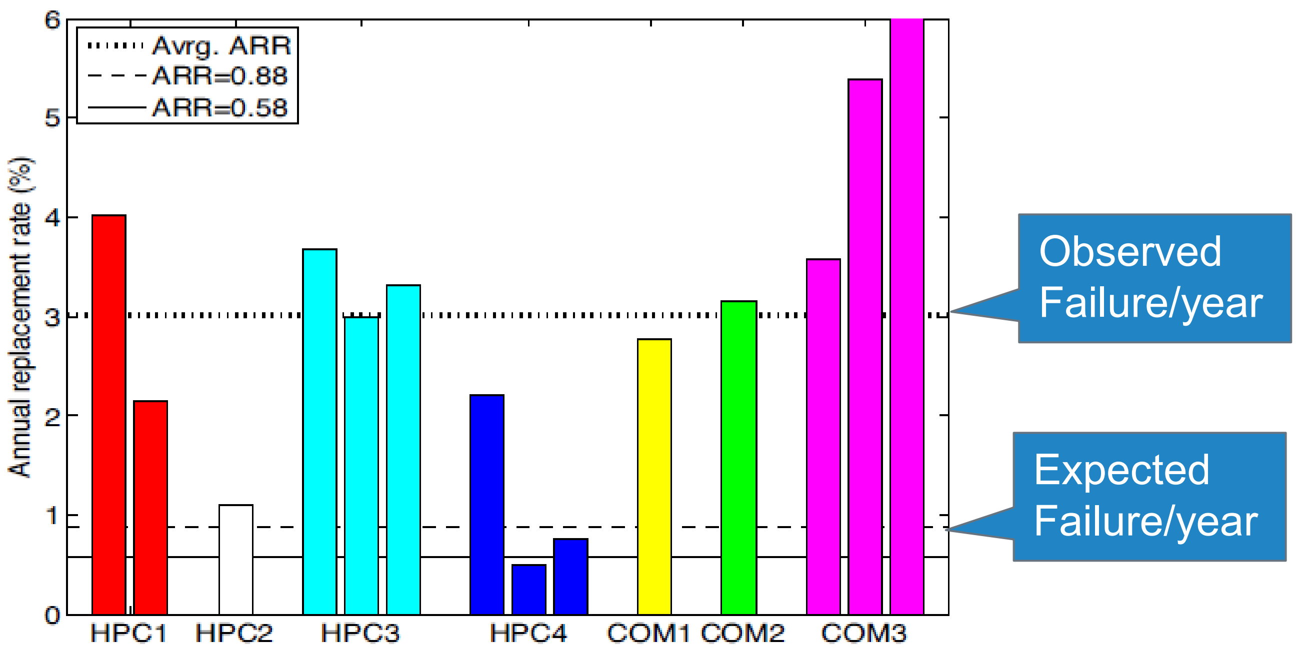
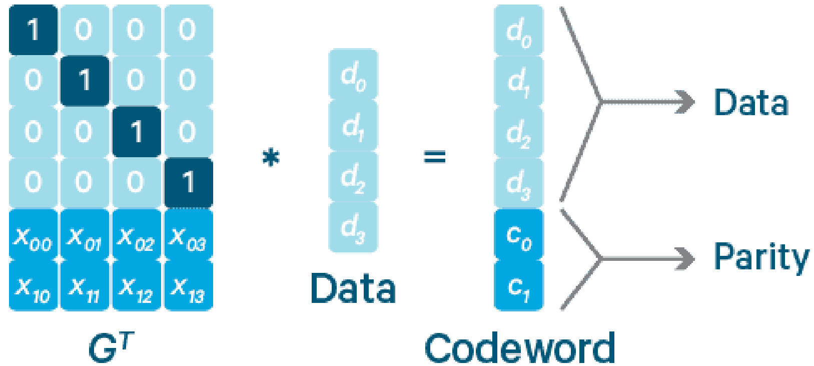


Figure 2: *Lifecycle failure pattern for hard drives [33].*



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

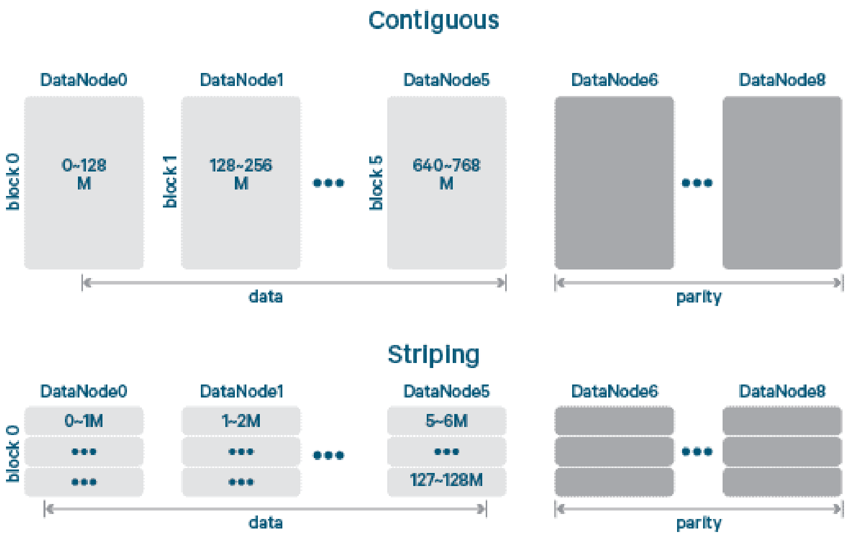
Reed-Solomon encoding with four data units and two parity units



Erasure Coding

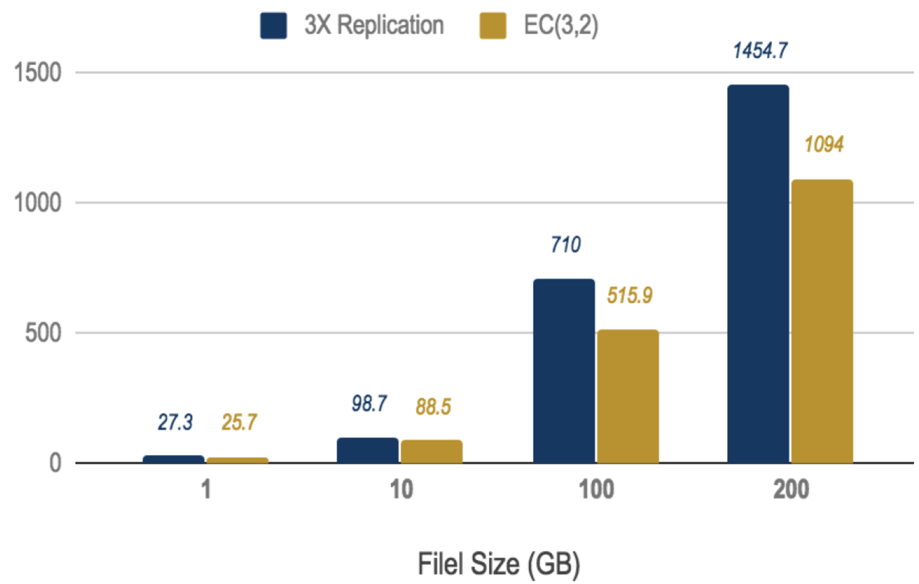
- Tolerate failures without full replication
  - Use parity blocks, *m*, from *k* blocks
- Direct data access without failure
  - Reconstruct missing block(s) from parity if there is failure

	Data Durability	Storage Efficiency
Single replica	0	100%
Three-way replication	2	33%
XOR with six data cells	1	86%
RS(6,3)	3	67%
RS(10,4)	4	71%

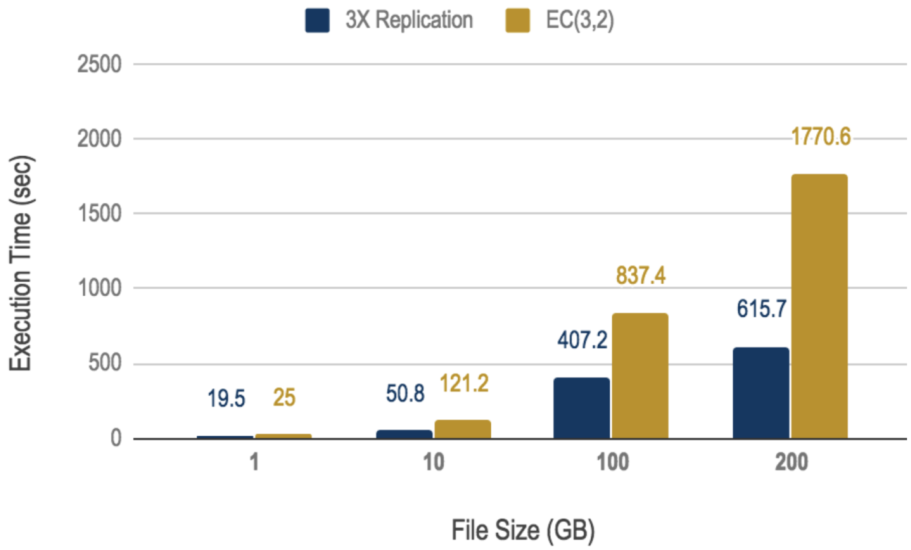


https://blog.cloudera.com/introduction-to-hdfs-erasure-coding-in-apache-hadoop/

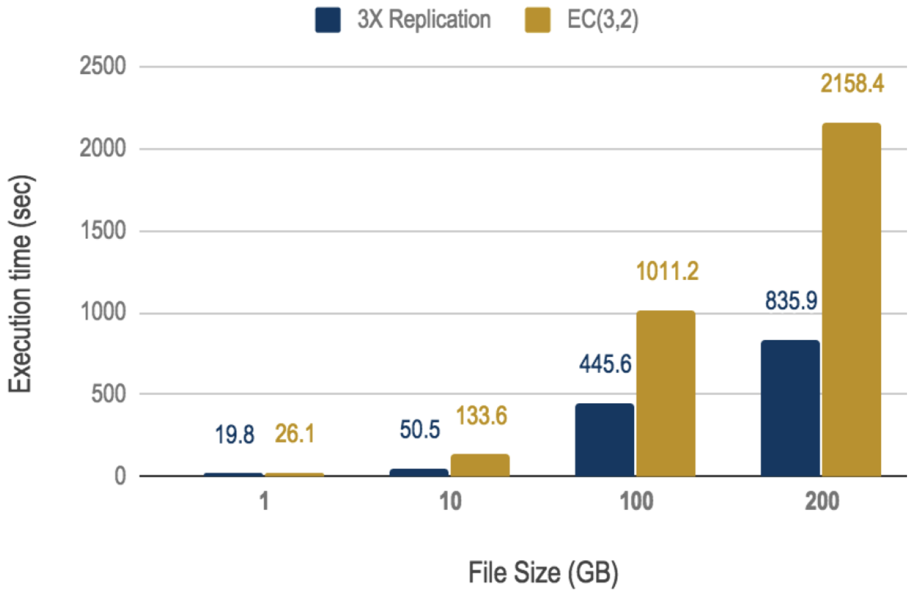
Results



(a) Writing Performance

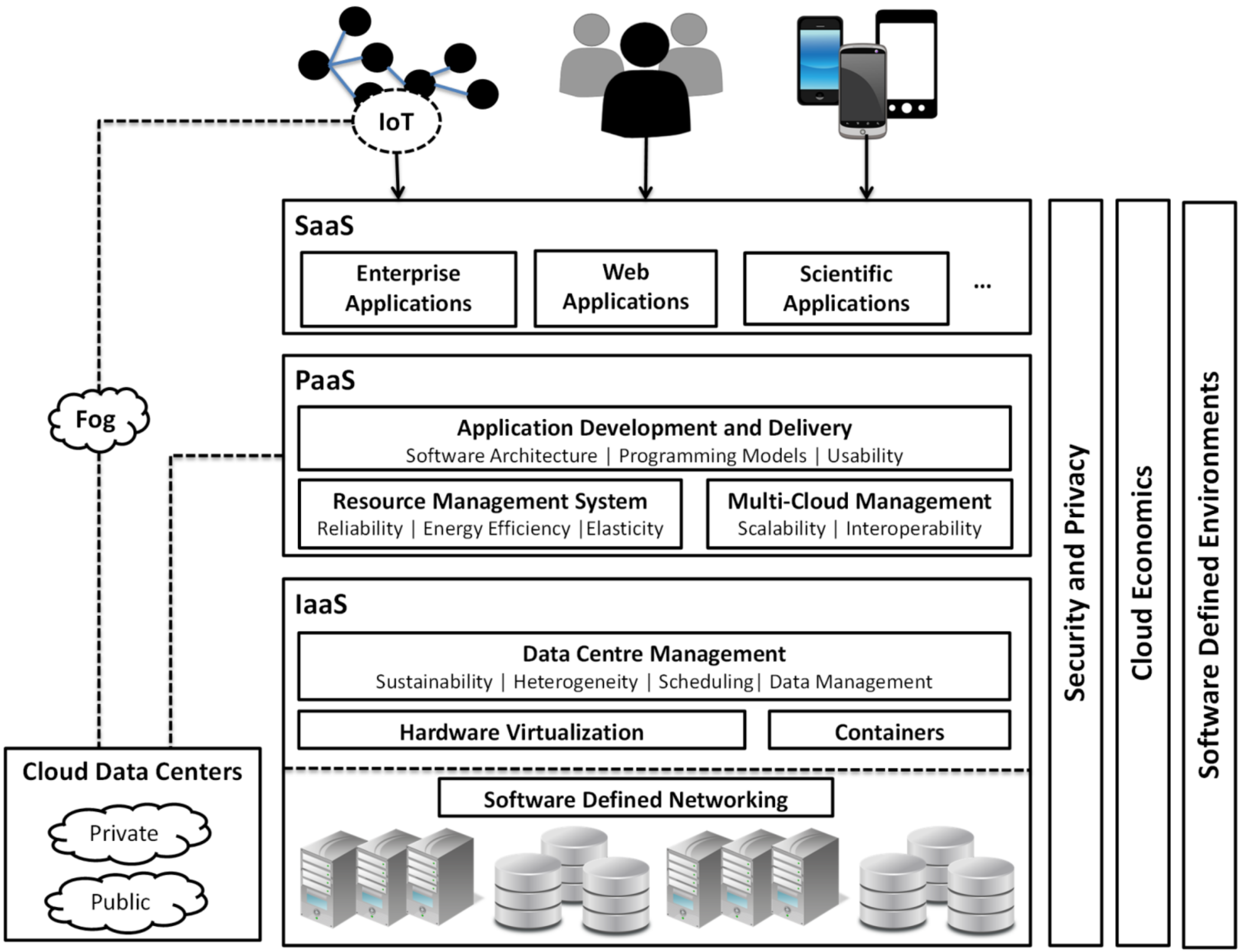


(b) Reading Performance, No Nodes Down



(c) Reading Performance, 2 Nodes Down

Cloud Ecosystem



A Manifesto for Future Generation Cloud Computing: Research Directions for the Next Decade, *ACM computing surveys (CSUR)*. 2018

Cloud Storage Categories (IaaS)

- Object storage
  - AWS S3
  - Azure Blob
- Block-level Storage
  - AWS Elastic Block Storage
  - Azure Disks
- Network file system
  - AWS Elastic File System (NFS), Lustre
  - Azure Files (NFS, SMB), HPC Cache
- Backup
  - AWS Backup
  - Azure Backup
- Sync and Transfer
  - AWS DataSync, Snow and Import/Export
  - Azure FileSync and Bulk Transfer Disks

https://aws.amazon.com/products/storage/
https://docs.microsoft.com/en-us/azure/storage/

Other Distributed Storage Systems

- Weil, Sage A., et al. "Ceph: A scalable, high-performance distributed file system." *OSDI* 2006
- Kademlia: A Peer-to-peer information system based on the XOR Metric, Petar Maymounkov and David Mazières, *International Workshop on Peer-to-Peer Systems*, 2002

Additional Reading

- HDFS Architecture Guide, D. Borthakur, 2008, *http://hadoop.apache.org/docs/stable1/hdfs_design.pdf*
- HDFS SCALABILITY: THE LIMITS TO GROWTH, Konstantin V. Shvachko, *;login:* April 2010, Volume 35, Number 2, https://www.usenix.org/publications/login/april-2010-volume-35-number-2/hdfs-scalability-limits-growth
- CoHadoop: Flexible Data Placement and Its Exploitation in Hadoop, Eltabakh et al, *VLDB, 2011*