Distributed Systems

15. Distributed File Systems

Paul Krzyzanowski

Rutgers University

Fall 2017

Google Chubby
(≈ Apache Zookeeper)

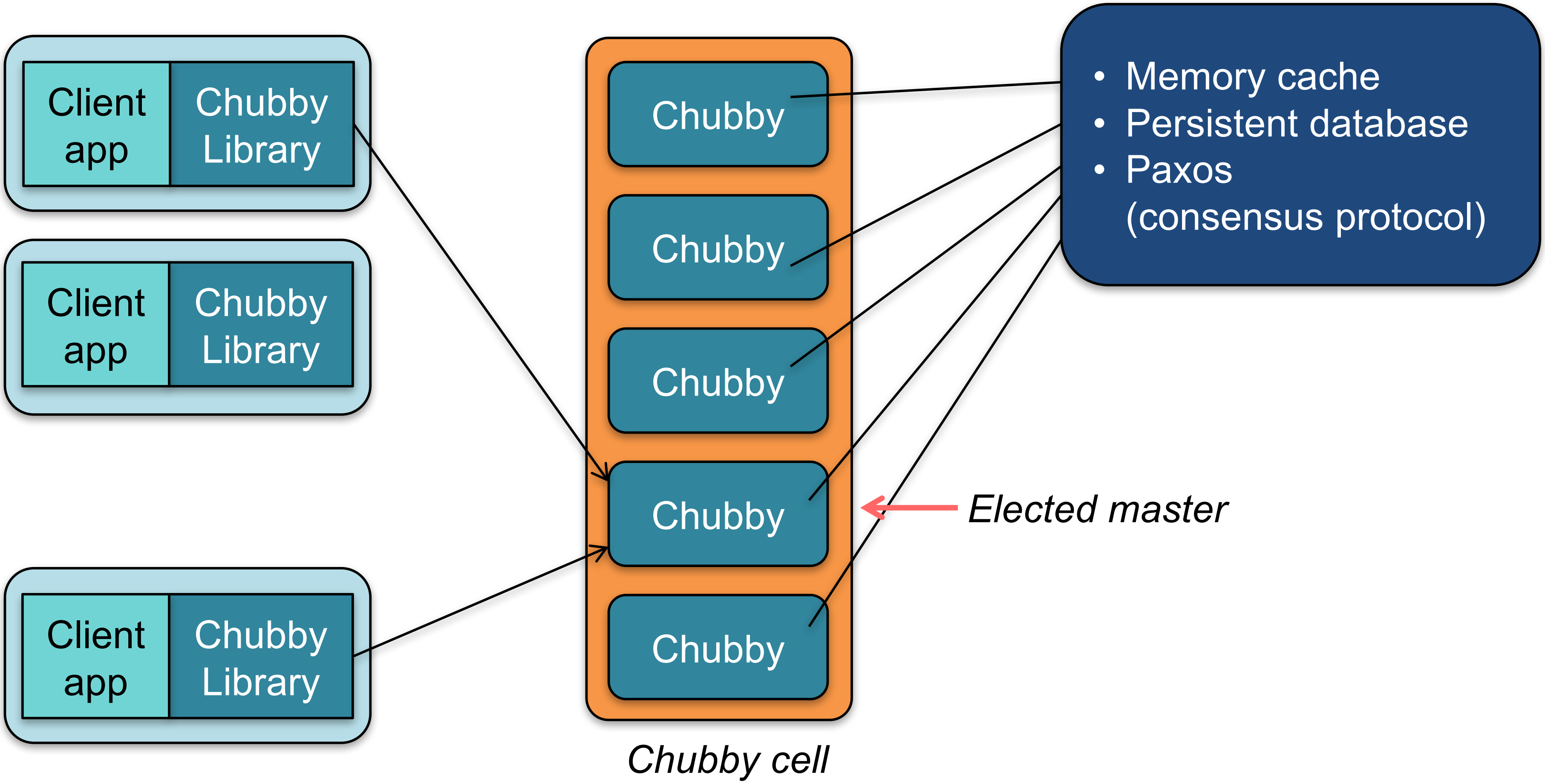
Chubby

Distributed lock service + simple fault-tolerant file system

- Interfaces
  - File access
  - Event notification
  - File locking
- Chubby is used to:
  - Manage coarse-grained, long-term locks (hours or days, not < sec)
    - get/release/check lock – identified with a name
  - Store small amounts of data associated with a name
    - E.g., system configuration info, identification of primary coordinators
  - Elect masters
- Design priority: availability rather than performance

Chubby Deployment

- Client library + a Chubby cell (5 replica servers)



- Memory cache
- Persistent database
- Paxos (consensus protocol)

Chubby Master

- Chubby has at most one master
  - All requests from the client go to the master

- All other nodes (replicas) must agree on who the master is
  - Paxos consensus protocol used to elect a master
  - Master gets a lease time
    - Re-run master selection after lease time expires to extend the lease

      …or if the master fails
  - When a Chubby node receives a proposal for a new master

    It will accept it *only* if the old master's lease expired

Simple User-level API for Chubby

- User-level RPC interface
  - Not implemented under VFS
  - Programs must access chubby via an API

- Look up Chubby nodes via DNS
- Ask any Chubby node for the master node
- File system interface (names, content, and locks)

Chubby: File System Interface

- /ls/cell/rest/of/name
  - /ls: lock service (common to all Chubby names)
  - cell: resolved to a set of servers in a Chubby cell via DNS lookup
  - /rest/of/name: interpreted within the cell

naming looks
sort of like AFS

- Each file has
  - Name
  - Data
  - Access control list
  - Lock
  - No modification, access times
  - No seek or partial reads/writes; no symbolic links; no moves

Chubby: API

open()	Set mode: read, write & lock, change ACL, event list, lock-delay, create
close()	
GetContentsAndStat()	Read file contents & metadata
SetContents(), SetACL()	Write file contents or ACL
Delete()	
Acquire(), TryAcquire(), Release()	Lock operations
GetSequencer()	Sequence # for a lock
SetSequencer()	Associate a sequencer with a file handle
CheckSequencer()	Check if sequencer is valid

Chubby: Locks

- Every file & directory can act as a reader-writer lock
  - Either one client can hold an exclusive (writer) lock
  - Or multiple clients can hold reader locks
- Locks are advisory
- If a client releases a lock, the lock is immediately available
- If a client fails, the lock will be unavailable for a *lock-delay* period (typically 1 minute)

Using Locks for Leader Election

- Using Chubby locks makes leader election easy
  - No need for user servers to participate in a consensus protocol
    … the programmer doesn't need to figure out Paxos (or Raft)
  - Chubby provides the fault tolerance
  - Participant tries to acquire a lock
    - If it gets it, then it’s the master for whatever service it's providing!
- Example: electing a master & using it to write to a file server
  - Participant gets a lock, becomes master *(for its service, not Chubby)*
    - Gets a lock sequence count
  - In each RPC to a server, send the sequence count to the server
  - During request processing, a server will reject old (delayed) packets

```
if (sequence_count < current_sequence_count)
    reject request  /* it must be from a delayed packet */
```

Events

Clients may subscribe to events:

- File content modifications
- Child node added/removed/modified
- Chubby master failed over
- File handle & its lock became invalid
- Lock acquired
- Conflicting lock request from another client

Chubby client caching & master replication

- At the client
  - Data cached in memory by chubby clients
    - Cache is maintained by a Chubby lease, which can be invalidated
  - All clients write through to the Chubby master
- At the master
  - Writes are propagated via Paxos consensus to all Chubby replicas
    - Data updated in total order – replicas remain synchronized
    - The master replies to a client *after* the writes reach a majority of replicas
  - Cache invalidations
    - Master keeps a list of what each client may be caching
    - Invalidations sent by master and are acknowledged by client
    - File is then cacheable again
  - Chubby database is backed up to GFS every few hours

Parallel File Systems

Client-server file systems

- Central servers
  - Point of congestion, single point of failure
- Alleviate somewhat with replication and client caching
  - E.g., Coda, oplocks
  - Limited replication can lead to congestion
  - Separate set of machines to administer
- 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

Google File System (GFS)

(≈ Apache Hadoop Distributed File System)

GFS Goals

- Scalable distributed file system
- Designed for large data-intensive applications
- Fault-tolerant; runs on commodity hardware
- Delivers high performance to a large number of clients

Design Assumptions

- Assumptions for conventional file systems don't work
  - E.g., "*most files are small*", "*lots have short lifetimes*"
- Component failures are the norm, not an exception
  - File system = thousands of storage machines
  - Some % not working at any given time
- Files are huge. Multi-TB files are the norm
  - It doesn't make sense to work with billions of *n*KB-sized files
  - I/O operations and block size choices are also affected

Design Assumptions

- 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, small random reads – *these dominate*
    - Large appends
    - Hundreds of processes may append to a file concurrently
- FS will store a modest number of files for its scale
  - approx. a few million
- Designing the FS API with the design of apps benefits the system
    - Apps can handle a relaxed consistency model

Basic Design Idea

- "Normal" 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
    - Lists of block numbers containing file data are allocated from the same set of data blocks used for file data
- Parallel file systems: separate data and metadata
  - Metadata = information about the file
    - Includes name, access permissions, timestamps, size, location of data blocks
  - Data = actual file contents

Basic Design Idea

- Use separate servers to store metadata
  - Metadata includes lists of (*server, block_number*) sets that 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 often 4KB
  - Expect huge files, so use huge blocks
    - List of blocks that makes up a file becomes easier to manage
- Replicate data
  - Expect some servers to be down
  - Store data on multiple servers

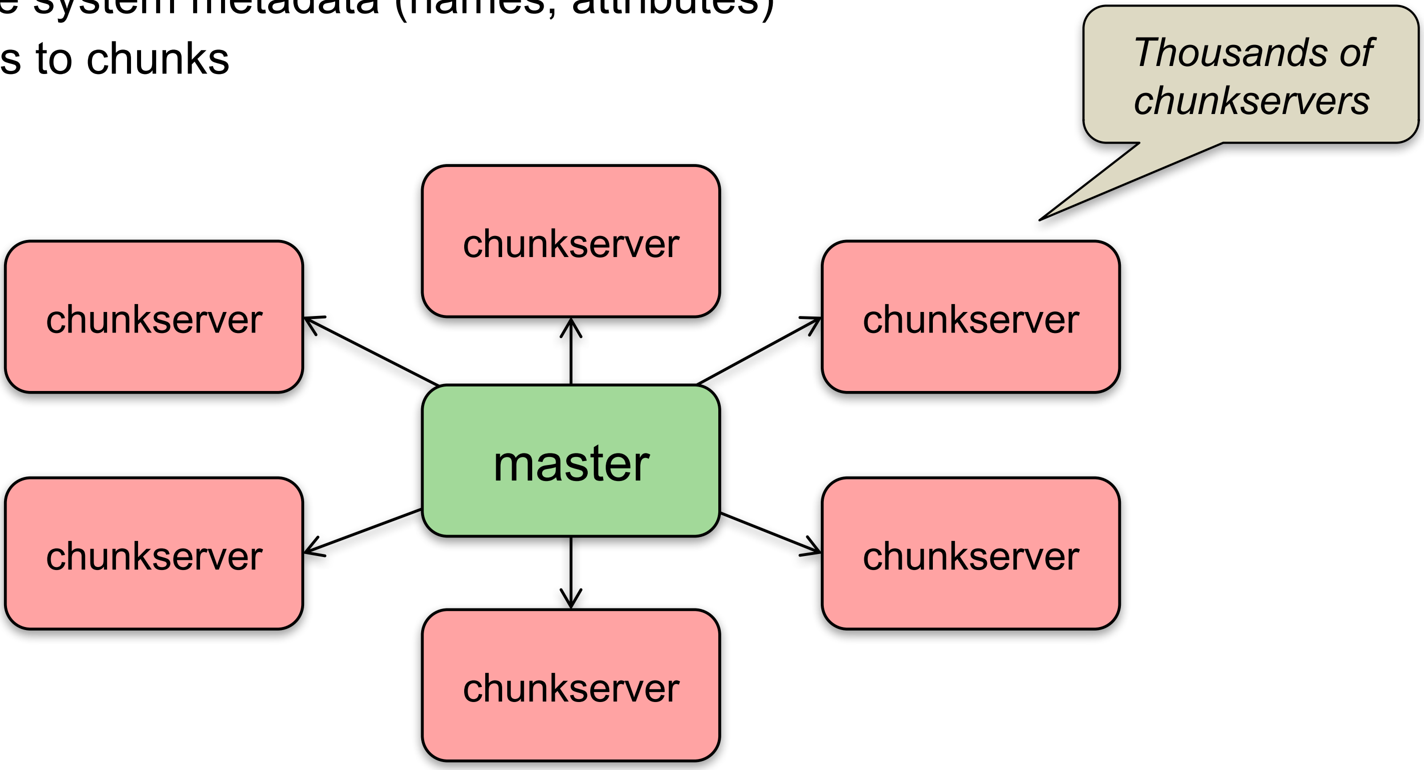
File System Interface

- GFS does *not* have a standard OS-level API
  - No POSIX 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

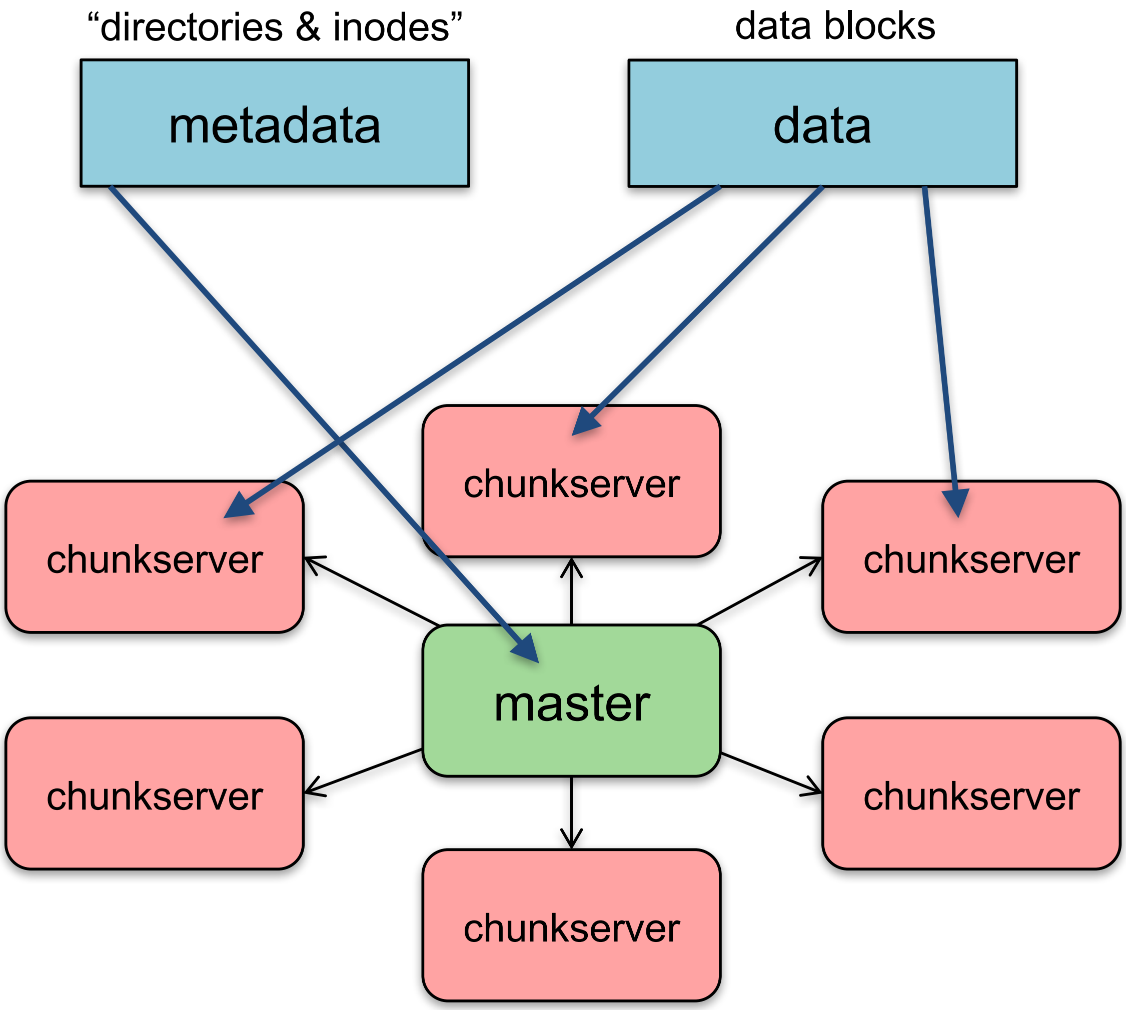
GFS cluster

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

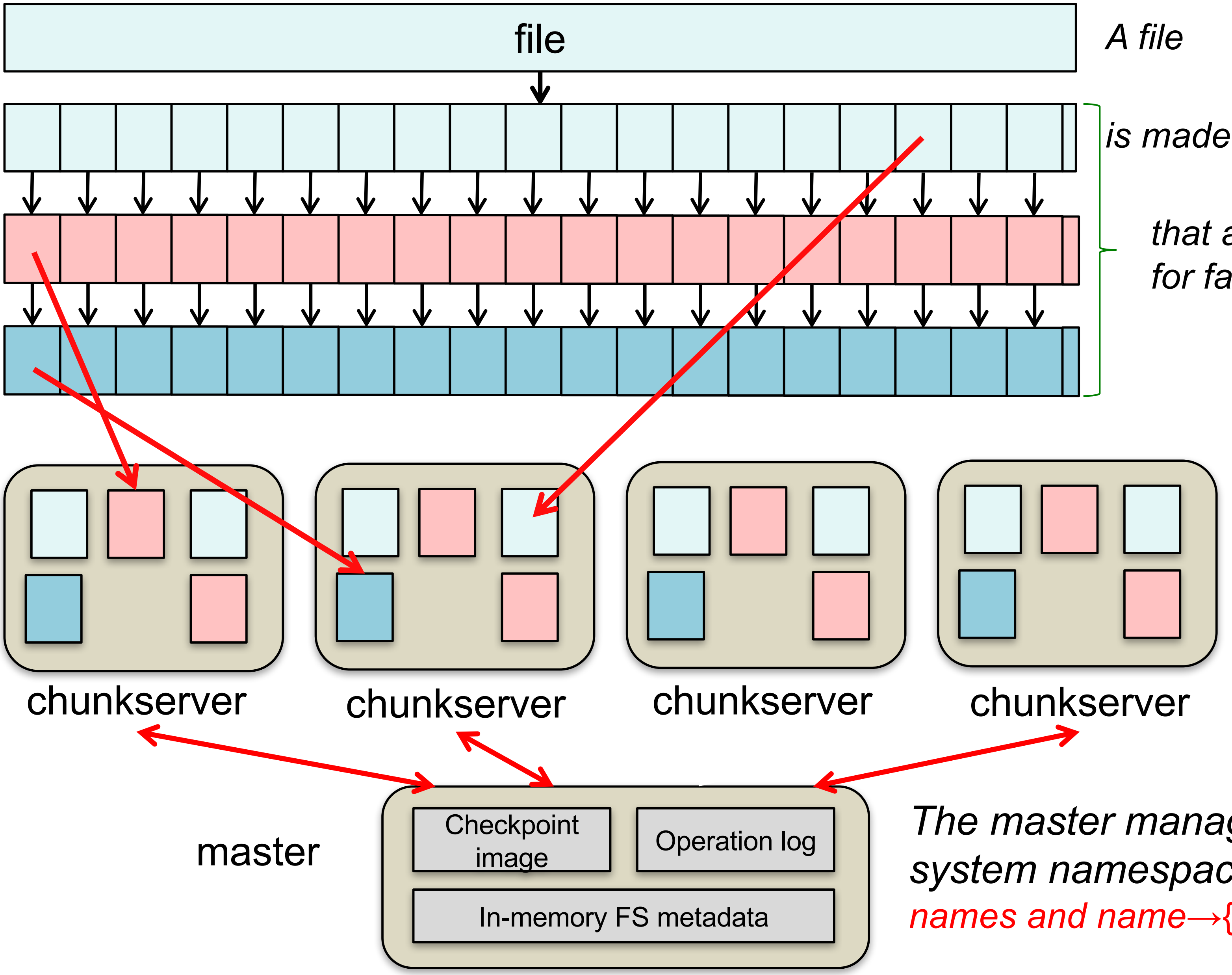


GFS Master & Chunkservers

GFS cluster



GFS Files



A file

is made of 64 MB chunks

that are replicated for fault tolerance

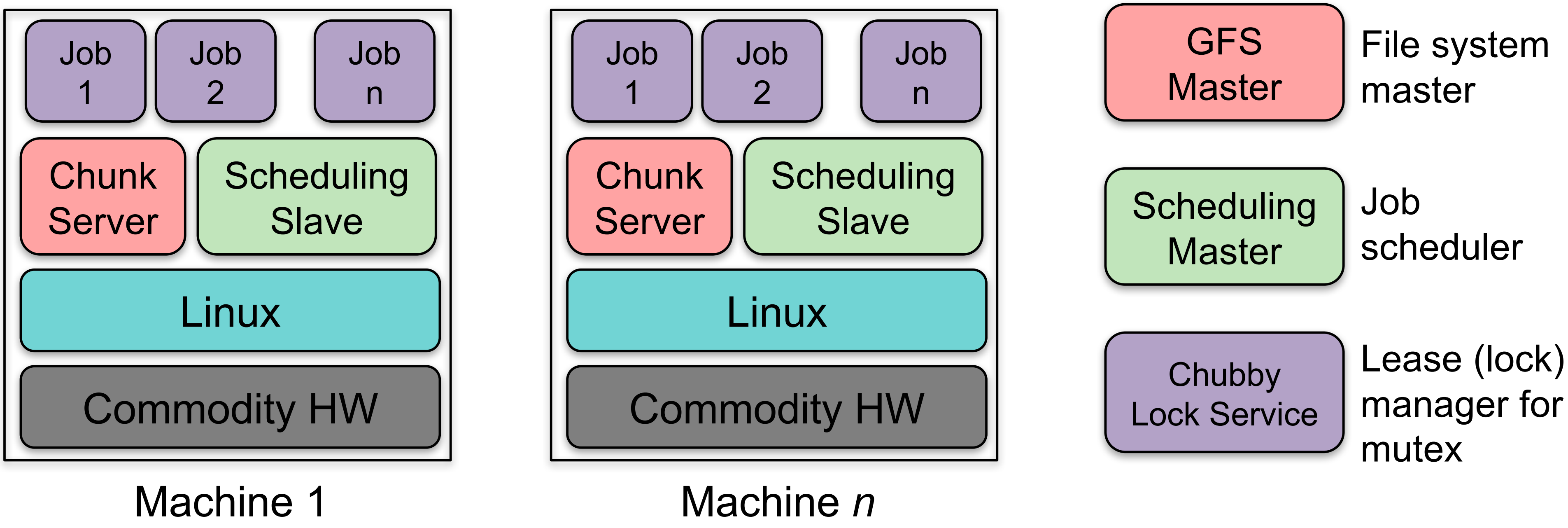
Chunks live on chunkservers

The master manages the file system namespace:
names and name→{chunk list}

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 filesystems, 40 GB/s read/write loads

Bring the computation close to the data



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

- Maintains all file system metadata
  - Namespace
  - Access control info
  - Filename to chunks mappings
  - Current locations of chunks
- Manages
  - Chunk leases (locks)
  - Garbage collection (freeing unused chunks)
  - Chunk migration (copying/moving chunks)
- Master replicates its data for fault tolerance
- Periodically communicates with all chunkservers
  - Via heartbeat messages
  - To get state and send commands

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

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

Why Large Chunks?

- Default chunk size = 64MB
  (compare to Linux ext4 block sizes: typically 4 KB and up to 1 MB)

- Reduces need for frequent communication with master to get chunk location info

- Clients can easily cache info to refer to all data of large files
  - Cached data has timeouts to reduce possibility of reading stale data

- Large chunk makes it feasible to keep a TCP connection open to a chunkserver for an extended time

- Master stores <64 bytes of metadata for each 64MB chunk

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

4. Contact any available chunkserver for chunk data

Writing to files

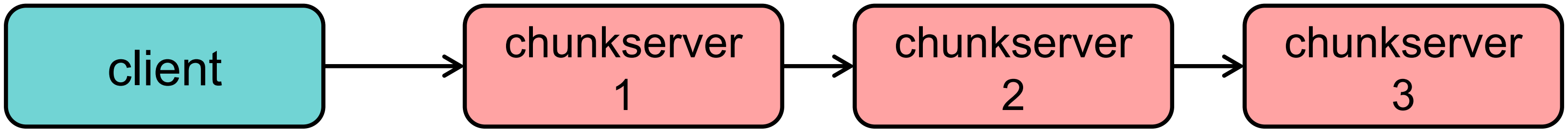
- Less frequent than reading
- Master grants a **chunk lease** to one of the replicas
  - This replica will be the **primary replica** chunkserver
  - Primary can request lease extensions, if needed
  - Master increases the chunk version number and informs replicas

Writing to files: two phases

Phase 1: Send data

Deliver data but don’t write to the file

- A client is given a list of replicas
  - Identifying the primary and secondaries
- Client writes to the closest replica chunkserver
  - Replica forwards the data to another replica chunkserver
  - That chunkserver forwards to another replica chunkserver
- Chunkservers store this data in a cache

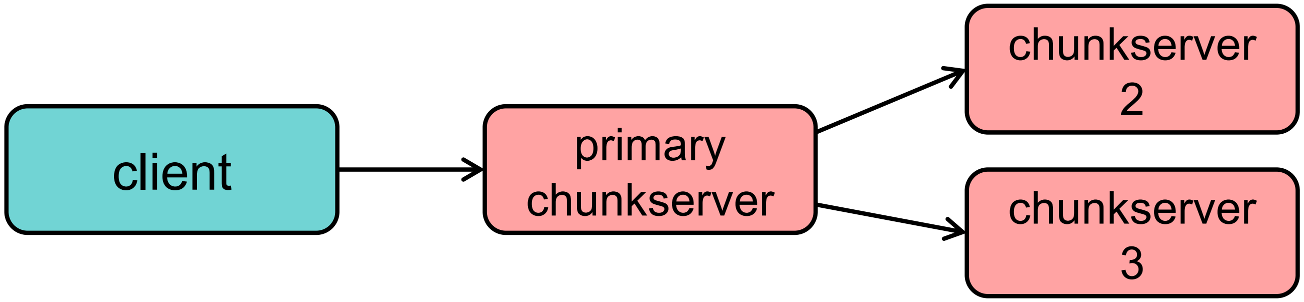


Writing to files: two phases

Phase 2: Write data

Add it to the file (commit)

- Client waits for replicas to acknowledge receiving the data
- Send 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 order to secondaries
- Once all acknowledgements have been received, the primary acknowledges the client



Writing to files

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)
- Namespace is a single lookup table
  - Maps pathnames to metadata

HDFS: Hadoop Distributed File System

- Primary storage system for Hadoop applications

- Hadoop
  - Software library – framework that allows for the distributed processing of large data sets across clusters of computers

- Hadoop includes:
  - MapReduce™: software framework for distributed processing of large data sets on compute clusters.
  - Avro™: A data serialization system.
  - Cassandra™: A scalable multi-master database with no single points of failure.
  - Chukwa™: A data collection system for managing large distributed systems.
  - HBase™: A scalable, distributed database that supports structured data storage for large tables.
  - Hive™: A data warehouse infrastructure that provides data summarization and ad hoc querying.
  - Mahout™: A Scalable machine learning and data mining library.
  - Pig™: A high-level data-flow language and execution framework for parallel computation.
  - ZooKeeper™: A high-performance coordination service for distributed applications.

HDFS Design Goals & Assumptions

- HDFS is an open source (Apache) implementation inspired by GFS design
- Similar goals and same basic design as GFS
  - Run on commodity hardware
  - Highly fault tolerant
  - High throughput – Designed for large data sets
  - OK to relax some POSIX requirements
  - Large scale deployments
    - Instance of HDFS may comprise 1000s of servers
    - Each server stores part of the file system's data
- But
  - No support for concurrent appends

HDFS Design Goals & Assumptions

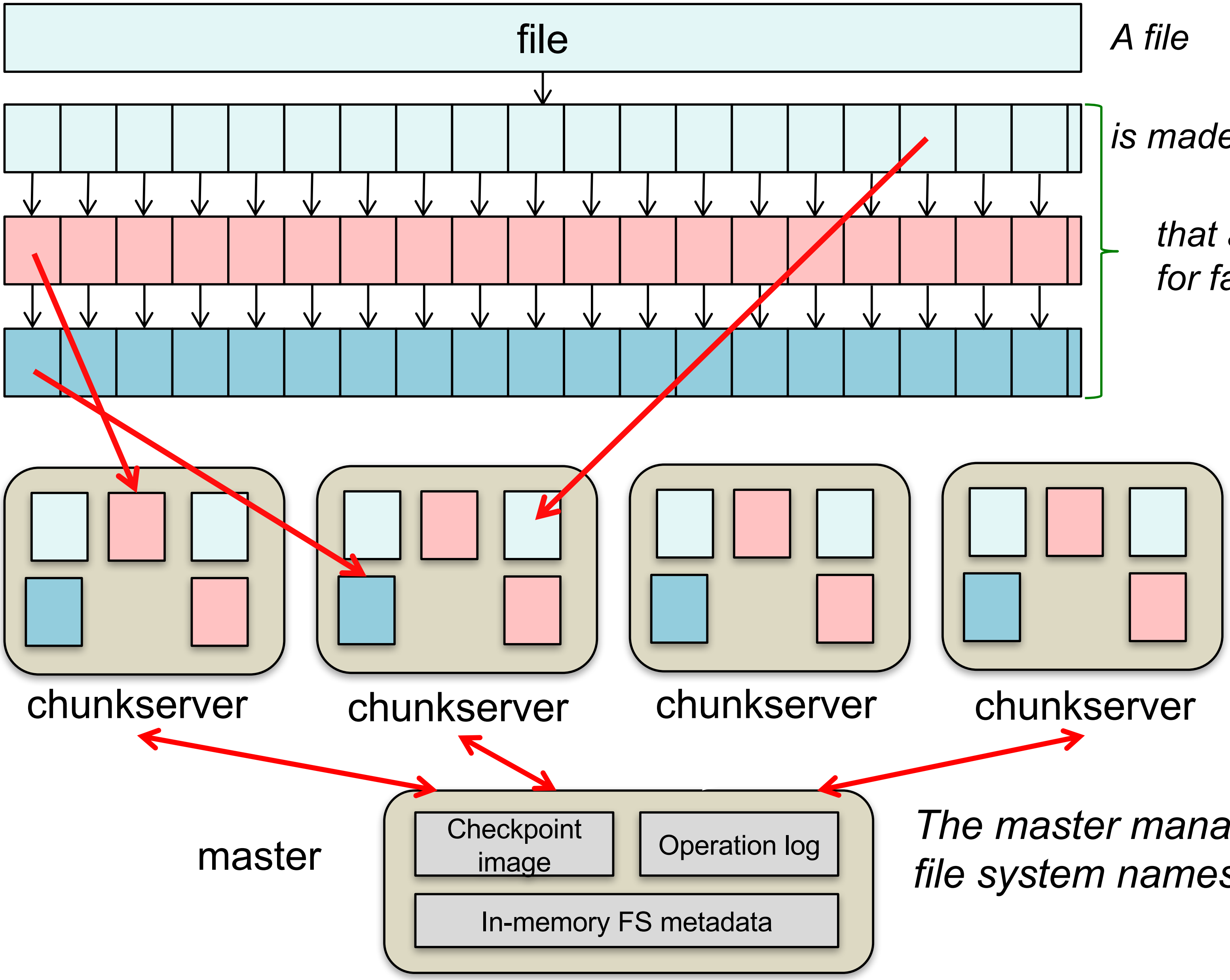
- Write-once, read-many file access model

- A file's contents will not change
  - Simplifies data coherency
  - Suitable for web crawlers and MapReduce applications

HDFS Architecture

- Written in Java
- Master/Slave architecture
- Single NameNode
  - Master server responsible for the namespace & access control
- Multiple DataNodes
  - Responsible for managing storage attached to its node
- A file is split into one or more blocks
  - Typical block size = 128 MB (vs. 64 MB for GFS)
  - Blocks are stored in a set of DataNodes

GFS



A file

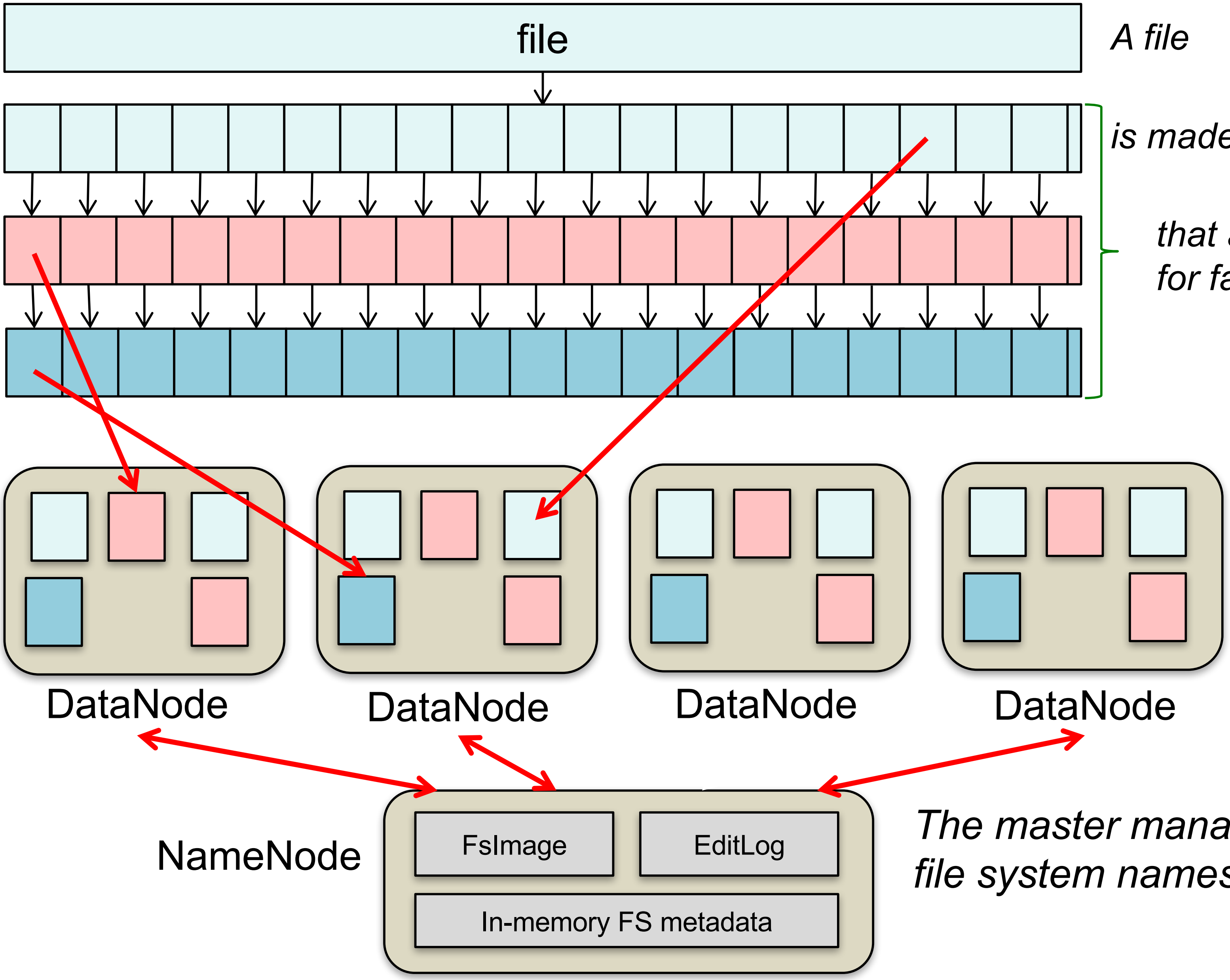
is made of 64 MB chunks

that are replicated for fault tolerance

Chunks live on chunkservers

The master manages the file system namespace

HDFS: same stuff … different names



A file

is made of 128 MB blocks

that are replicated for fault tolerance

Blocks live on DataNodes

The master manages the file system namespace

NameNode (= GFS master)

- Executes metadata operations
  - *open, close, rename*
  - Maps file blocks to DataNodes
  - Maintains HDFS namespace

- Transaction log (EditLog) records every change that occurs to file system metadata
  - Entire file system namespace + file-block mappings is stored in memory
  - … and stored in a file (FsImage) for persistence

- NameNode receives a periodic *Heartbeat* and *Blockreport* from each DataNode
  - Heartbeat = “I am alive” message
  - Blockreport = list of all blocks on a datanode
    - Keep track of which DataNodes own which blocks & replication count

DataNode (= GFS chunkserver)

- Responsible for serving read/write requests
- Blocks are replicated for fault tolerance
  - App can specify # replicas at creation time
  - Can be changed later
- Blocks are stored in the local file system at the DataNode

Rack-Aware Reads & Replica Selection

- Client sends request to NameNode
  - Receives list of blocks and replica DataNodes per block

- Client tries to read from the closest replica
  - Prefer same rack
  - Else same data center
  - Location awareness is configured by the admin

Writes

- Client caches file data into a temp file
- When temp file ≥ one HDFS block size
  - Client contacts NameNode
  - NameNode inserts file name into file system hierarchy & allocates a data block
  - Responds to client with the destination data block
  - Client writes to the block at the corresponding DataNode
- When a file is closed, remaining data is transferred to a DataNode
  - NameNode is informed that the file is closed
  - NameNode commits file creation operation into a persistent store (log)
- Data writes are chained: pipelined
  - Client writes to the first (closest) DataNode
  - That DataNode writes the data stream to the second DataNode
  - And so on…

Internet-based file sync & sharing: 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 possibility of low bandwidth to user
  - Scalable back-end architecture
  - 99%+ of code written in Python

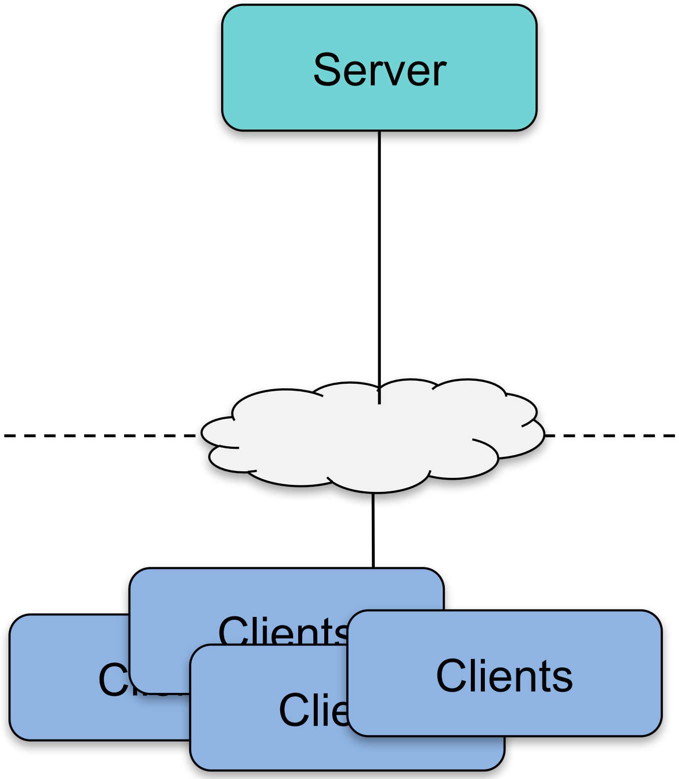
    ⇒server software migrated to Go in 2013

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
  - File upload : file download ratio roughly 1:1
    - Huge number of uploads compared to traditional services
- Must abide by most ACID 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
  - (OK to punt on “Isolated”)

Dropbox: architecture evolution: version 1

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

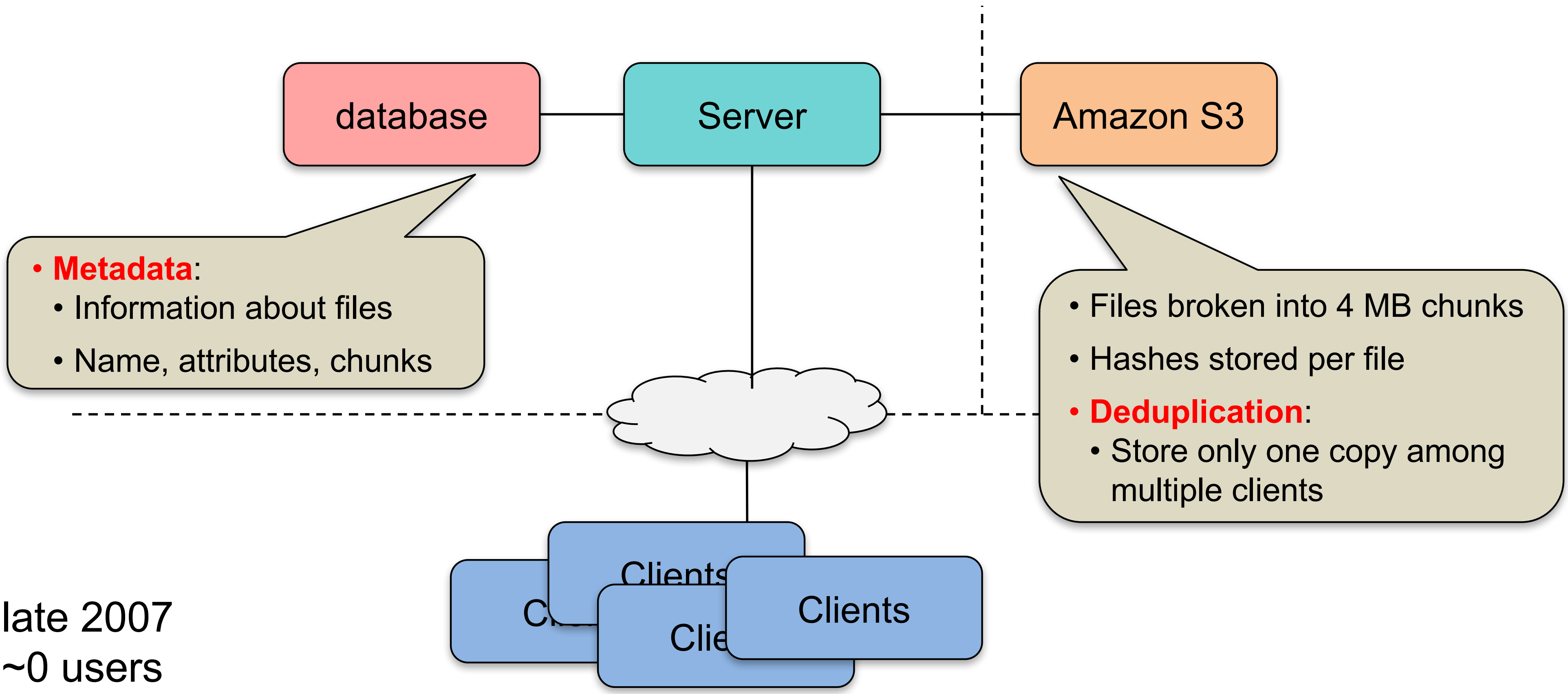


mid 2007
0 users

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 <u>polled</u> server for changes



- **Metadata**:
  - Information about files
  - Name, attributes, chunks

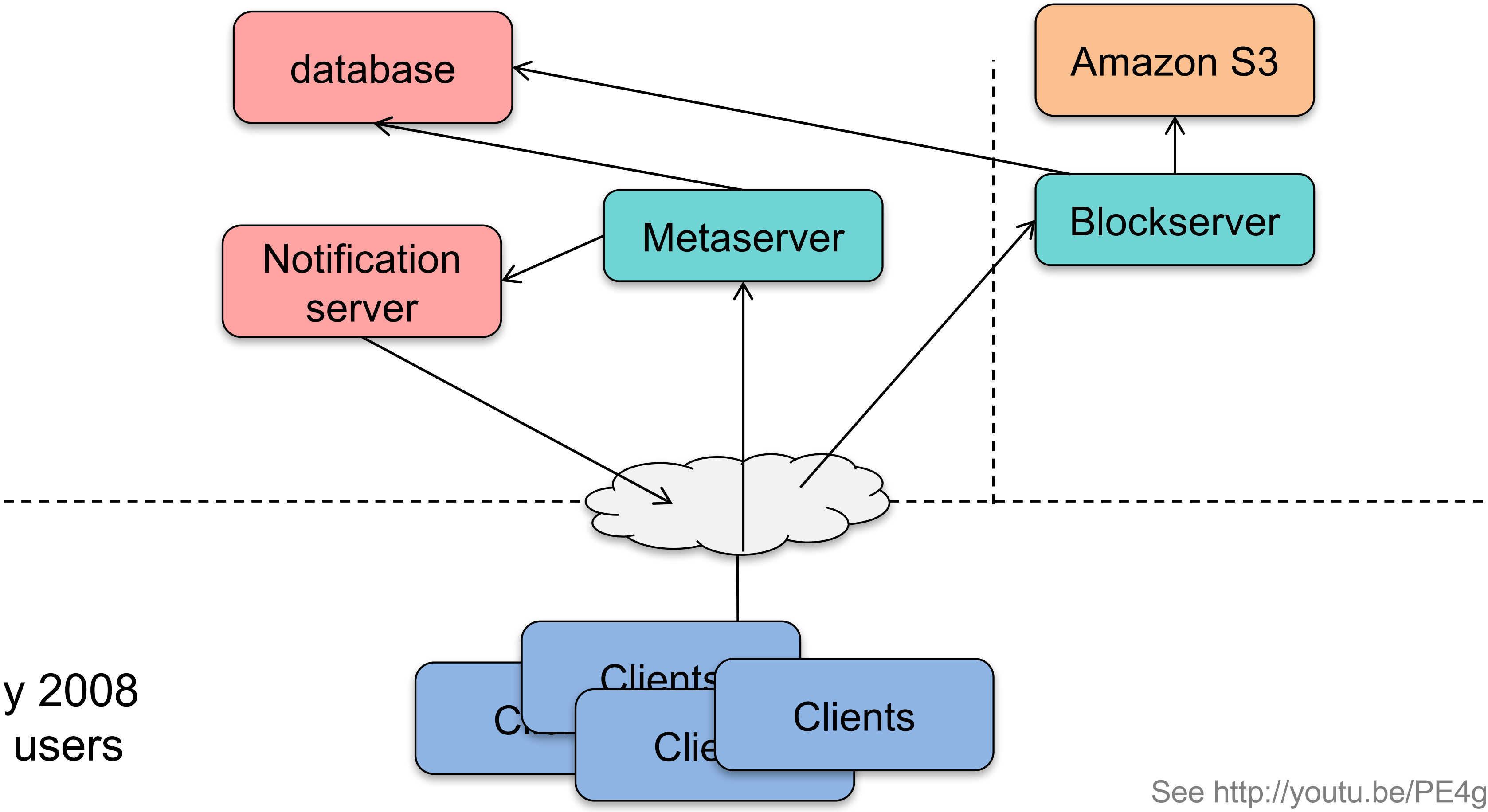
- Files broken into 4 MB chunks
- Hashes stored per file
- **Deduplication**:
  - Store only one copy among multiple clients

late 2007
~0 users

See http://youtu.be/PE4gwstWhmc

Dropbox: architecture evolution: version 3

- Move from polling to notifications: add **notification server**
- Split web server into two:
  - Amazon-hosted server hosts file content and accepts uploads (stored as blocks)
  - Locally-hosted server manages metadata

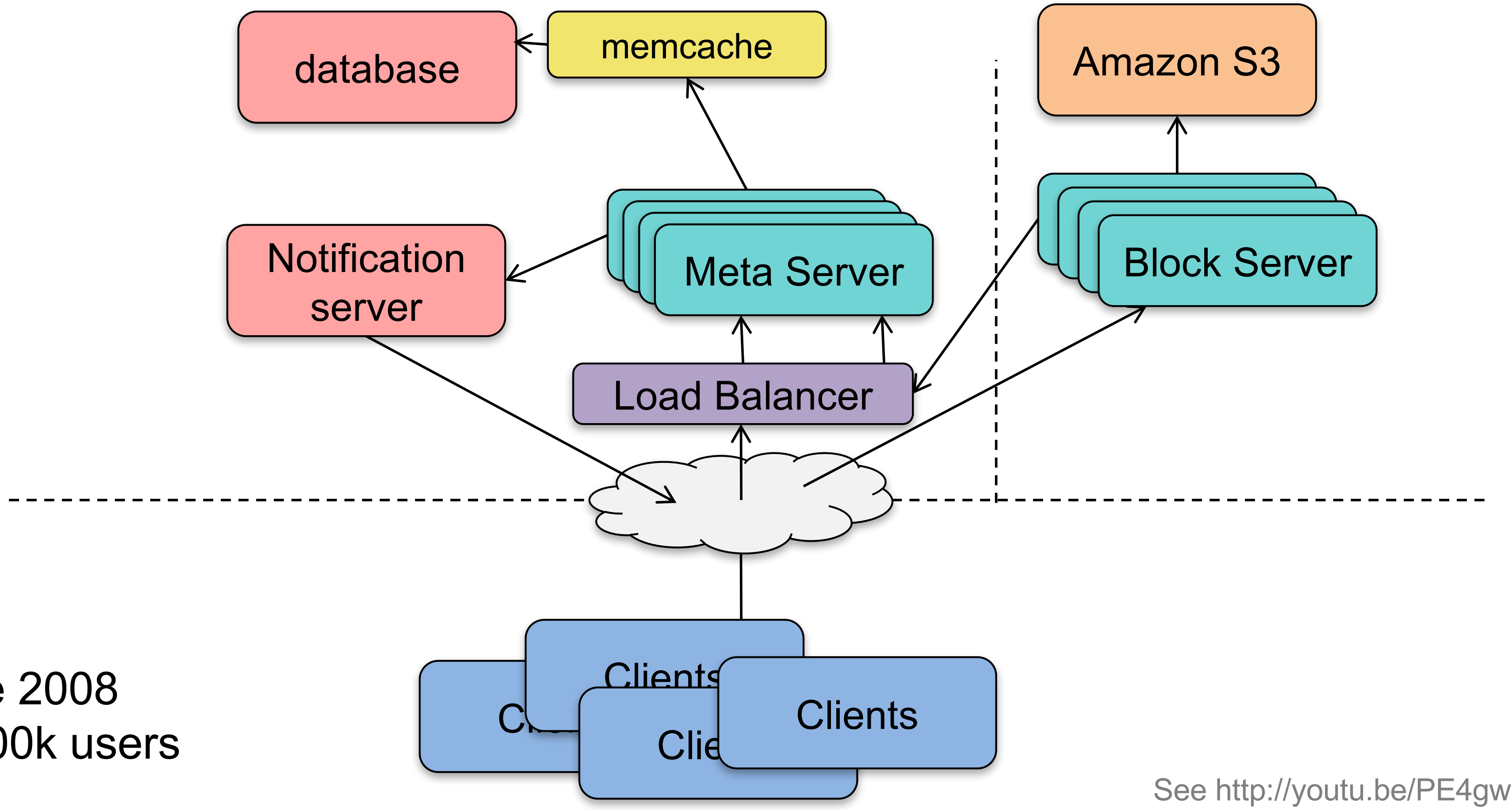


early 2008
50k users

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

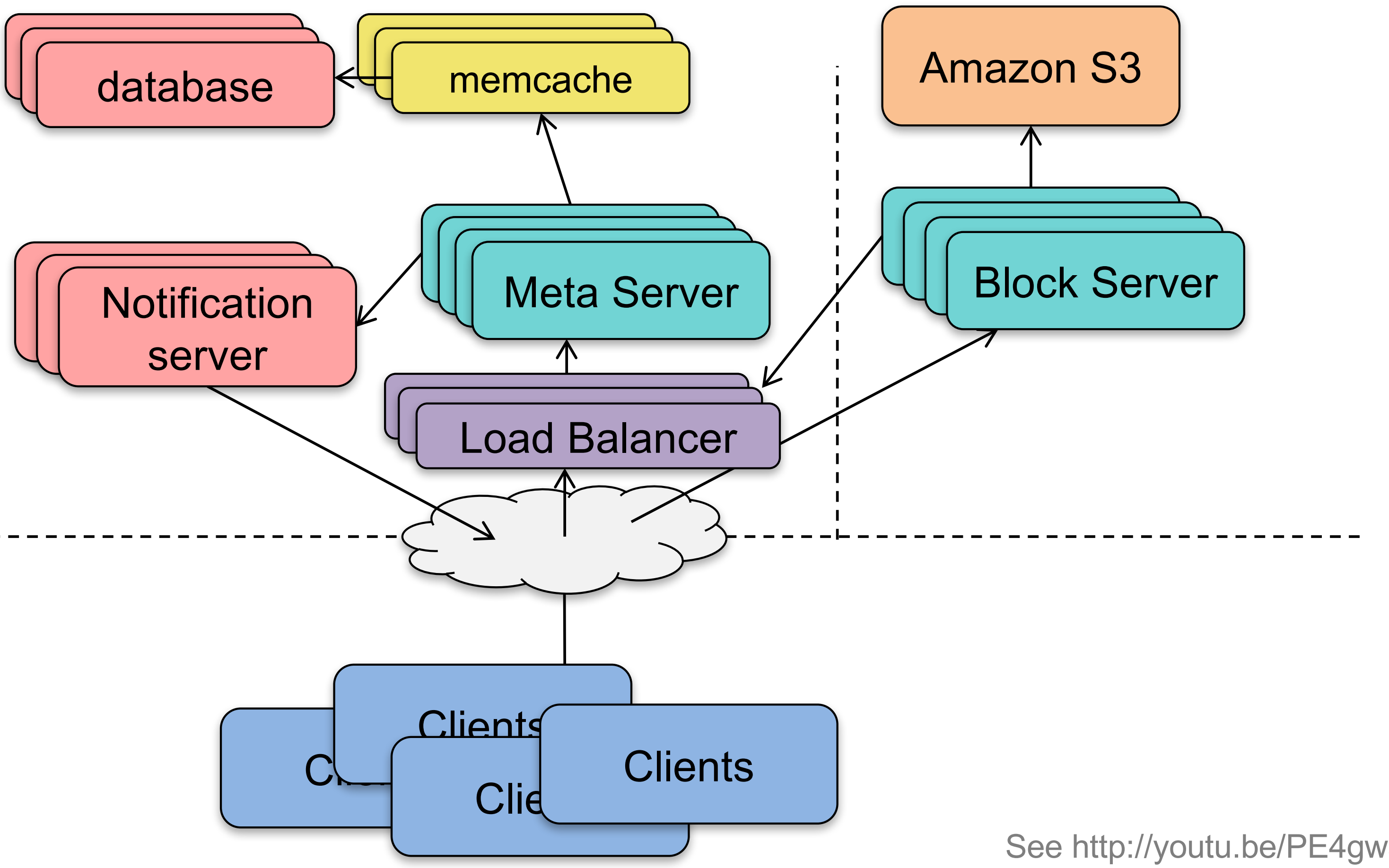


late 2008
~100k users

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

The End