A Review of
The Google File System

Taylor Autry and Eric Addison

Distributed Systems

UT Austin, Fall 2017

Overview

- What is a file system?
- What is a distributed file system?
- The Google File System (GFS)
  - Design Goals
  - Architecture & Implementation
- Operations
  - Write
  - Record Append
  - Snapshot
- Fault Tolerance and Diagnosis
- Current State of GFS

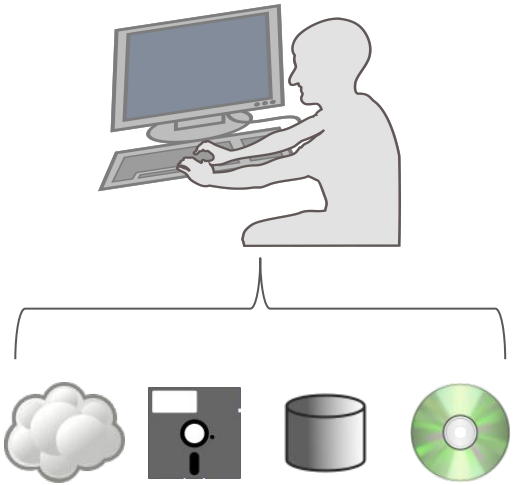
What is a (computer) File System?

A means of organizing and accessing files on storage

Logical Layer

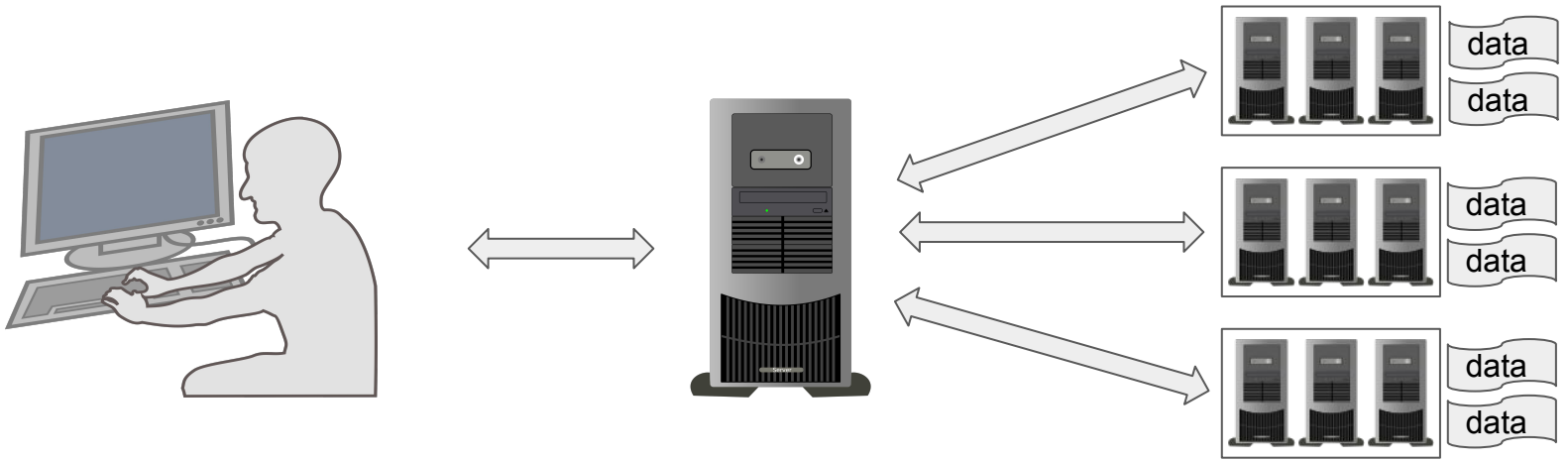
Virtual Layer

Physical Layer



What is a *Distributed* File System?

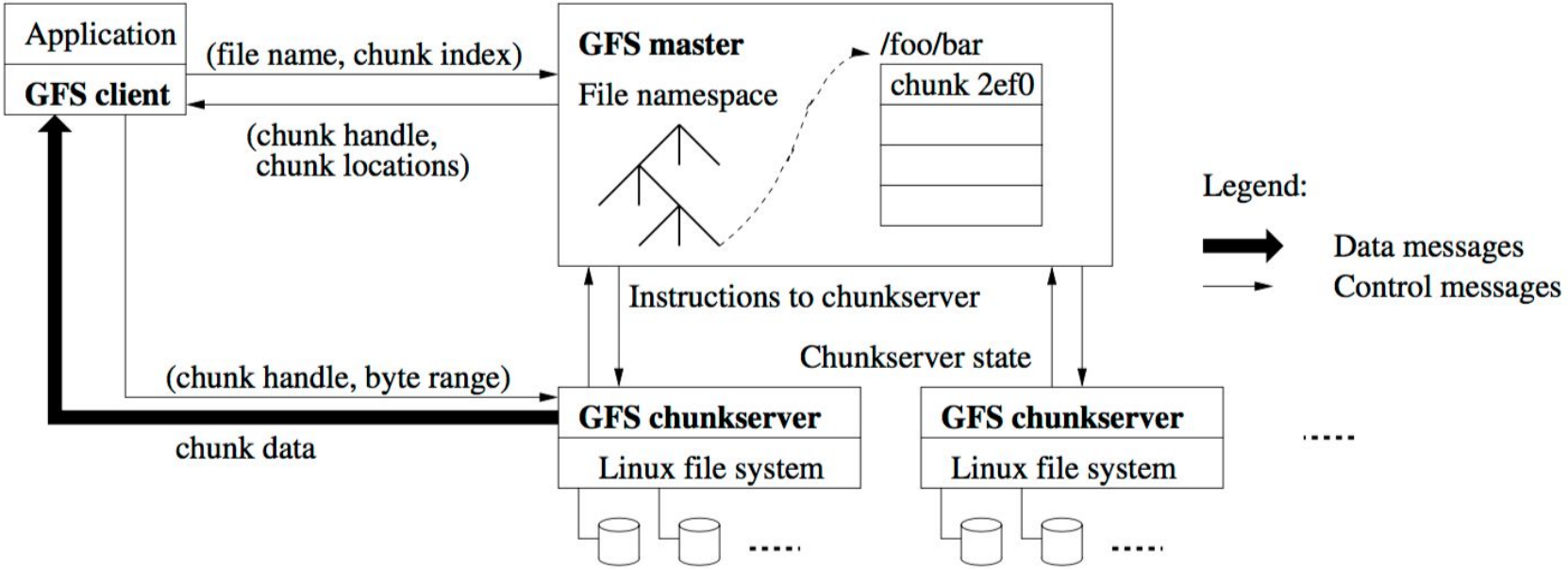
A File System that is simultaneously mounted on multiple servers that share data access via network protocols



The Google File System (GFS) - Design Goals

- Specifically designed to handle Google's application workload
- Shared goals - availability, scalability, performance, reliability
- Designers made several key assumptions
  - Hardware failures are the norm, not the exception
  - The files are huge
  - Writes are mostly appends
  - Must support multiple clients appending to same file
  - Reads are mostly sequential
  - Bandwidth is more important than latency

The Google File System (GFS) - Architecture



The Google File System. Sanjay Ghemawat, Howard Gobioff, Shun-Tak Leung, Proceedings of the 19th ACM Symposium on Operating Systems Principles, 2003, Figure 1

The Google File System (GFS) - Architecture

- Chunk size is 64MB
  - Reduces load on master
  - Reduces network traffic
  - Reduces storage requirements for metadata
  - Disadvantage - small files could become hotspots

- Master Metadata
  - File and chunk namespace
  - File to chunk mapping → Written to operation log
  - Location of replicas -- Not persisted locally

GFS Operations

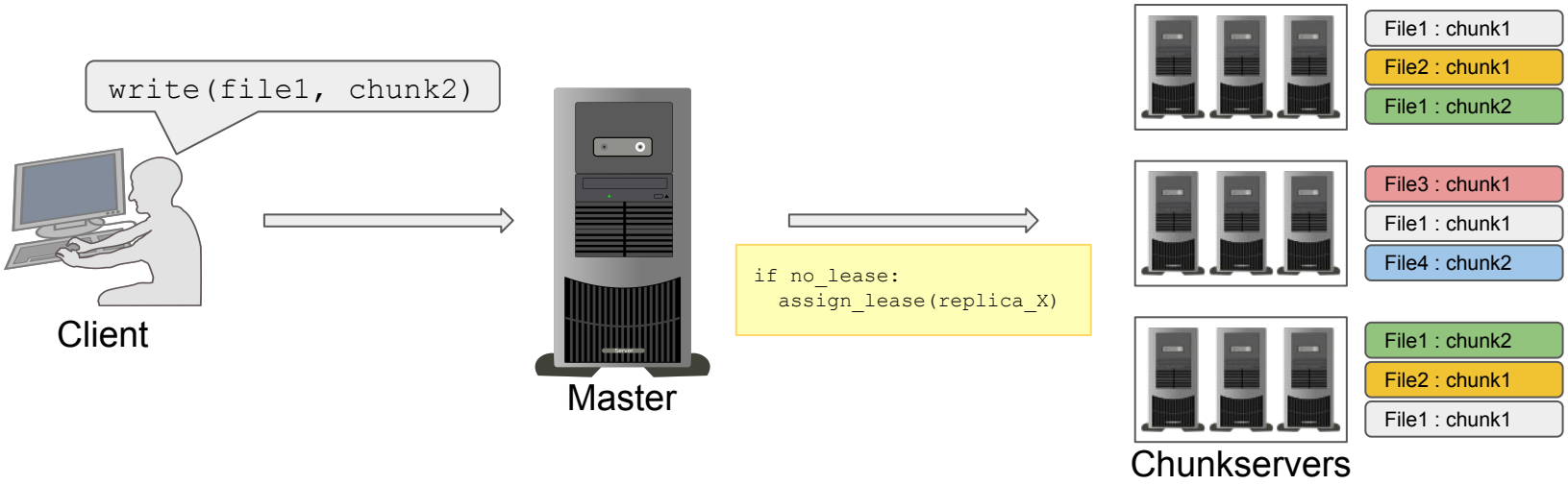
- GFS “provides a familiar file system interface”
  - Does not implement a standard like POSIX, though
- Typical file access operations are supported
  - e.g. `open, close, read, write`

- Snapshot
  - Low-cost copy operation
- Record Append
  - Concurrent atomic file appending

Flow of a Mutation: the `Write` Operation

Step 1: Client asks master for current lease-holder and replica locations

If no current lease, master assigns lease to one of the replicas



Flow of a Mutation: the `Write` Operation

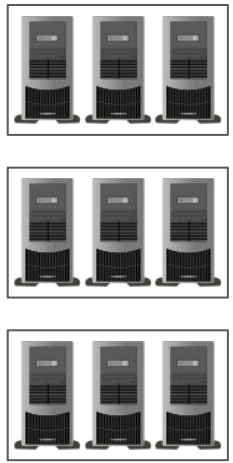
Step 2: Master responds with identity of primary chunk and location of secondaries

Client caches this data, no further communication with master (unless problems arise)

Client

```
primary:  192.124.52.51
Replica1: 192.124.52.52
Replica2: 192.201.42.12
...
```

Master

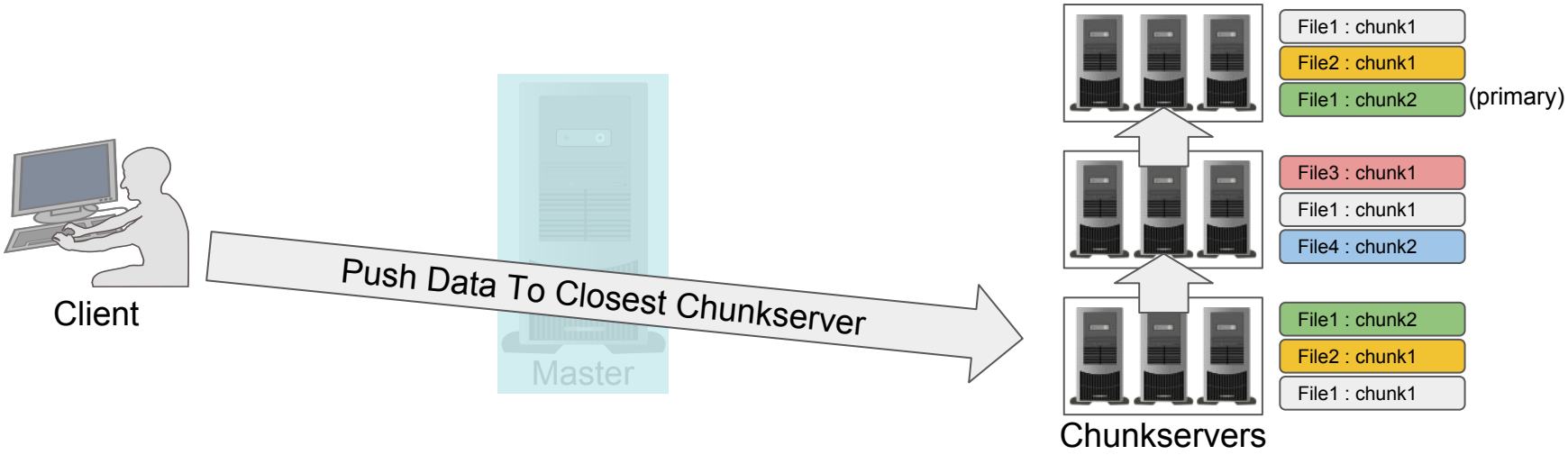


Chunkservers

Flow of a Mutation: the `Write` Operation

Step 3: Client pushes directly data to closest replica

Data is pushed along a linear chain of chunkservers, optimized by proximity

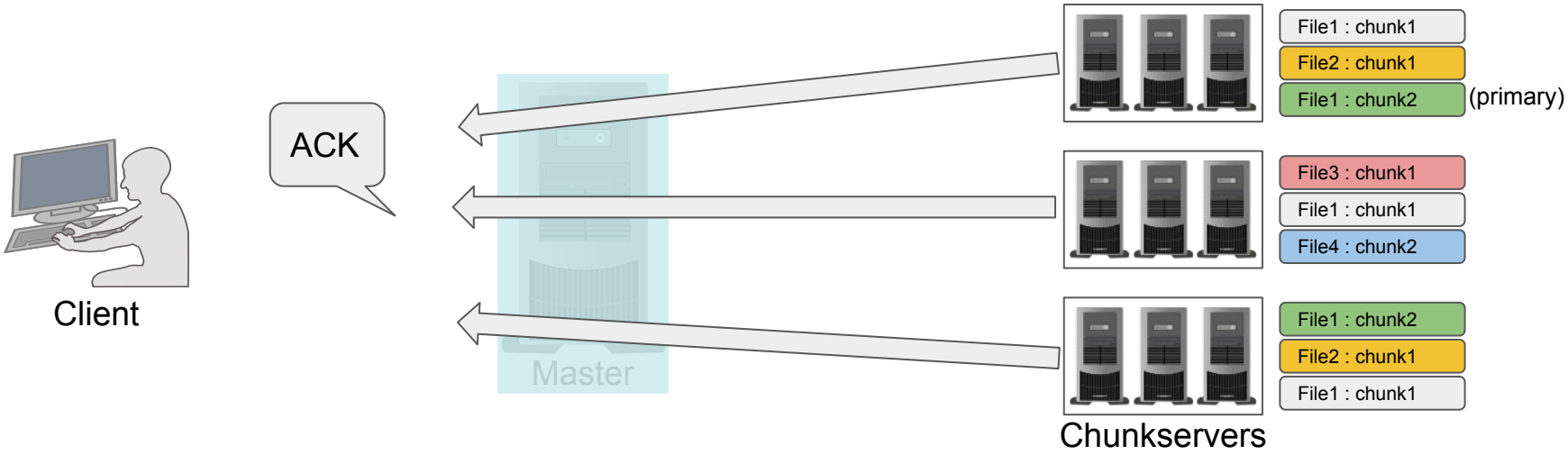


Flow of a Mutation: the `Write` Operation

Step 4: **Chunkservers ACK receipt of data**

Client sends write request to primary

Primary assigns serial #s to mutations and applies to its own state

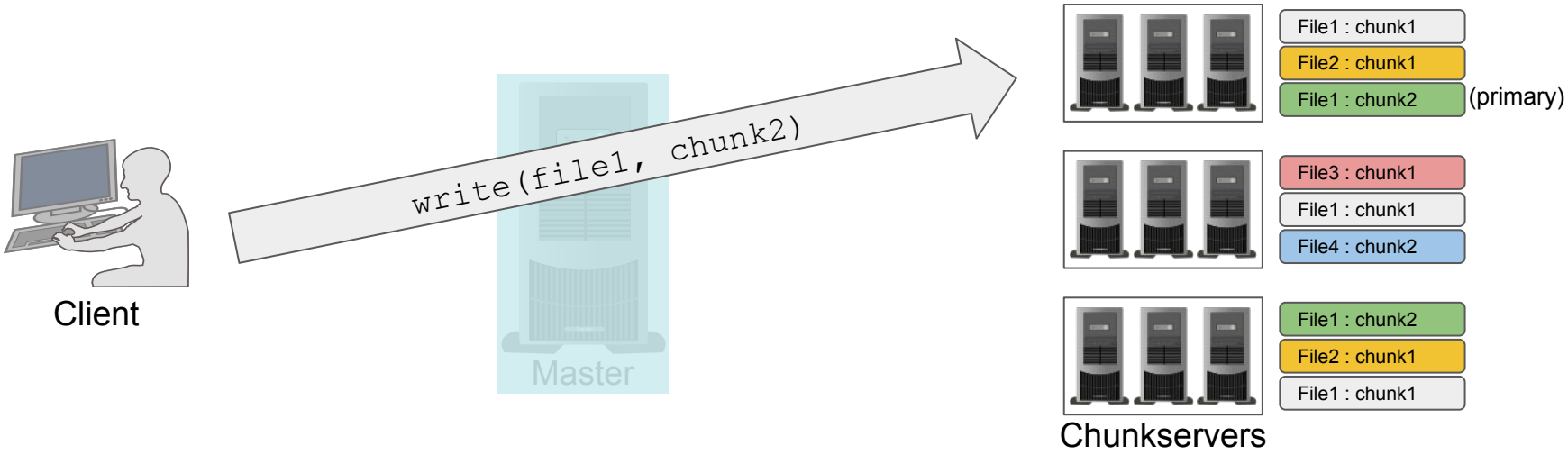


Flow of a Mutation: the `Write` Operation

Step 4: Chunkservers ACK receipt of data

Client sends write request to primary

Primary assigns serial #s to mutations and applies to its own state



Flow of a Mutation: the `Write` Operation

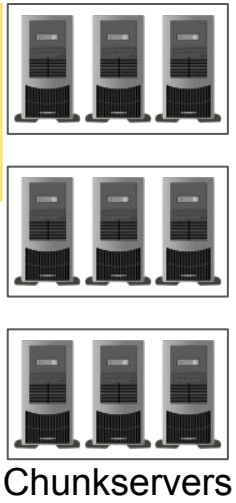
Step 4: Chunkservers ACK receipt of data

Client sends write request to primary

Primary assigns serial #s to mutations and applies to its own state

Client


```
mut1_serial =
10939581240
mut2_serial =
10939581241
apply_mutations()
...
```

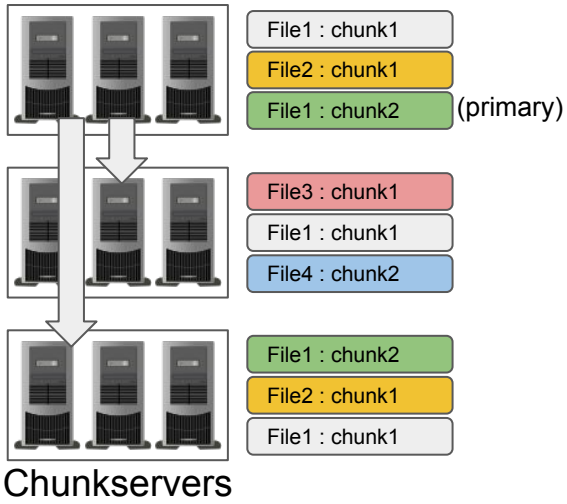


Flow of a Mutation: the `Write` Operation

Step 5: **Primary forwards `write` requests to secondary replicas**

Secondary replicas apply mutations ordered by serial #

Client

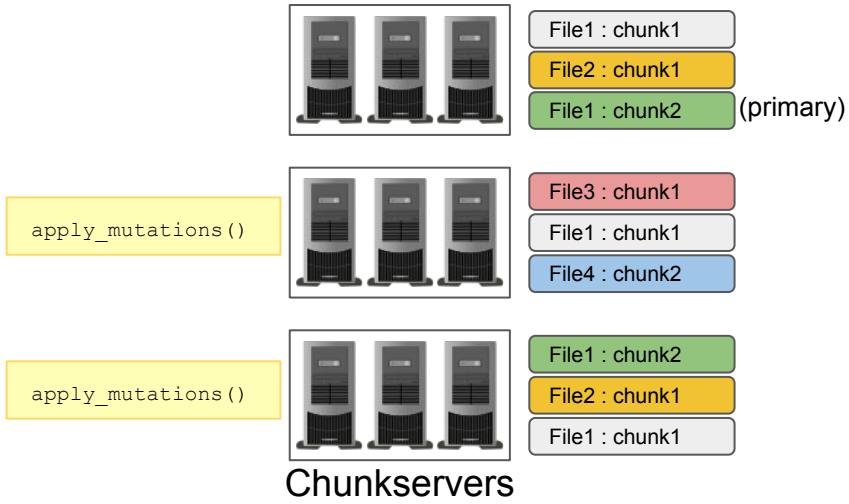


Flow of a Mutation: the `Write` Operation

Step 5: Primary forwards `write` requests to secondary replicas

Secondary replicas apply mutations ordered by serial #

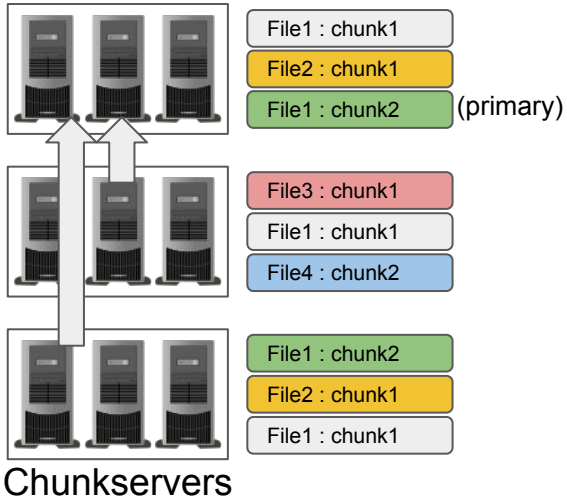
Client



Flow of a Mutation: the `Write` Operation

Step 6: Secondaries reply to primary indicating complete

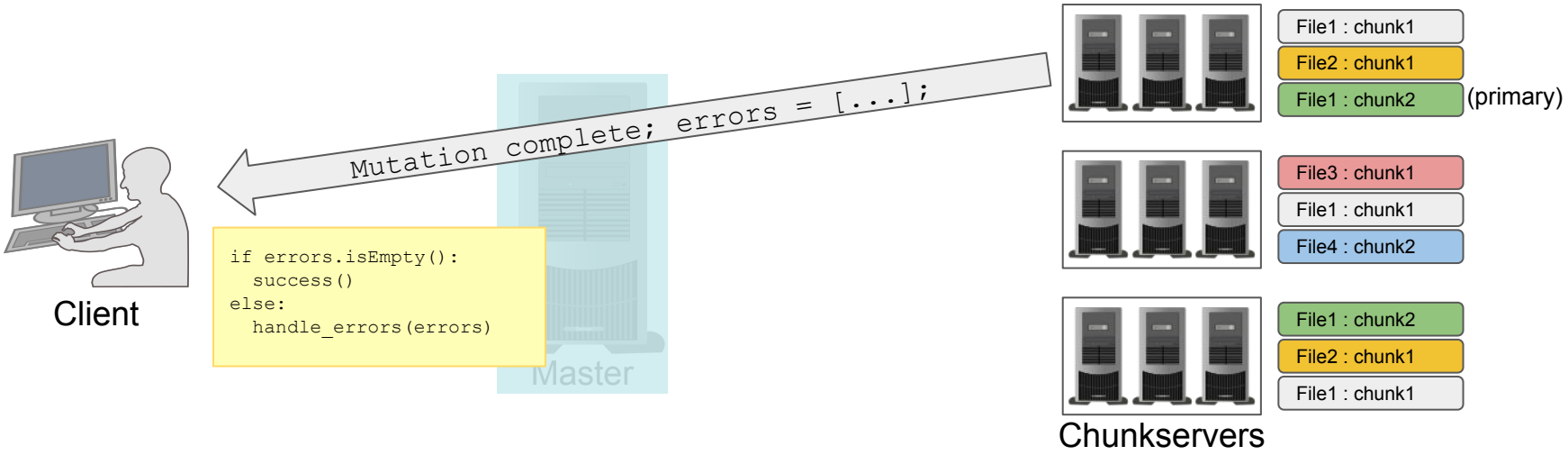
Client



Flow of a Mutation: the `Write` Operation

Step 7: Primary replies to client, reporting any errors

Client handles errors by (essentially) retrying failed mutation



The `Snapshot` Operation

- Make a very fast (almost instantaneous) copy of a file or directory
- Copy-on-write

Step 0: Client makes `copy` request

Client

Master

File1 : chunk1
File2 : chunk1
File1 : chunk2

File3 : chunk1
File1 : chunk1
File4 : chunk2

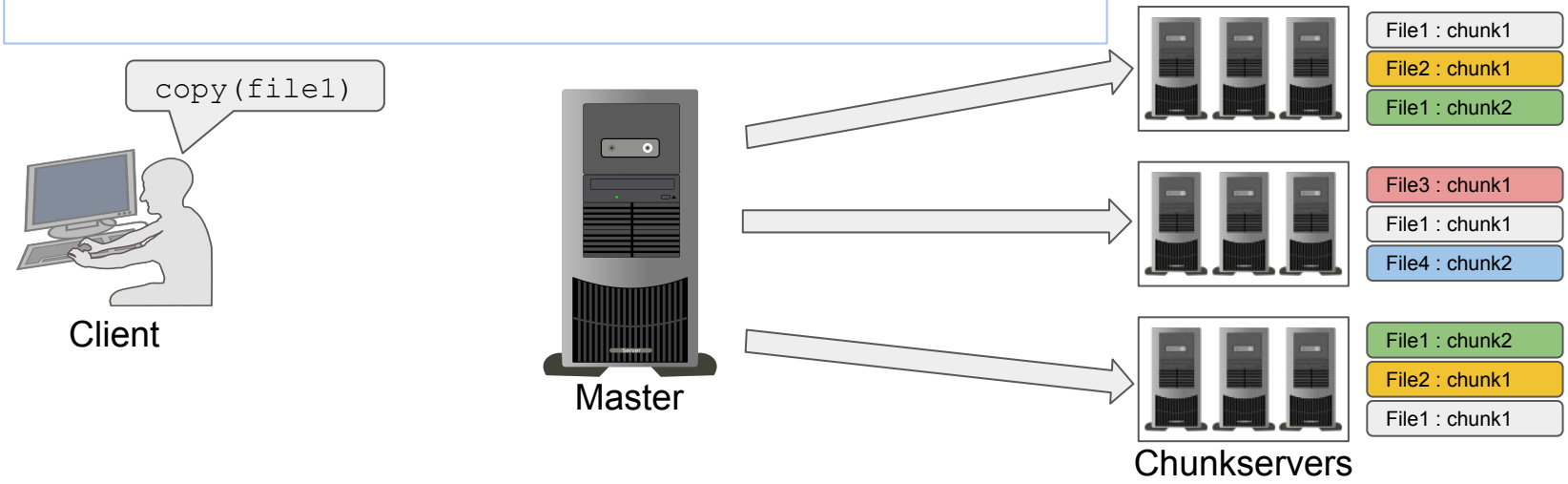
File1 : chunk2
File2 : chunk1
File1 : chunk1

Chunkservers

The `Snapshot` Operation

- Make a very fast (almost instantaneous) copy of a file or directory
- Copy-on-write

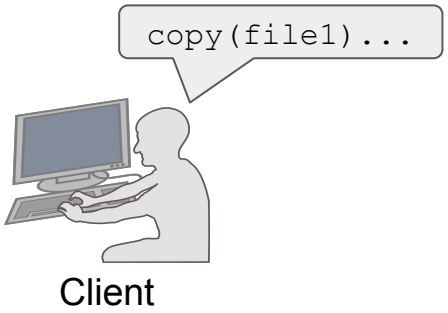
Step 1: Master revokes any outstanding leases on relevant chunks



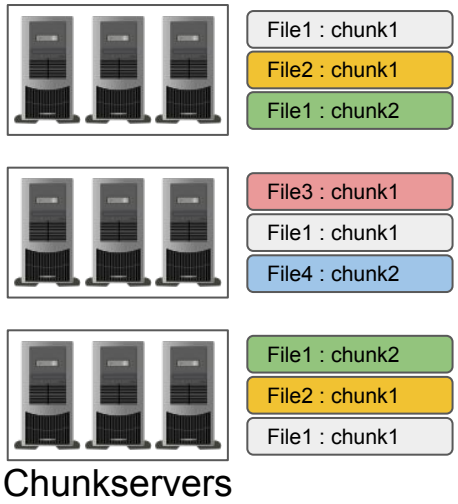
The `Snapshot` Operation

- Make a very fast (almost instantaneous) copy of a file or directory
- Copy-on-write

Step 2: Master logs the operation to disk




```
log("copy(file1)")
flush_log()
...
```

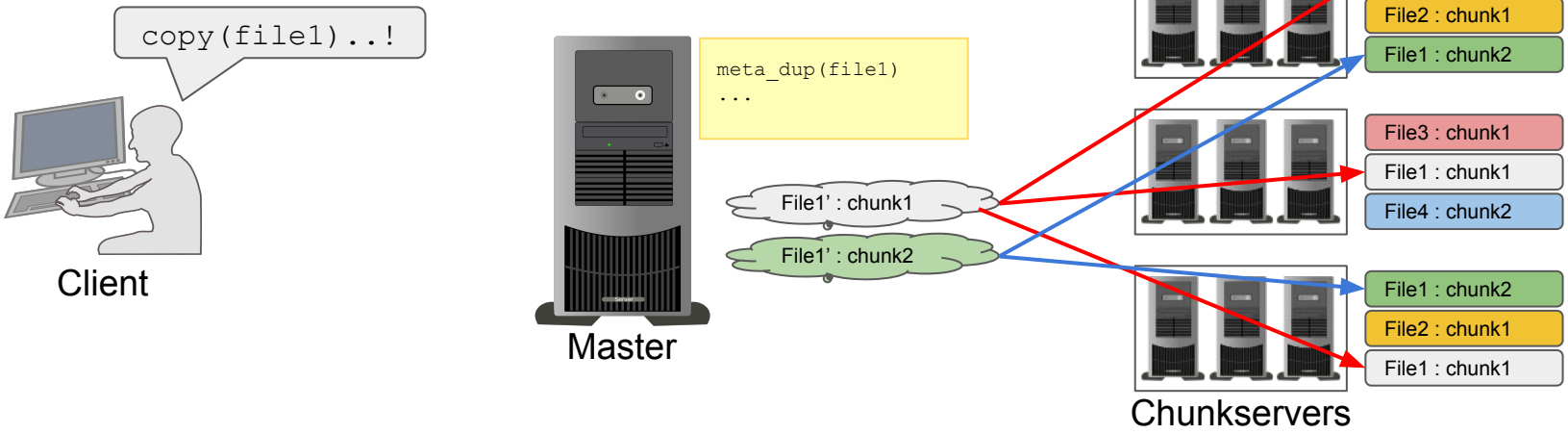


The `Snapshot` Operation

- Make a very fast (almost instantaneous) copy of a file or directory
- Copy-on-write

Step 3: Master applies copy by duplicating metadata

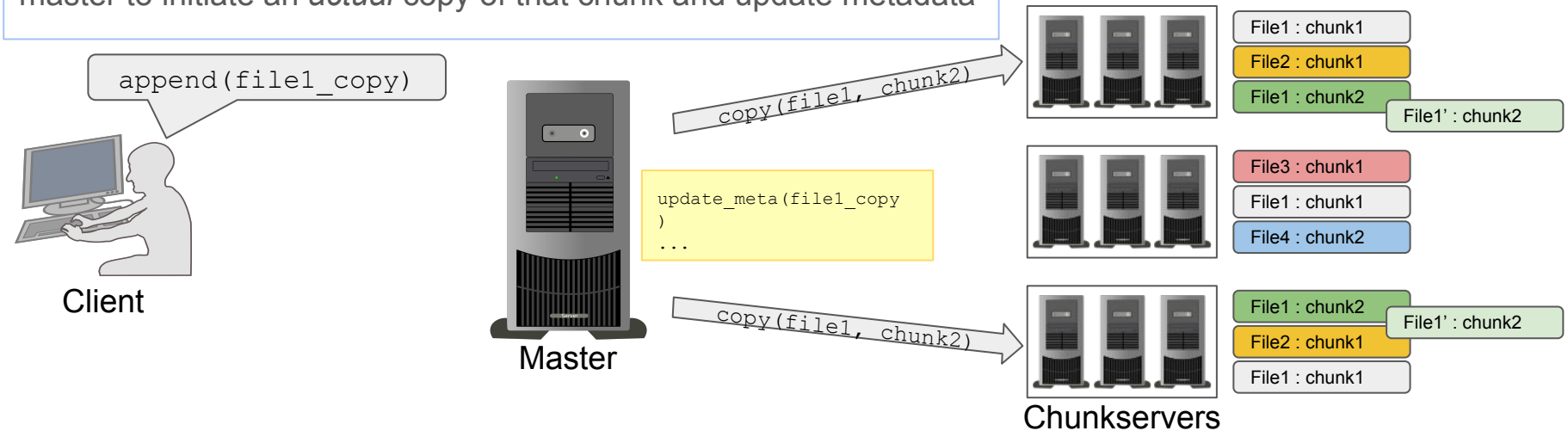
New files point to the *same* chunks as the source files



The `Snapshot` Operation

- Make a very fast (almost instantaneous) copy of a file or directory
- Copy-on-write

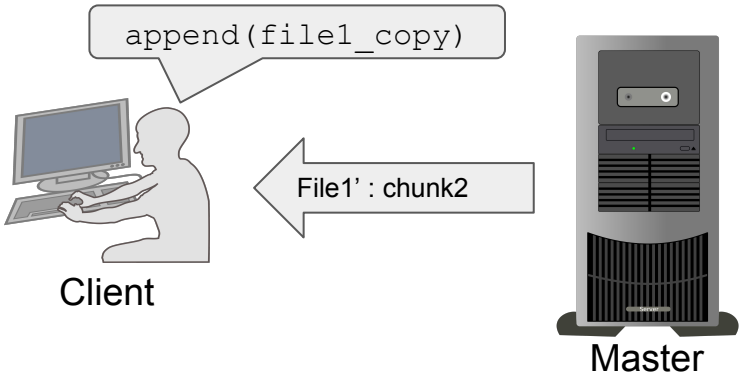
Step 4: The next write to any of the affected chunks causes the master to initiate an *actual* copy of that chunk and update metadata

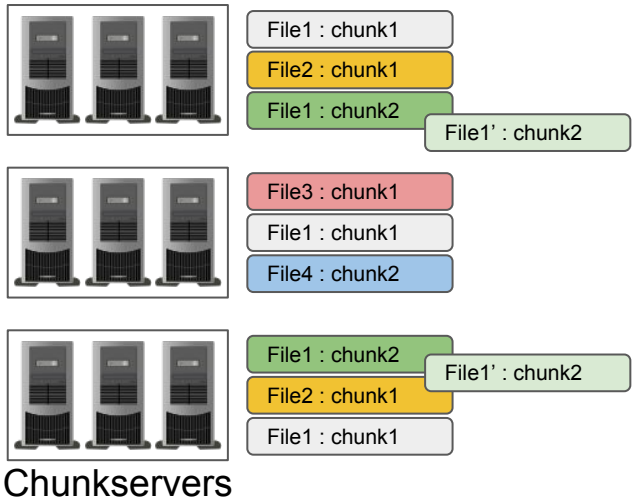


The `Snapshot` Operation

- Make a very fast (almost instantaneous) copy of a file or directory
- Copy-on-write

Step 5: Return the location of the *new* primary chunk and proceed as normal





The `Record_Append` Operation

- Atomic append operation
- Important for target use-case: distributed read/append
- Client specifies data, GFS determines byte offset
- GFS Guarantees record will be appended in its entirety *at least once*
  - If a replica fails during append, client retries the operation
  - May result in different data between replicas, or duplicate records
  - Successful append regions are *defined*
  - Failed regions are *inconsistent*
  - Clients are expected to be able to handle inconsistent regions
- Record_append

Fault Tolerance and Diagnosis

- High availability
  - Fast recovery - failure is the new normal
  - Chunkserver Replication - controlled by the master
  - Master Replication - “shadows”, not mirrors
- Data integrity
  - Checksum used to detect corruption
  - Verified on reads - stops propagation of errors
  - Recomputed on writes - optimized for appends
- Diagnostic Tools
  - Logs, logs, more logs
  - Written continuously and asynchronously
  - Can be freely deleted without affecting the system

Current State of the GFS ---> Colossus!

- GFS was introduced (to the public) in 2003
- GFS was designed primarily for batch operations (web crawling, search indexing, etc)
- GFS was *not* designed for low-latency, highly interactive web apps!
- Enter GFSv2 -- Colossus!
- Not much is publicly known about colossus! (only a few articles)
    - Removes the single-master paradigm
    - Built for real-time applications
    - Coupled with improved search-engine-engine “Caffeine”
    - Underlies *ALL OF GOOGLE’S ONLINE EMPIRE!* (search, gmail, drive, GCP, etc)

Resources

https://research.google.com/archive/gfs.html

http://www.cs.unc.edu/~jasleen/Courses/Fall09/slides/dfs-Google.pdf

https://pdos.csail.mit.edu/6.824/papers/gfs-faq.txt

https://www.wired.com/2012/07/google-colossus/

https://www.theregister.co.uk/2009/08/12/google_file_system_part_deux/

All images from https://openclipart.org/