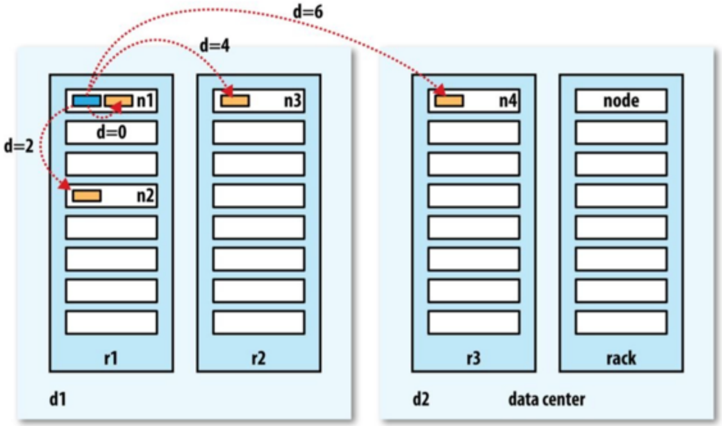
Google File System (GFS) and Hadoop Distributed File System (HDFS)

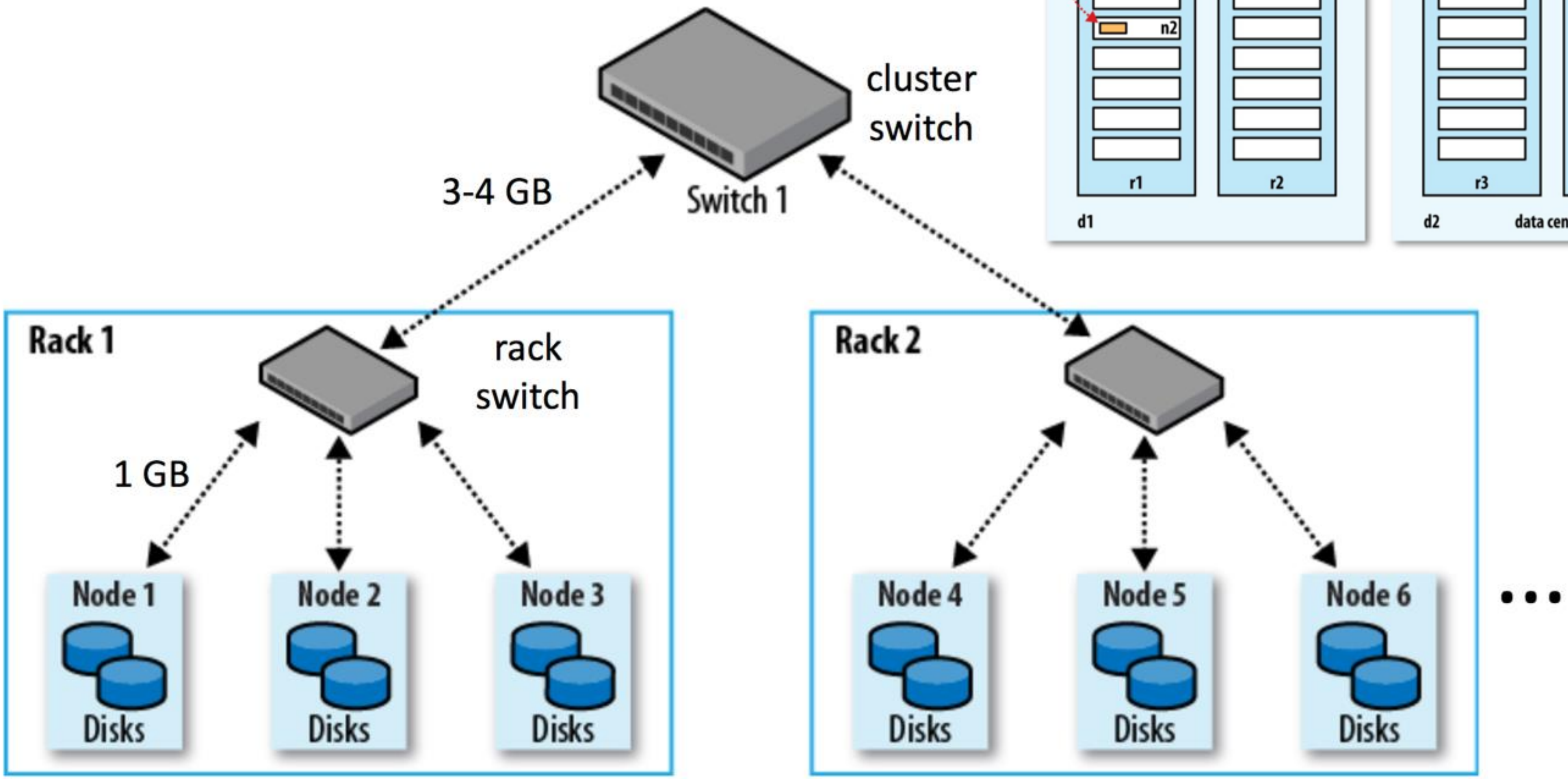
Hadoop: Architectural Design Principles

- Linear scalability
  - More nodes can do more work within the same time – Linear on data size, linear on compute resources
- **Move computation to data**
  - Minimize expensive data transfers
  - Data is large, programs are small
- Reliability and Availability: Failures are common
  - **Persistent storage**
- Simple computational model (MapReduce)
  - Hides complexity in efficient execution framework
- **Streaming data access** (avoid random reads)
  - More efficient than seek-based data access

Need for a suitable file system

A Typical Cluster Architecture





~ 30-40 servers per rack

Failures in literature

- LANL data (DSN 2006)
  - Data collected over 9 years
  - 4750 machines, 24101 CPUs Distribution of failures
    - Hardware ~ 60%,
    - Software ~ 20%,
    - Network/Environment/Humans ~ 5%,
    - Aliens ~ 25%
  - Depending on a system, failures occurred between once a day to once a month
  - Most of the systems in the survey were the cream of the crop at their time
- Disk drive failure analysis (FAST 2007)
  - Annualized Failure Rates vary from 1.7% for one year old drives to over 8.6% in three year old ones
  - Utilization affects failure rates only in very old disk drive populations
  - Temperature change can cause increase in failure rates but mostly for old drives
- Memory also fails (DRAM errors analysis, SIGMETRICS 2009)

GFS & HDFS

Distributed file systems manage the storage across a network of machines.

- GFS
  - Implemented especially for meeting the rapidly growing demands of Google’s data processing needs.
  - The Google File System, Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, SOSP’03
- HDFS
  - Hadoop has a general-purpose file system abstraction (i.e., can integrate with several storage systems such as the local file system, HDFS, Amazon S3, etc.).
  - HDFS is Hadoop’s flagship file system.
    - Implemented for the purpose of running Hadoop’s MapReduce applications.
  - Based on work done by Google in the early 2000s
  - The Hadoop Distributed File System, Konstantin Shvachko, Hairong Kuang, Sanjay Radia, Robert Chansler, IEEE2010

Roadmap

Assumptions / Requirements

Design choices

Architecture / Implementation

Assumptions

Section 2.1: The Google File System, Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung, SOSP'03

- "Modest" number of very large files
- Data access
  - write-once, read-many-times pattern
  - Large reads: Time to read the whole dataset is more important
  - Mostly, files are appended to, perhaps concurrently

- High sustained throughput favored over low latency

- Commodity hardware
  - Need fault-tolerance
  - High component failure rates: Inexpensive commodity components fail all the time

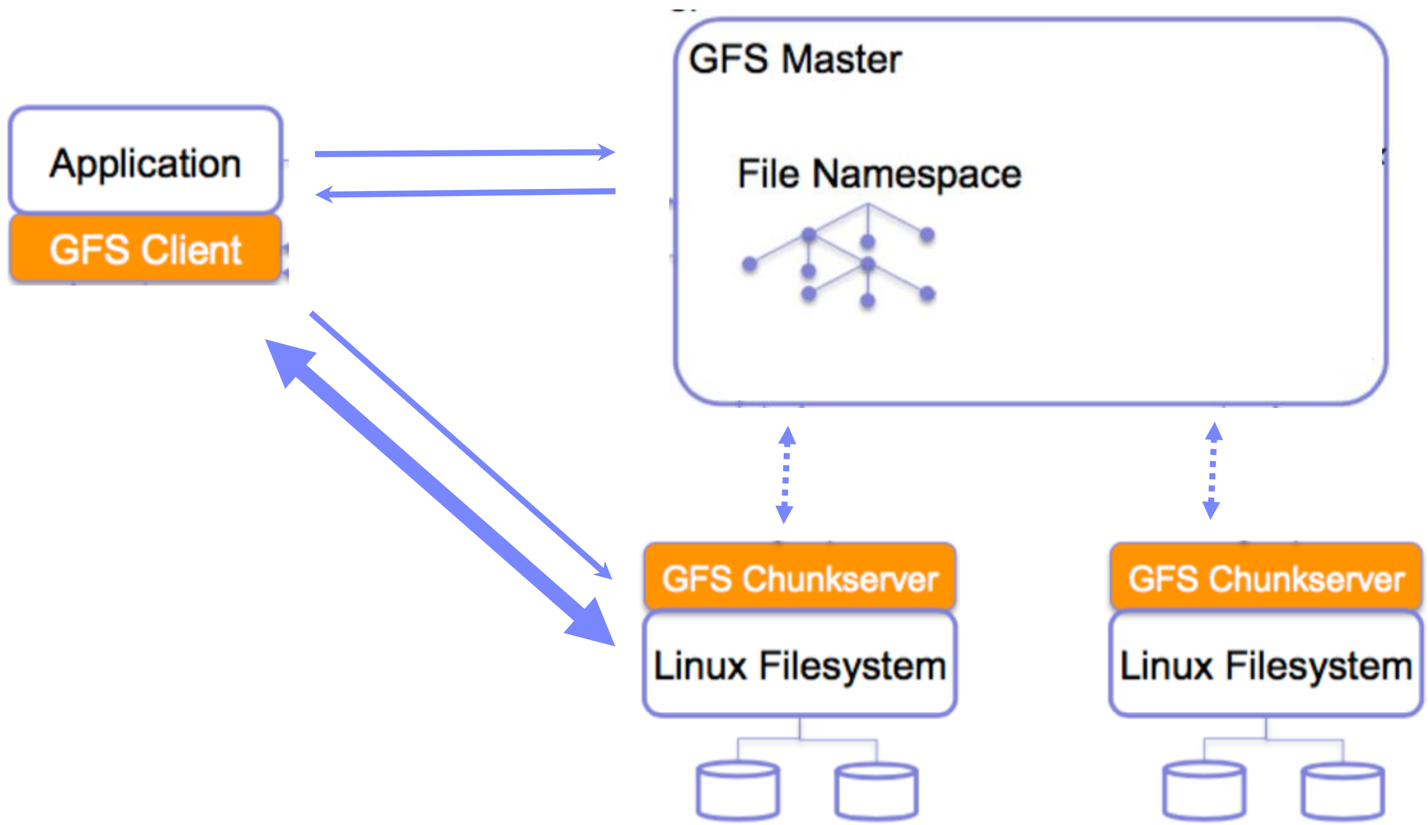
- Not a good fit for
  - low-latency data access
  - lots of small files
  - multiple writers, arbitrary file modifications

Design

- 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
- ***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

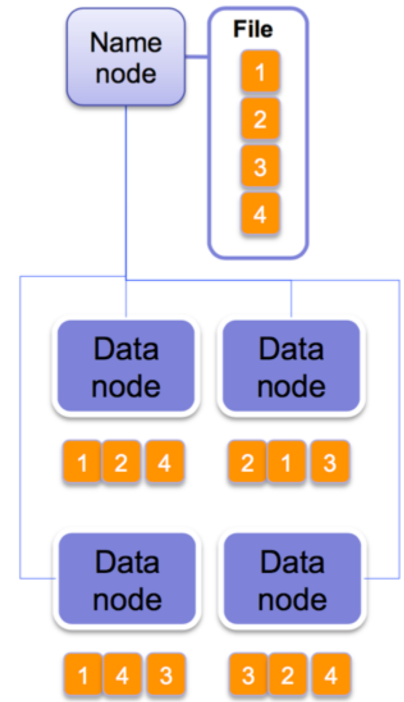
Overview



GFS Master
File Namespace
Application
GFS Client
GFS Chunkserver
Linux Filesystem
GFS Chunkserver
Linux Filesystem

Namenodes and Datanodes

- Two types of nodes:
  - One Namenode/Master
  - Multiple Datanodes/Chunkservers
- Name node manages the filesystem namespace.
  - File system tree and metadata, stored persistently
  - Block locations, stored transiently
- Data nodes store and retrieve data blocks when they are told to by clients or the Namenode.
- Data nodes report back to the Namenode periodically with lists of blocks that they are storing.



HDFS/GFS – Hadoop/MapReduce Component Naming Conventions

- MapReduce daemons
  - **JobTracker**: client communication, job scheduling, resource management, lifecycle coordination (~ master in Google MR)
  - **TaskTrackers**: task execution module (~ worker in Google MR)

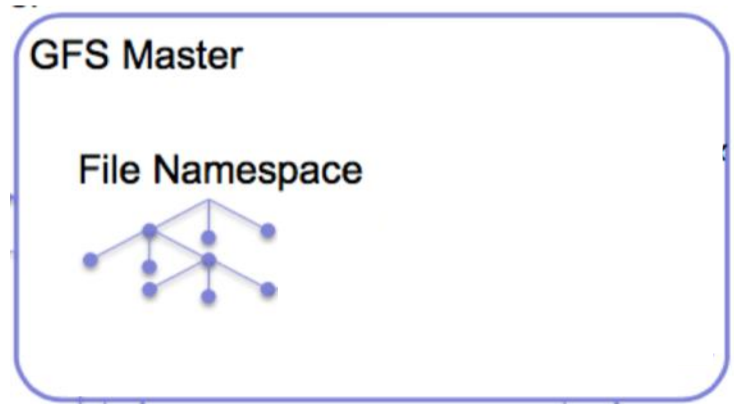
- HDFS daemons
  - **NameNode**: namespace and block management (~ master in GFS)
  - **DataNodes**: block replica container (~ chunkserver in GFS)

Blocks

- Files are broken into block-sized chunks (64 MB by default)
- With the **large** block abstraction:
  - A file can be larger than any single disk in the network
  - Storage subsystem is simplified (e.g., metadata bookkeeping)
  - Replication for fault-tolerance and availability is facilitated
  - Reduces clients' need to interact with the master
    - Reads and writes on the same chunk require only one initial request to the master for chunk location information.
    - Applications mostly read and write large files sequentially
  - Client can reduce network overhead by keeping a persistent TCP connection to the chunkserver over an extended period of time
  - Reduces the size of the metadata stored on the master.
    - This allows us to keep the metadata in memory
- Potential disadvantage: Chunkserver becomes hotspot with small file

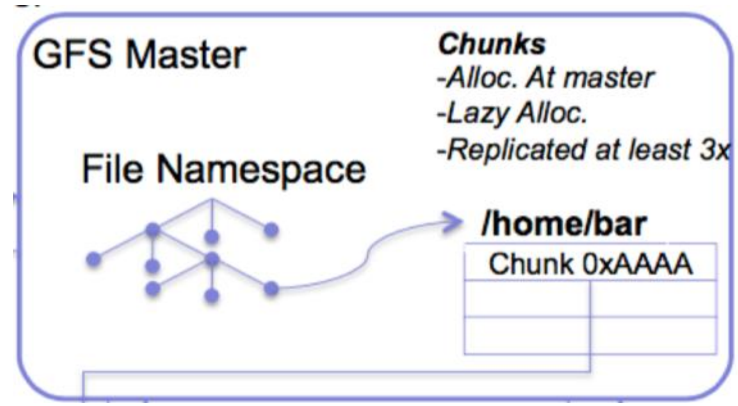
Single master

- Advantage: simplified design
  - Global knowledge of the system
- Problem:
  - **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)



Metadata

- Global metadata is stored on the master
  - File and chunk namespaces
  - Mapping from files to chunks
  - Locations of each chunk's replicas
  - Access control information



- All in memory (64 bytes / chunk)
  - **Fast**
  - Easily accessible (fast scan, e.g., for balancing)

- Master has an ***operation log*** for persistent logging of critical metadata updates
  - Persistent on local disk
  - Replicated
  - Checkpoints for faster recovery

Master’s Responsibilities

- Metadata storage
- Namespace management/locking
- Periodic communication with chunkservers
  - give instructions, collect state, track cluster health
- Chunks management
  - Creation
    - place new replicas on chunkservers with below-average disk space utilization
    - limit “recent” creations on each chunkserver. It predicts imminent heavy write traffic
    - spread replicas of a chunk across racks.
  - Re-replication: number of available replicas falls below a user-specified goal.
    - a chunkserver becomes unavailable, corrupted replica, disks is disabled because of errors, replication goal is increased.
  - Rebalancing
    - examine the current replica distribution and move replicas for better disk space and load balancing.

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
    - Periodically ask the chunkservers which blocks they have.
      Those that cannot be referenced anymore can be removed.

- Stale replica deletion
  - detect “stale” replicas using chunk version numbers

Mutations

- Mutation = write or record append
  - Must be done for all replicas
- Goal: ***minimize*** master involvement
- Lease mechanism:
  - Master picks one replica as primary; gives it a “lease” for mutations
- Data flow decoupled from control flow

Mutation is not random access

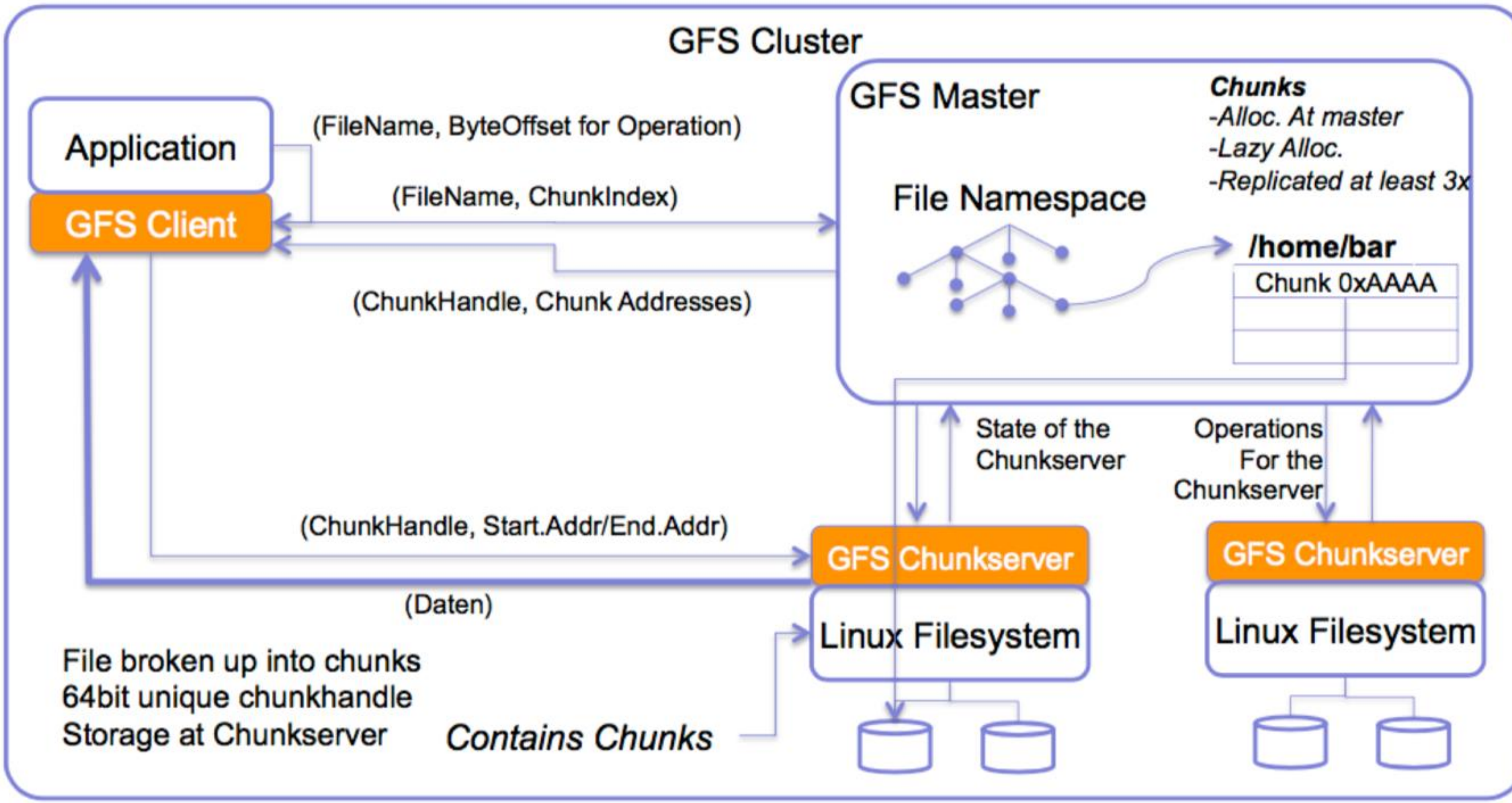
Caches

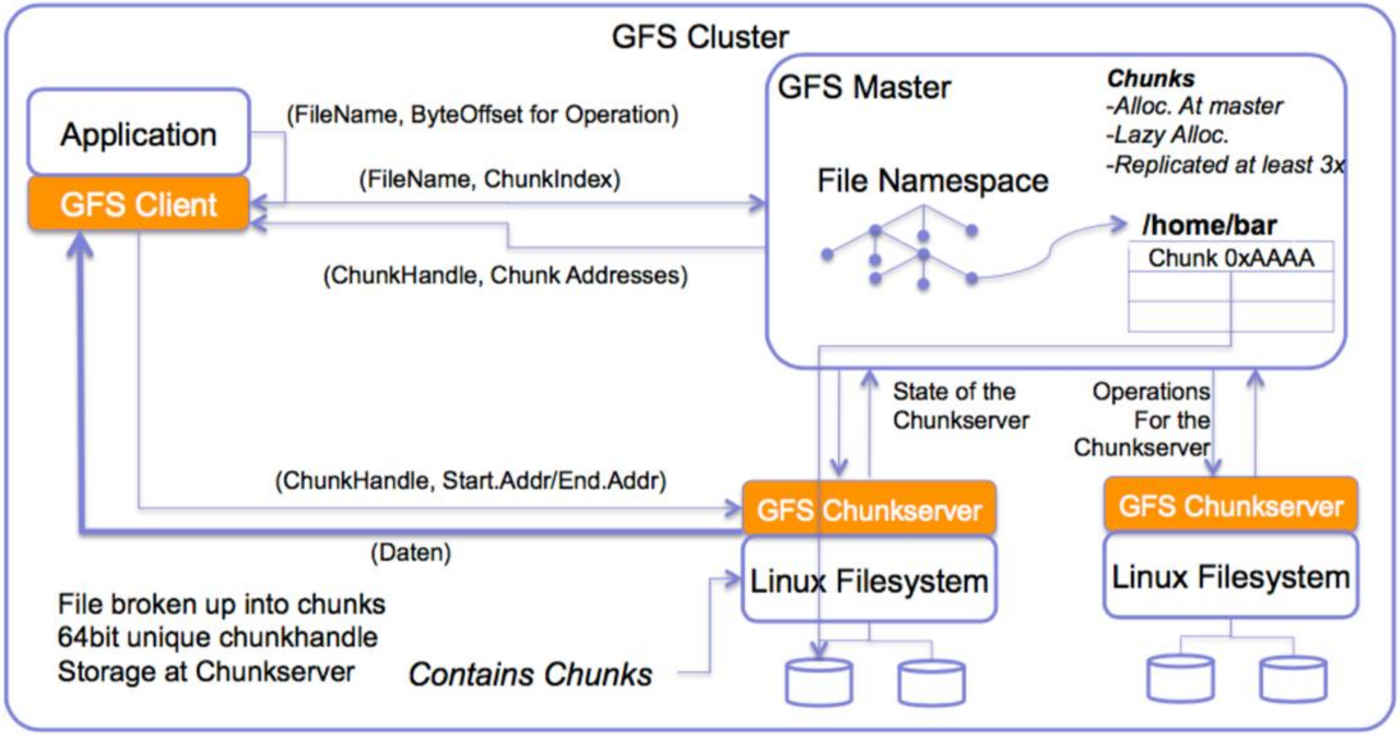
- Data is not cached by **clients** (only metadata is)
  - Access pattern: stream through huge files
  - Working set too large to be cached

- System is highly simplified: no cache coherence problem!

- **Chunkservers** do cache data *indirectly*
  - Chunks are stored as local files and so Linux's buffer cache already keeps frequently accessed data in memory

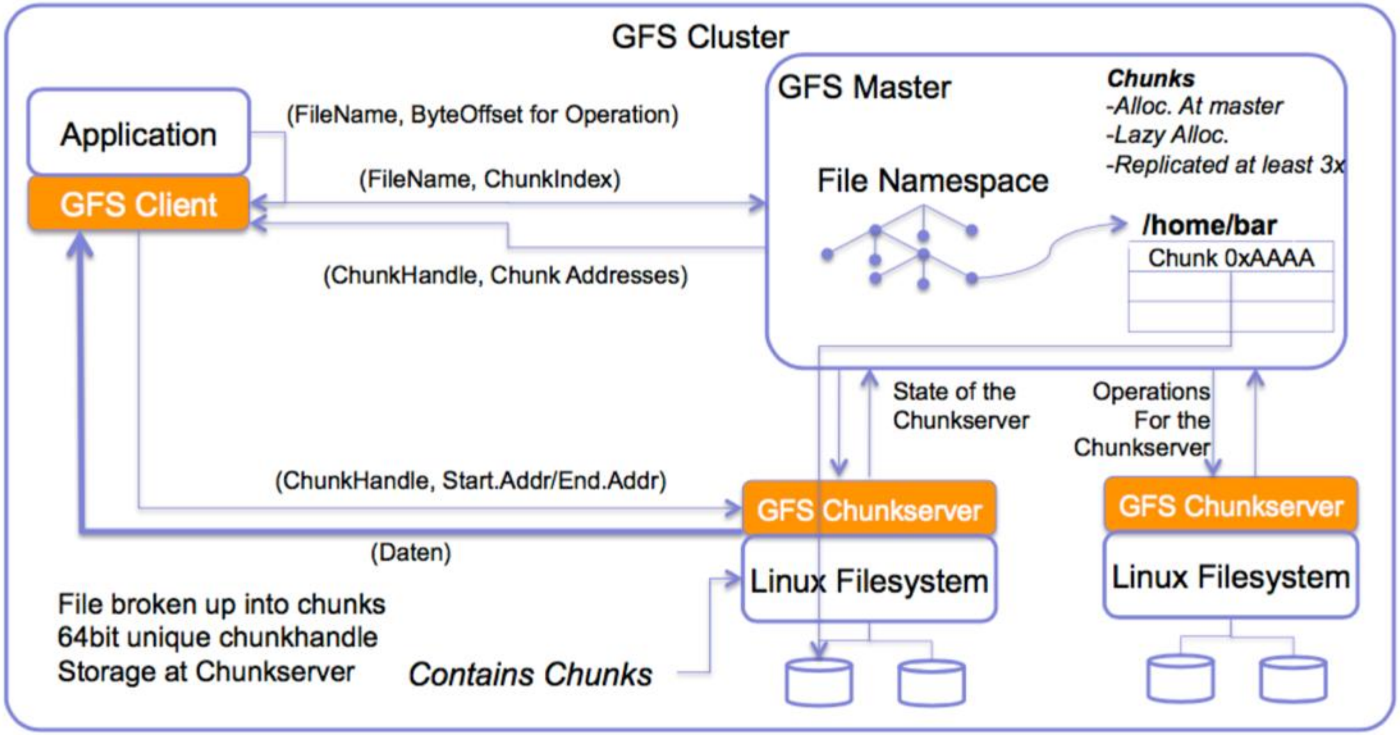
GFS - Overview





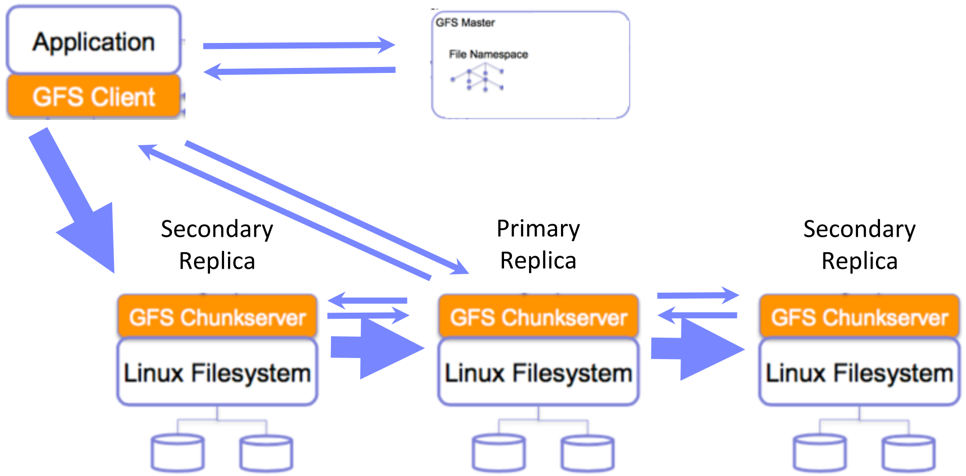
Read:

- The client sends the master the file name and chunk index.
  - Using the fixed chunk size, the client translates the file name and byte offset specified by the application into a chunk index within the file.
- The master replies with a chunk handle and locations of the replicas.
  - The client caches this information.



- The client sends a request to one of the replicas,
  - most likely the closest one.
  - The request specifies the chunk handle and a byte range within that chunk.
  - Further reads of the same chunk require no more client-master interaction
- The client typically asks for multiple chunks in the same request
  - The master can also include the information for chunks immediately following those requested.
  - Sidesteps several future client-master interactions.

Write

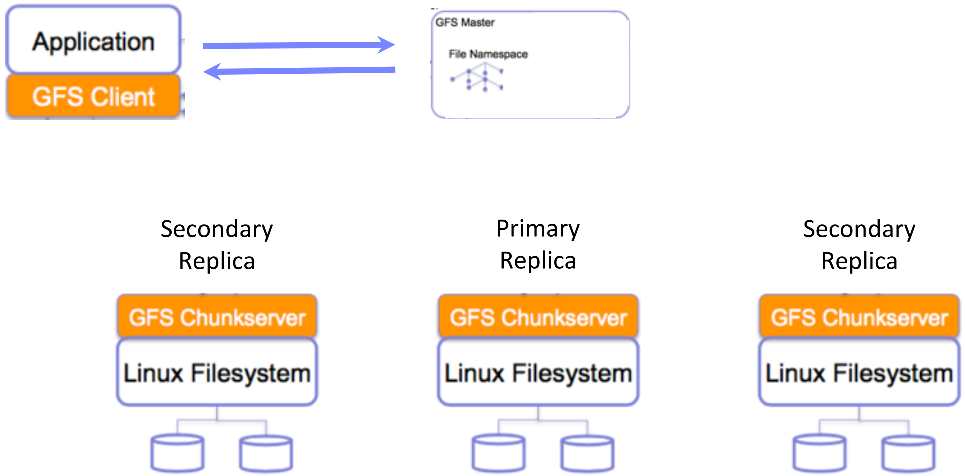


- A primary replica holds a **lease**: coordinate writing on a chunk
- Separate data flow from control flow

Write

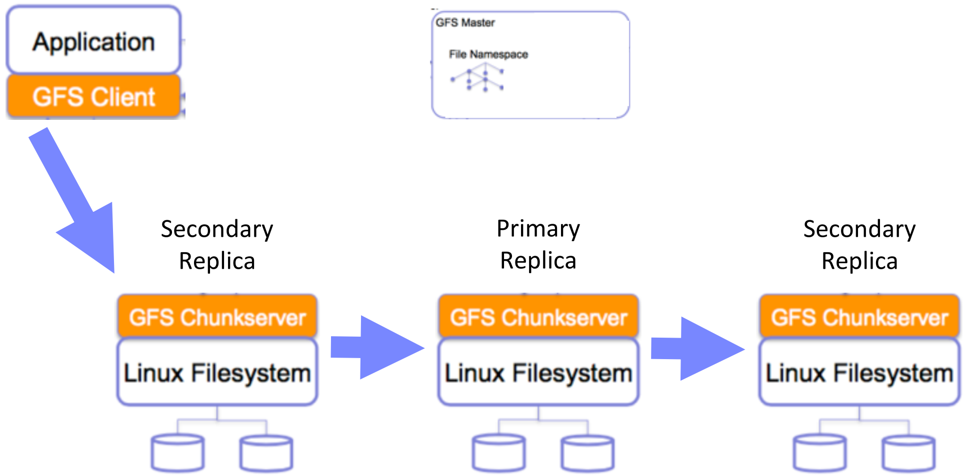
- The master grants a chunk **lease** to one of the replicas, which we call the primary.
  - The primary picks a serial order for all mutations to the chunk.
  - All replicas follow this order when applying mutations.
- A lease has an initial timeout of 60 seconds.
  - The primary can request and typically receive extensions
  - HeartBeat messages regularly exchanged between master and chunkservers.
  - If the master loses communication with a primary, it can safely grant a new lease to another replica after the old lease expires.

Write



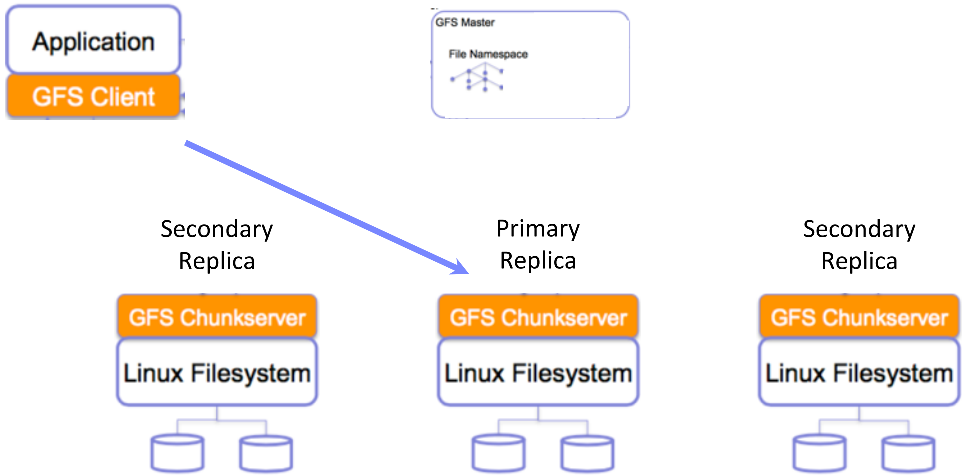
- The client asks the master which chunkserver holds the current lease for the chunk and the locations of the other replicas.
  - If no one has a lease, the master grants one to a replica it chooses
- The master replies with the identity of the primary and the locations of the other (secondary) replicas.
  - The client caches this data for future mutations.
  - It needs to contact the master again only when the primary becomes unreachable or replies that it no longer holds a lease

Write



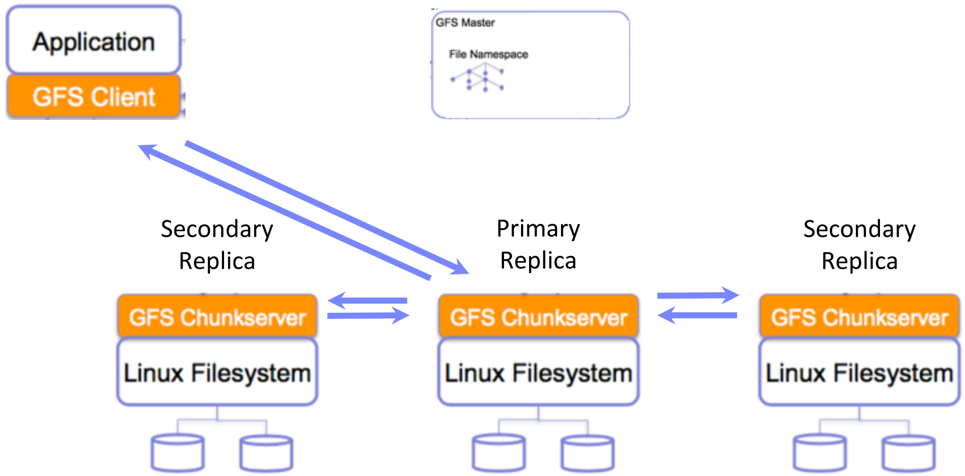
- The client pushes the data to all the replicas.
- Each chunkserver will store the data in an internal LRU buffer.
  - Decoupling the data flow from the control flow, improve performance by scheduling the expensive data flow based on the network topology
- Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary.

Write



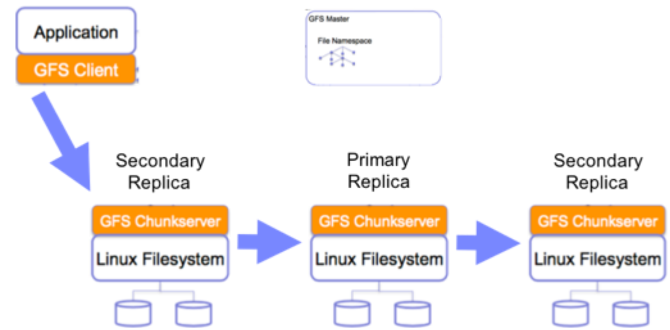
- Once all the replicas have acknowledged receiving the data, the client sends a write request to the primary.
  - The request identifies the data pushed earlier to all of the replicas.
  - The primary assigns consecutive serial numbers to all the mutations it receives, possibly from multiple clients, which provides serialization.

Write



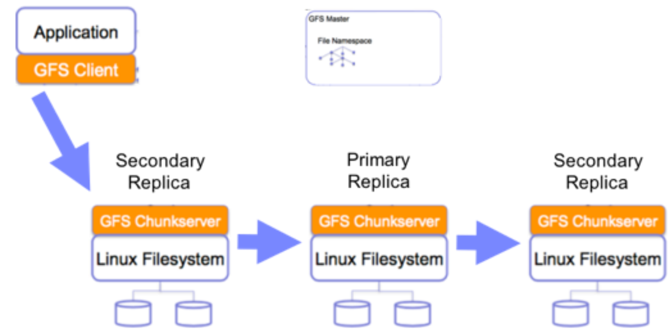
- The primary forwards the write request to all secondary replicas.
- Each secondary replica applies mutations in the same serial number order assigned by the primary.
- The secondaries all reply to the primary indicating that they have completed the operation.
- The primary replies to the client.
  - Any errors encountered at any of the replicas are reported to the client.

Write: Decouple data and control flow



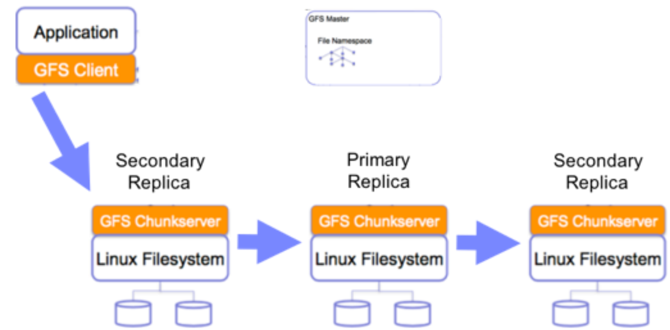
- Control flows from client to the primary and then to all secondaries
- Data is pushed linearly along a carefully picked chain of chunkservers in a **pipelined** fashion.
  - Avoid network bottlenecks and high-latency links (e.g., inter-switch links): each machine forwards data to the "closest" machine in the network.
- Once a chunkserver receives some data, it starts forwarding immediately.
  - Switched network with full-duplex links.
  - Sending the data immediately does not reduce the receive rate.

Write: Decouple data and control flow



- Control flows from client to the primary and then to all secondaries
- Data is pushed linearly along a carefully picked chain of chunkservers in a **pipelined** fashion.
  - Avoid network bottlenecks and high-latency links (e.g., inter-switch links): each machine forwards data to the "closest" machine in the network.
- Once a chunkserver receives some data, it starts forwarding immediately.
  - Switched network with full-duplex links.
  - Sending the data immediately does not reduce the receive rate.

Write: Time estimation



- Ignore network congestion
- Transfer B bytes to R replicas

$$B/T + RL$$

  - T is the network throughput
  - L is latency to transfer bytes between two machines.

- E.g., network links are typically 100 Mbps (T), and L is far below 1 ms. Therefore, 1 MB can ideally be distributed in about 80 ms. (2003)

References

- **“Web Search for a Planet: The Google Cluster Architecture”,** L. Barroso, J. Dean, U. Hoelzle, IEEE Micro 23(2), 2003.
- **“The Google File System”,** S. Ghemawat, H. Gobioff, S. Leung, SOSP 2003.
- **“The Hadoop Distributed File System”**, K. Shvachko et al, MSST 2010.
- **“Hadoop: The Definitive Guide”**, T. White, O’Reilly, 3rd edition, 2012.

Questions?

Sources & References

General intro to bigdata

- http://www.systems.ethz.ch/sites/default/files/file/BigData_Fall2012/BigData-2012-M5-1%20updated.pdf

Storage systems and big data

- www.cs.colostate.edu/~cs655/lectures/CS655-L4-StorageSystems.pdf

Specific on GFS

- https://courses.cs.washington.edu/courses/cse490h/08wi/lecture/lec6.ppt

Specific on GFS

- http://prof.ict.ac.cn/DComputing/uploads/2013/DC_4_0_GFS_ICT.pdf