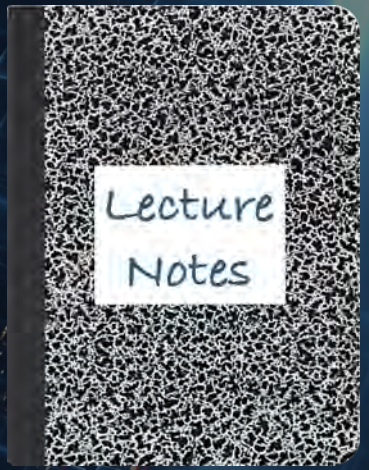
CS 417 – DISTRIBUTED SYSTEMS



Week 7: Decentralized Storage
Part 1: Distributed File Systems

Paul Krzyzanowski

Network-Attached Storage: client-server file systems

- Network Attached Storage is built on a central server architecture
  - Point of congestion, single point of failure
- Alleviate performance somewhat with replication and client caching
  - E.g., Coda, tokens (aka *leases*, *oplocks*)
  - Limited replication can lead to congestion
- But file data is still centralized
  - A file server stores all data from a file – not split across servers
  - Even if replication is in place, a client downloads all data for a file from one server
- File sizes are limited to the capacity available on a server
  - What if you need a 1,000 TB file?

Distributed File Systems

- **Conventional file systems**
  - Store data & metadata on the same storage device
  - Example:
    - Linux directories are just files that contain lists of names & inodes
    - inodes are data structures placed in well-defined areas of the disk that contain information about the file
- **Distributed file systems**
  - File data can span multiple servers
  - Metadata can be on separate servers from the data
    - **Metadata** = information about the file
    - Includes name, access permissions, timestamps, file size, & locations of data blocks
    - **Data** = actual file contents

Google File System (GFS)

(≈ Apache Hadoop Distributed File System, HDFS)

Google File System (GFS) Workload Assumptions

- **A relatively small number of huge files** (thousands to a few million)
  - Assumptions for conventional file systems don't work
    - E.g., "*most files are small*", "*many files have short lifetimes*"
- **File access**
  - Most files are appended, not overwritten
    - Random writes within a file are almost never done
    - Once created, files are mostly read; often sequentially
  - Workload is mostly:
    - Reads: **large streaming reads** (from some offset), small random reads
    - **Large concurrent appends**: hundreds of processes may append to a file concurrently
- **Commodity hardware is used** (PCs running Linux)
- Needs to deliver high performance to a **large number of clients**
  - I/O performance is important
  - … and the ability to support concurrent appends

GFS Design Goals

- **Scalable** distributed file system: add more servers for more capacity
- **Support huge files**. Multi-TB files are the norm
  - It doesn't make sense to work with billions of n-KB-sized files
  - This impacts I/O operations and block size choices
- **Fault-tolerance** is critical
  - Component failures are the norm
  - File system = thousands of storage machines
  - Some % are not working at any given time
- Deliver **high performance** to a large number of clients

Basic Design Principles

- Use separate servers to store metadata
  - Metadata includes lists of (*server, block_number*) sets that identify which blocks on which servers hold file data
  - We need more bandwidth for data access than metadata access
    - Metadata is small; file data can be huge
- Use large logical blocks
  - Most "normal" file systems are optimized for small files
    - A block size is typically 4KB
  - Expect huge files, so use *huge blocks* … >1,000x larger
    - The list of blocks that makes up a file becomes easier to manage
- Replicate data
  - Expect some servers to be down
  - Store copies of data blocks on multiple servers

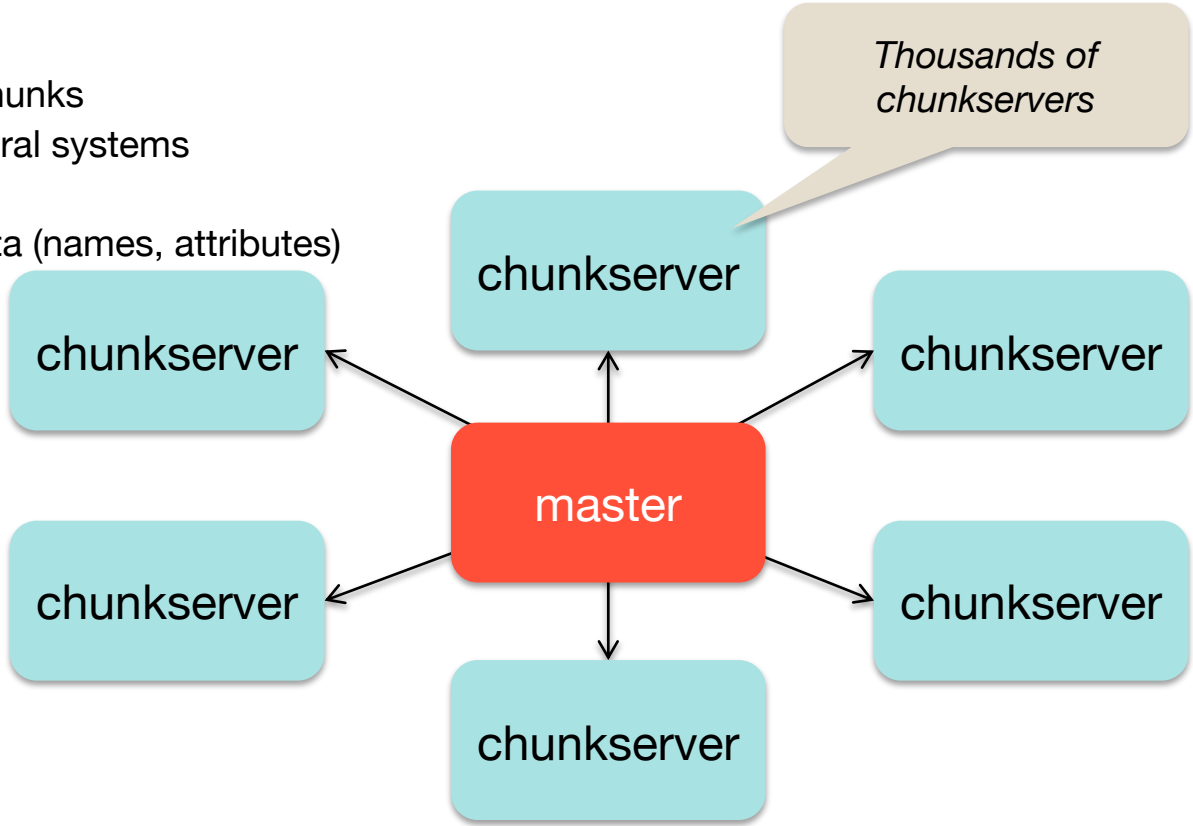
File System Interface

- GFS does *not* have a standard OS-level API
  - No POSIX system call level API – no kernel/VFS implementation
  - User-level API for accessing files
  - GFS servers are implemented in user space using native Linux FS
- Files organized hierarchically in directories
- Operations
  - Basic operations
    - *Create, delete, open, close, read, write*
  - Additional operations
    - *Snapshot:* create a copy of a file or directory tree at low cost
    - *Append:* allow multiple clients to append atomically without locking

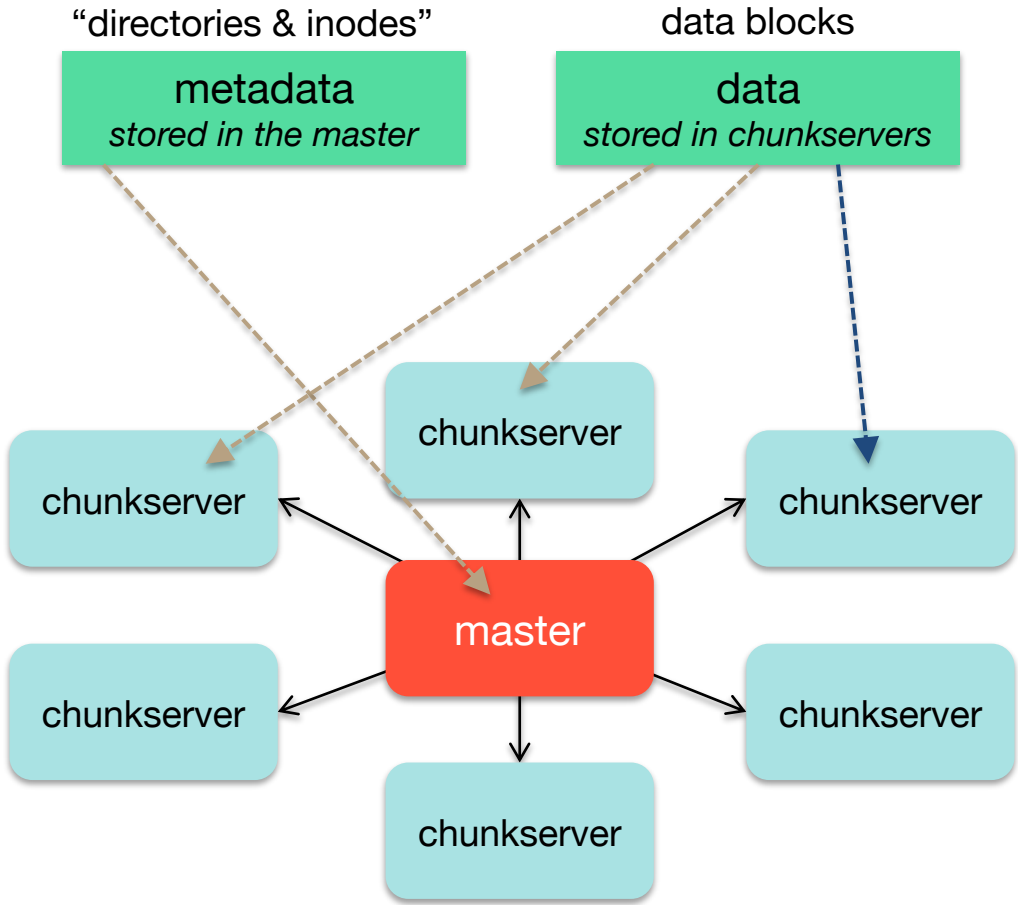
GFS Master & Chunkservers

GFS cluster

- Multiple **chunkservers**
  - Data storage: fixed-size chunks
  - Chunks replicated on several systems
- One **master**
  - Stores file system metadata (names, attributes)
  - Maps files to chunks



GFS Master & Chunkservers in a GFS cluster



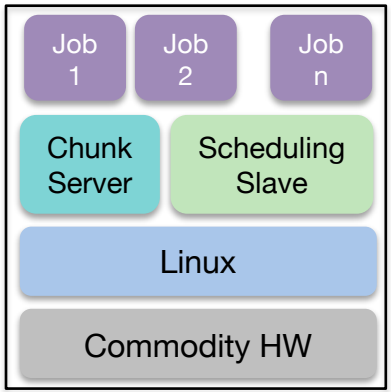
Core Part of Google Cluster Environment

Google Cluster Environment

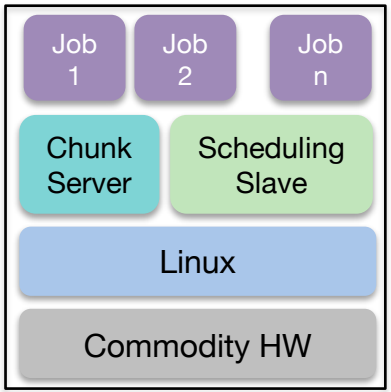
- Core services: GFS + cluster scheduling system
- Typically, 100s to 1000s of active jobs
- 200+ clusters, many with 1000s of machines
- Pools of 1000s of clients
- 4+ PB file systems, 40 GB/s read/write loads

Bring the computation close to the data

Applications coexist on the same set of servers as the file data



Machine 1



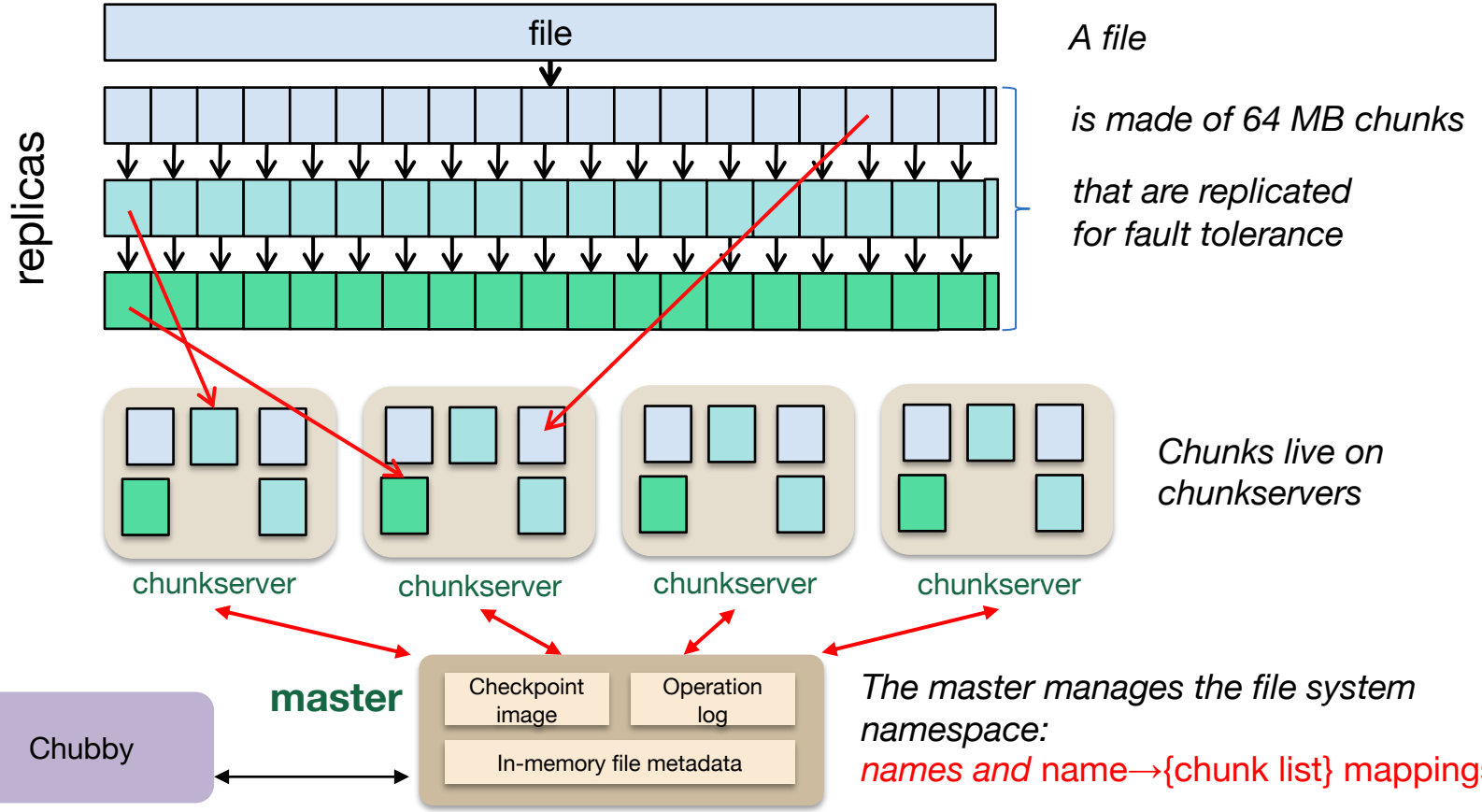
Machine *n*

GFS Master — File system master

Scheduling Master — Job scheduler

Chubby Lock Service — Lease (lock) manager

Chunks + Replication



Chunks and Chunkservers

- Chunk size = 64 MB (default)
  - Chunkserver stores a 32-bit checksum with each chunk
    - In memory & logged to disk: allows it to detect data corruption
- **Chunk Handle**: identifies a chunk
  - Globally unique 64-bit number
  - Assigned by the master when the chunk is created
- **Chunkservers** store chunks on local disks as Linux files
- Each chunk is **replicated** on multiple chunkservers
  - Three replicas (different levels can be specified)
  - Popular files may need more replicas to avoid hotspots

Master: metadata

- Maintains all file system metadata – **cached in memory**
  - Namespace
  - Filename to chunk ID mappings
  - Current locations of chunk replicas
- Manages
  - Chunk leases (locks)
  - Garbage collection (freeing unused chunks)
  - Chunk migration (copying/moving chunks)
- Fault tolerance
  - Operation log replicated on multiple machines
  - New master can be started if the master fails
- Periodically communicates with all chunkservers
  - Via heartbeat messages to get state and send commands

Role of Chubby:

- **Master election**
- **Small, critical data**
  - root of namespace
  - access control info
  - Bootstrap info
- **Master failure detection**
  (get notified if the master is no longer holding a lock)

One master = simplified design

- All metadata stored in master's memory
  - Super-fast access
- Namespaces and *name-to-chunk_list* maps
  - Stored in memory
  - Also persist in an ***operation log*** on the disk
    - Replicated onto remote machines for backup
- **Operation log**
  - Similar to a journal
  - All operations are logged
  - Periodic checkpoints (stored in a B-tree) to avoid playing back entire log
- Master does not store chunk locations persistently
  - This is queried from all the chunkservers: avoids consistency problems

On startup, the master polls chunkservers to find what chunks they have. This avoids the problem of the master getting out of sync

Why Large Chunks?

- Default chunk size = 64MB
  (for comparison, Linux ext4 block sizes: typically, 4 KB and up to 1 MB)
- Reduces space on the master: master stores <64 bytes of metadata for each 64MB chunk
- Reduces need for frequent communication with the master to get chunk location info –
  one query can give info on the location of lots of data
- Clients can easily cache the list of chunk locations to refer to all the data in large files
  - Cached data has timeouts to reduce the possibility of reading stale data
- Large chunks makes it feasible to keep a TCP connection open to a chunkserver for an extended time
  - Reduce the overhead of setting up a connection

Client Interaction Model

- GFS client code linked into each app
  - No OS-level API – you have to use a library
  - Interacts with master for metadata-related operations
  - Interacts directly with chunkservers for file data
    - All reads & writes go directly to chunkservers
    - Master is not a point of congestion

- Neither clients nor chunkservers cache data
  - Except for the caching by the OS system buffer cache
  - Clients cache metadata – e.g., location of a file's chunks

Reading Files

1. Contact the master
2. Get file's metadata: list chunk handles
3. Get the location of each of the chunk handles
   - Multiple replicated chunkservers per chunk
   - Some might be local to your machine or rack
4. Contact any available chunkserver directly for chunk data

Topology-aware replica selection

Writing to Files

- Less frequent than reading

- Master grants a **chunk lease** to one of the replicas
  - This replica will be the **primary replica** chunkserver
  - The lease lasts for the duration of the file modification
    - Primary can request lease extensions, if needed
  - Master increases the chunk version number and informs replicas

Writing: Phase 1 – Pipelining Data

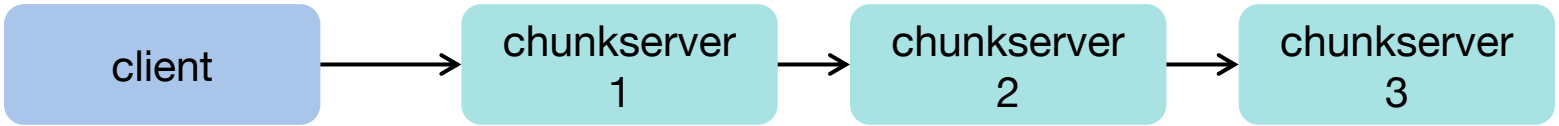
Phase 1: Send data

Deliver data but don't write to the file

- Client asks the master for a list of chunkservers with replicas: primary & secondaries
- Client writes to the closest replica chunkserver that has not received the data
  - Replica forwards the data to another replica chunkserver
  - That chunkserver forwards to another replica chunkserver …
- Chunkservers store this data in a cache – *it's not part of the file yet*

Goal: Maximize bandwidth via pipelining

Minimize latency by forwarding data while it is being received



Writing: Phase 2 – Primary Manages Serialization

Phase 2: Write data

Add it to the file (commit)

- Client waits for replicas to acknowledge receiving the data
- Sends a *write* request to the primary, identifying the data that was sent
- The primary is responsible for serialization of writes
  - Assigns consecutive serial numbers to all writes that it received
  - Applies writes in serial-number order and forwards write requests in that order to secondaries
- Once all acknowledgments have been received, the primary acknowledges the client

Writing: Separating Data & Control

Data Flow (*phase 1*) is different from **Control Flow** (*phase 2*)

- Data Flow (*upload*):
  - Client to chunkserver to chunkserver to chunkserver…
  - Order does not matter
- Control Flow (*write*):
  - Client to primary; primary to all secondaries
  - Locking used; Order maintained

Chunk version numbers are used to detect if any replica has stale data (was not updated because it was down)

Namespace

- No per-directory data structure like most file systems
  - E.g., directory file contains names of all files in the directory
- No aliases (hard or symbolic links)
- The namespace is a single lookup table
  - Maps pathnames to metadata

HDFS: Hadoop Distributed File System

Apache Hadoop Project

Framework for distributed processing of large data sets across distributed clusters

Hadoop components include:

Storage

- HDFS: Large-scale distributed file system
- HBase: Scalable NoSQL database that supports structured data storage for large tables

Computation/Data Processing

- MapReduce: Framework for distributed processing of large data sets on compute clusters.
- Spark: Framework for fast, in-memory processing.
- Mahout, MLib: Machine learning and data mining libraries.

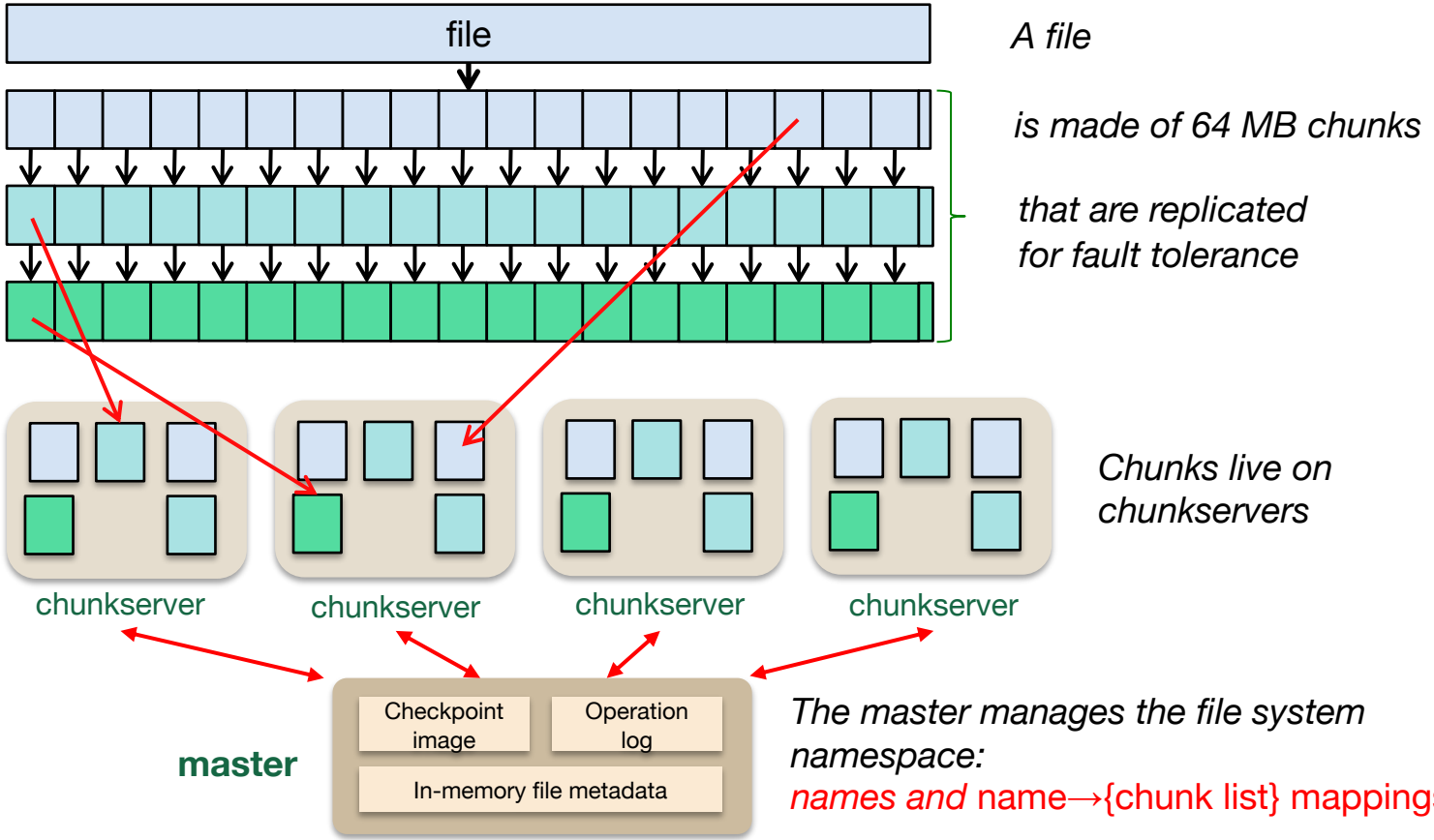
Data Access

- Hive, Pig: Tools for querying and data summarization
- Solr, Lucene: Tools for text searching and indexing

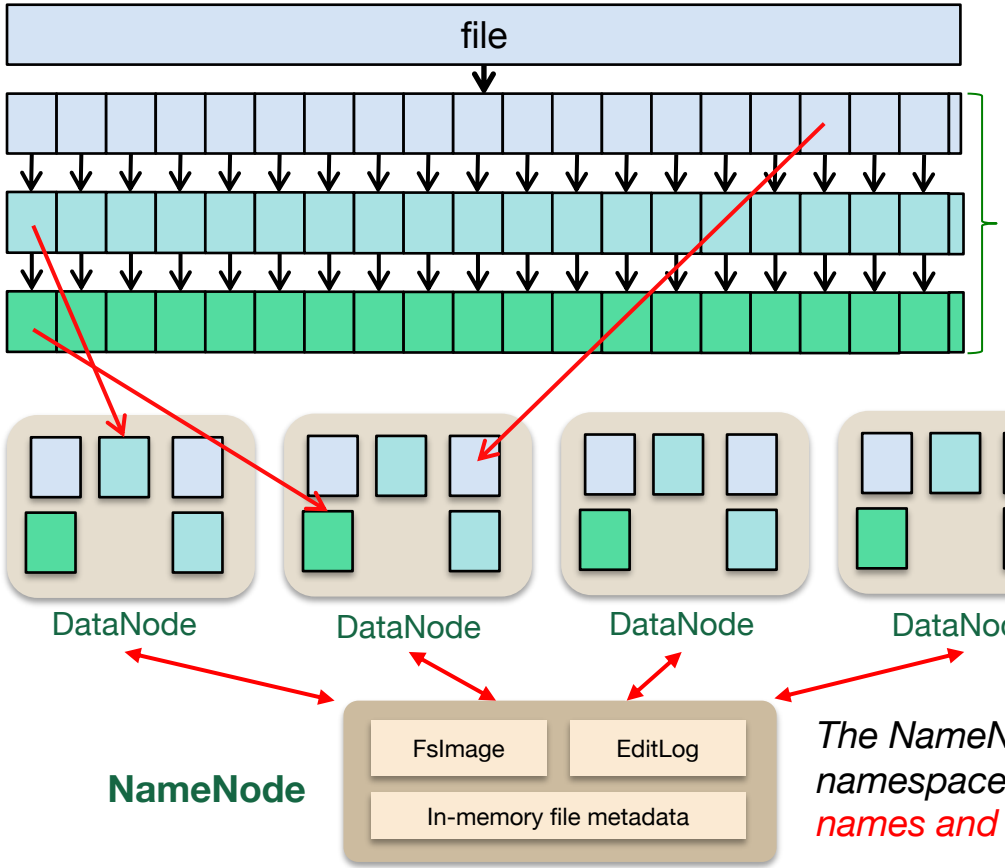
Management & Coordination

- ZooKeeper™: A high-performance coordination service for distributed applications
- YARN: Submit & schedule jobs; tracks client resources
- Oozie: Server-based workflow scheduling system to orchestrate dependencies

GFS Files


file
A file
is made of 64 MB chunks
that are replicated
for fault tolerance
chunkserver
chunkserver
chunkserver
chunkserver
Chunks live on
chunkservers
master
Checkpoint
image
Operation
log
In-memory file metadata
The master manages the file system
namespace:
names and name→{chunk list} mappings

HDFS: same stuff … different names



A file

is made of 128 MB blocks

that are replicated for fault tolerance

Blocks live on DataNodes

The NameNode manages the file system namespace:
names and name→{block list} mappings

Beyond GFS

GFS's master created scaling bottlenecks as Google grew

Colossus: kept same idea of storing data in large chunks

- **Eliminated single-master bottleneck**
  - Moved metadata into a distributed environment backed by Bigtable
- **Scaled metadata far beyond GFS**
  - Storing metadata in Bigtable let Colossus scale more than 100x the size of the largest GFS clusters
- **Stronger background repair & rebalancing**
  - Background storage managers to handle rebalancing & recovery

Dropbox

Dropbox

Dropbox: designed as a cloud-based file storage/synchronization service

- Store files on one PC; they're magically available on another PC
- Not network-attached storage; files remain local but get copied to the server

- Consumer-facing version of a file system that also:
  - Separates file data from metadata
  - Manages file data as independent chunks that may live on different servers

Dropbox file synchronization

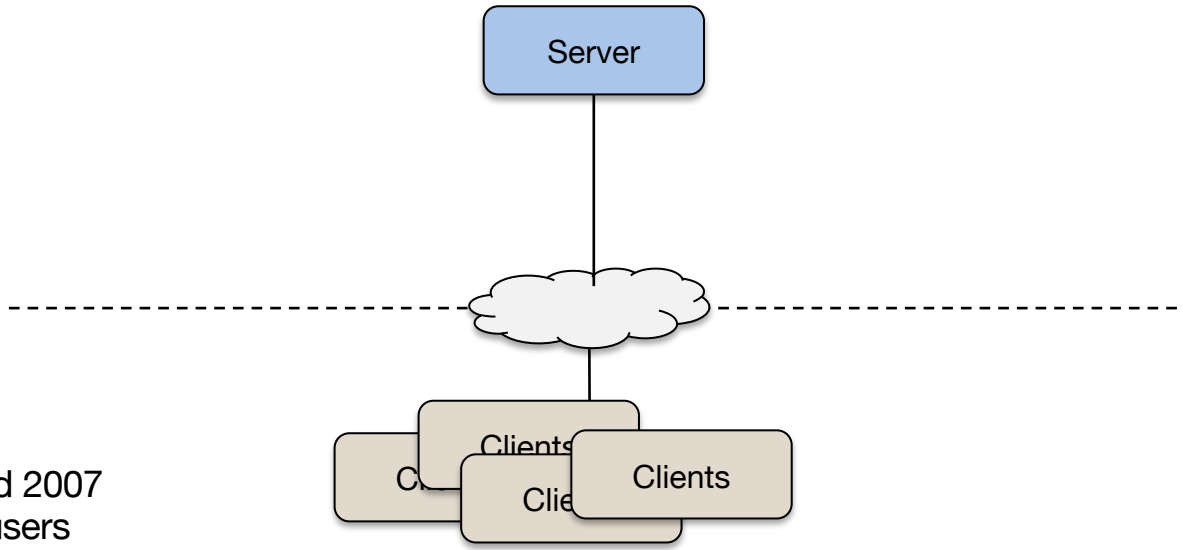
- Client runs on desktop
- Uploads any changes made within a dropbox folder
- Huge scale: 100+ million users syncing 1 billion files per day
- Design
  - Small client that doesn't take a lot of resources
  - Expect the possibility of low bandwidth to the user
  - Scalable back-end architecture
  - 99%+ of code was written in Python
    ⇒infrastructure (storage, monitoring) software migrated to Go in 2013
    ⇒Core services migrated from Go and Python to Rust for performance & memory efficiency
    ⇒Frontend services are still in Python

What's different about Dropbox?

- Most web-based apps have high read to write ratios
  - E.g., *twitter*, *facebook*, *reddit*, … 100:1, 1000:1, or higher
- But with Dropbox…
  - Everyone's computer has a complete copy of their Dropbox
  - Traffic happens only when changes occur
  - The file download:upload ratio roughly 1:1
    - Huge number of uploads compared to traditional services
- Must abide by most consistency requirements … sort of
  - Atomic: don't share partially-modified files
  - Consistent:
    - Operations have to be in order and reliable
    - Cannot delete a file in a shared folder but have others see
  - Durable: Files cannot disappear

Dropbox: architecture evolution: version 1

– One server: web server, app server, mySQL database, sync server

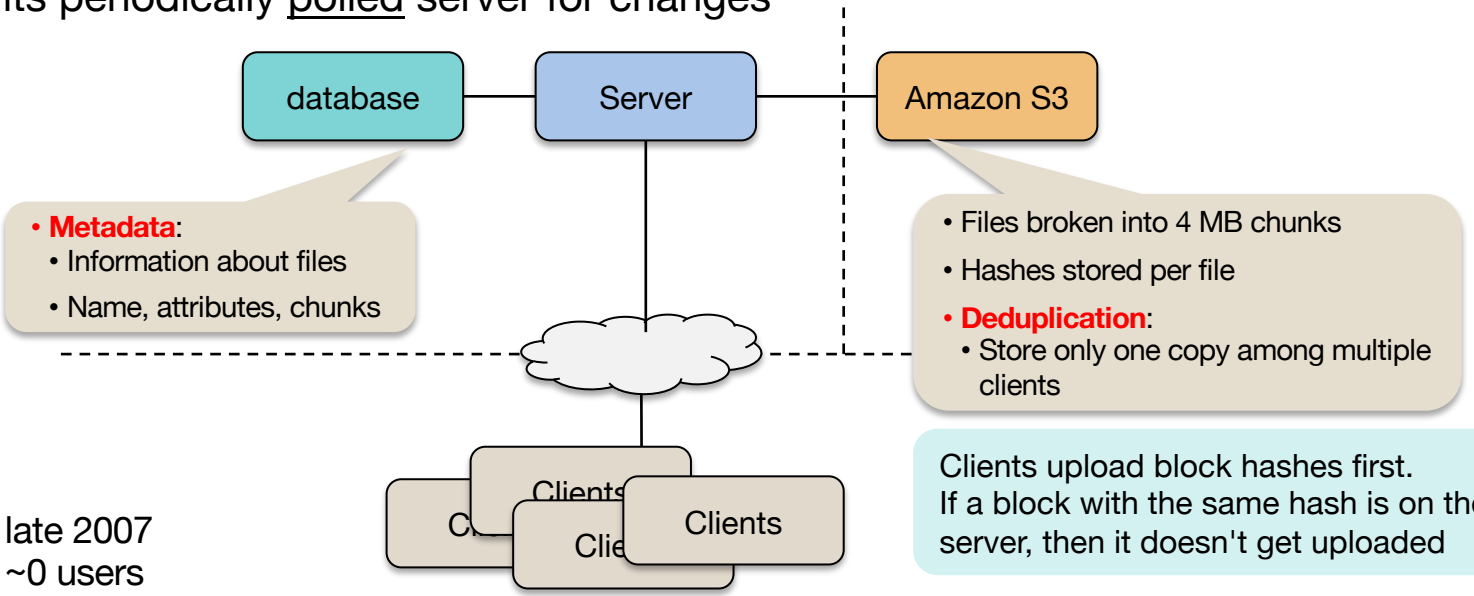


mid 2007
0 users

See http://youtu.be/PE4gwstWhmc

Dropbox: architecture evolution: version 2

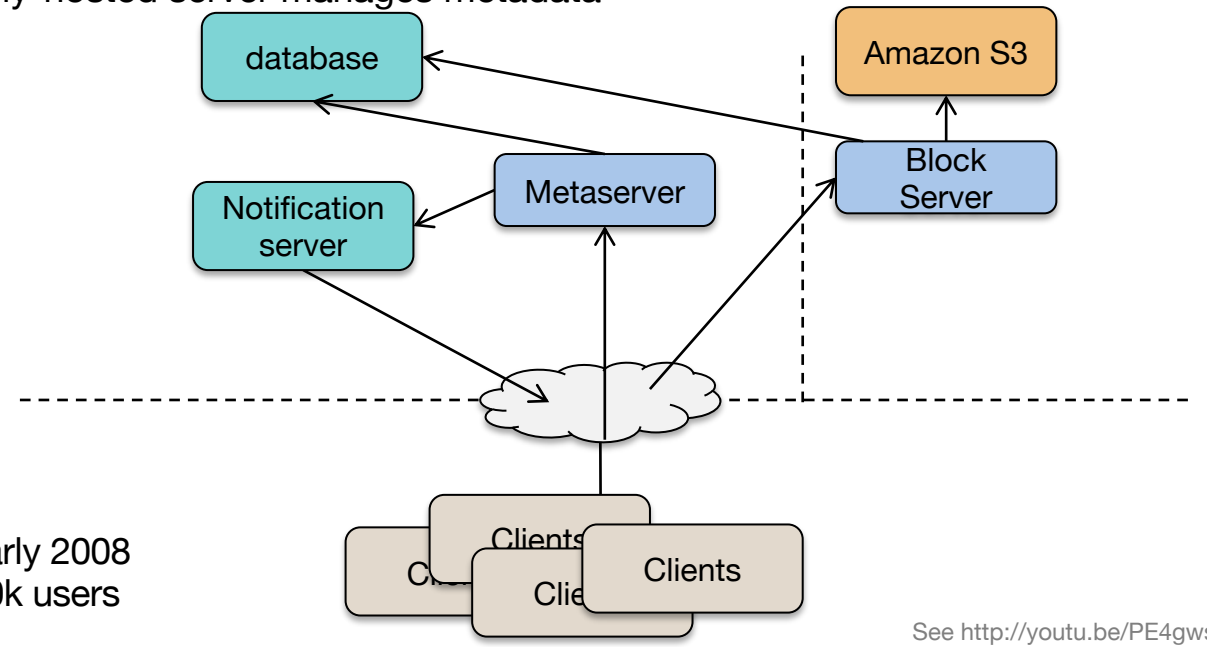
- Server ran out of disk space:
  moved data to Amazon S3 service (key-value store)
- Servers became overloaded: moved mySQL DB to another machine
- Clients periodically polled server for changes



See http://youtu.be/PE4gwstWhmc

Dropbox: architecture evolution: version 3

- Clients polling for changes was a bottleneck: add a **notification server**
- Split web server into two:
  - Amazon-hosted server hosts file content and accepts uploads (stored as blocks)
  - Locally-hosted server manages metadata

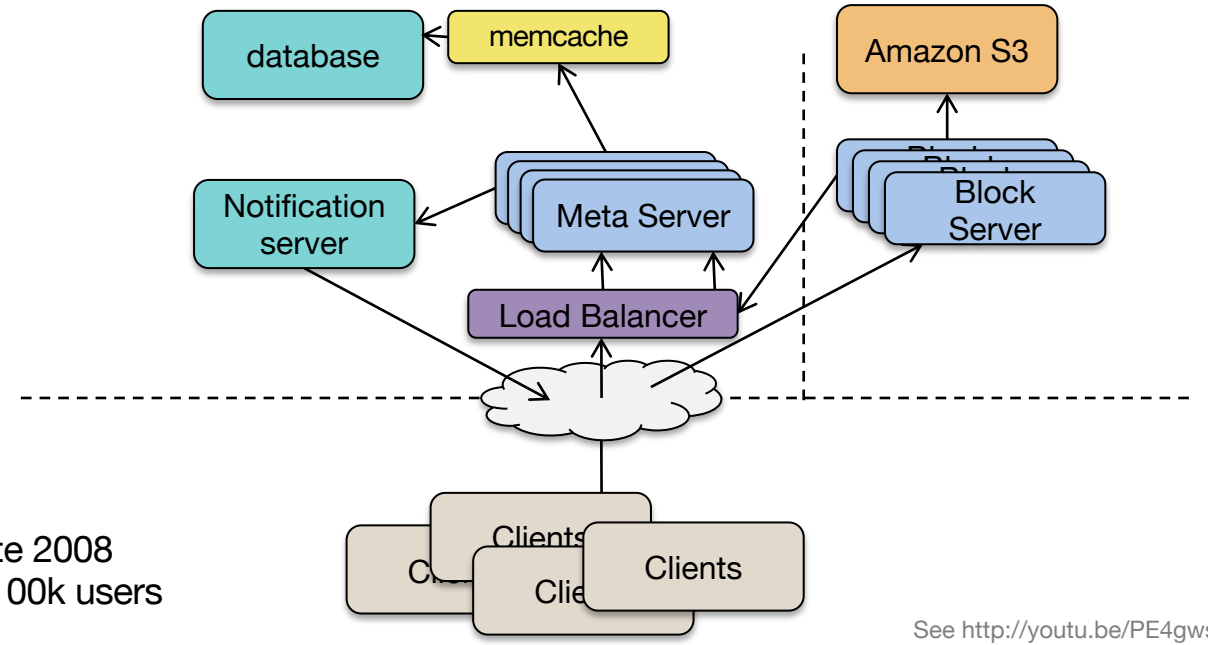


early 2008
50k users

See http://youtu.be/PE4gwstWhmc

Dropbox: architecture evolution: version 4

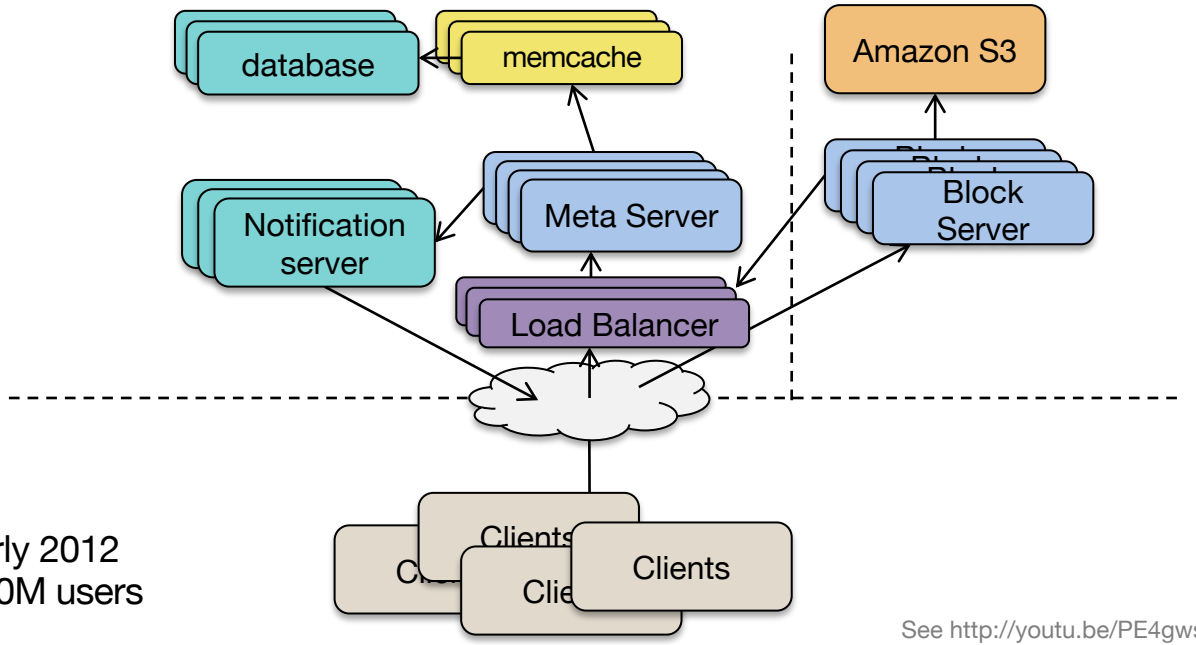
- Add more metaservers and blockservers
- Blockservers do not access DB directly; they send RPCs to metaservers
- Add a memory cache (memcache) in front of the database to avoid scaling



See http://youtu.be/PE4gwstWhmc

Dropbox: Architecture Evolution: version 5

- 10s of millions of clients – Clients have to connect before getting notifications
- Add 2-level hierarchy to notification servers: ~1 million connections/server



early 2012
>50M users

See http://youtu.be/PE4gwstWhmc

Continued evolution

- This covered the formative years of Dropbox
- The service evolved and continues to evolve
- Lots of features were added:
  - On-demand file downloads
    - Metadata stored in the user's file system; files can be downloaded when accessed
  - Version history, file recovery (including after ransomware attacks)
  - Sharing permissions, team ownership
  - Password protection
  - Document signing

The End