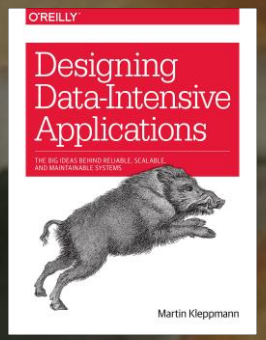
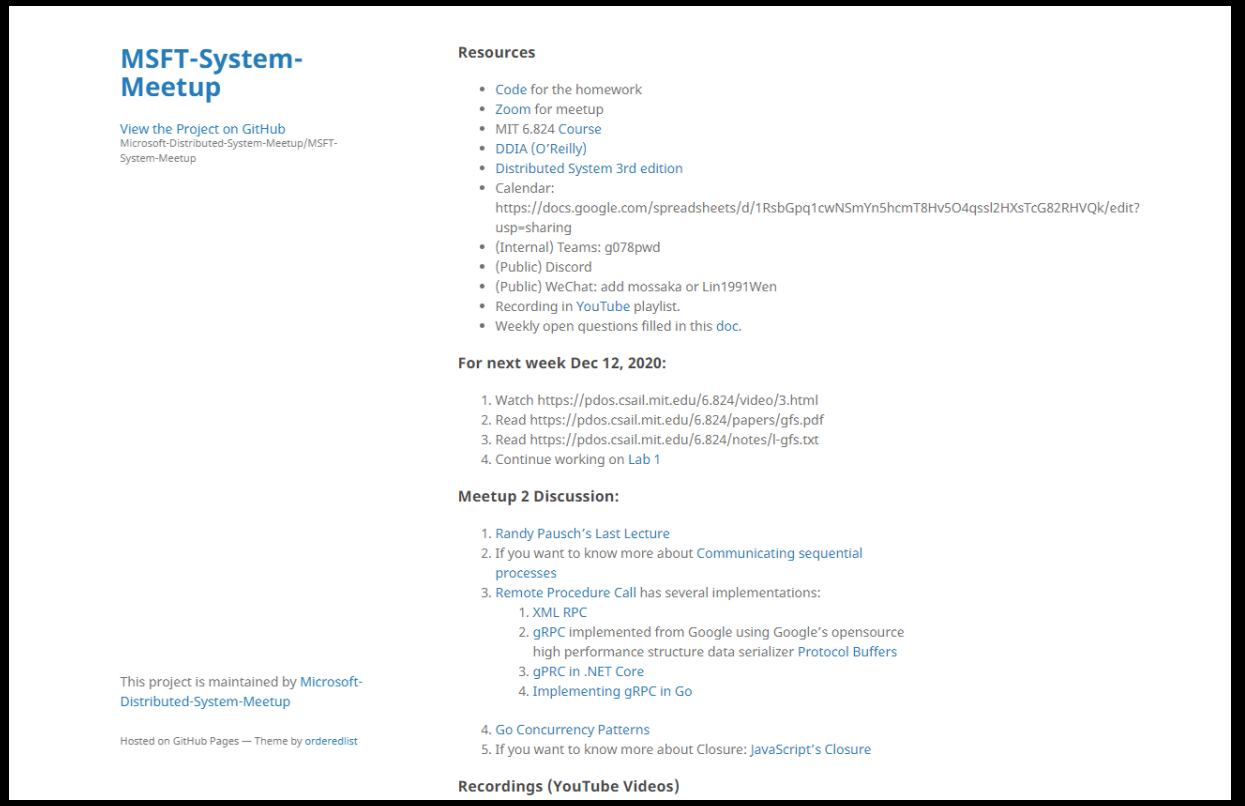
MSFT Sys Meetup



http://

Our new meetup website

- https://microsoft-distributed-system-meetup.github.io/MSFT-System-Meetup/



MSFT-System-Meetup

View the Project on GitHub
Microsoft-Distributed-System-Meetup/MSFT-System-Meetup

This project is maintained by Microsoft-Distributed-System-Meetup

Hosted on GitHub Pages — Theme by orderedlist

Resources

- Code for the homework
- Zoom for meetup
- MIT 6.824 Course
- DDIA (O'Reilly)
- Distributed System 3rd edition
- Calendar: https://docs.google.com/spreadsheets/d/1RsbGpq1cwNSmYn5hcmT8Hv5O4qssl2HXsTcG82RHVQk/edit?usp=sharing
- (Internal) Teams: g078pwd
- (Public) Discord
- (Public) WeChat: add mossaka or Lin1991Wen
- Recording in YouTube playlist.
- Weekly open questions filled in this doc.

For next week Dec 12, 2020:

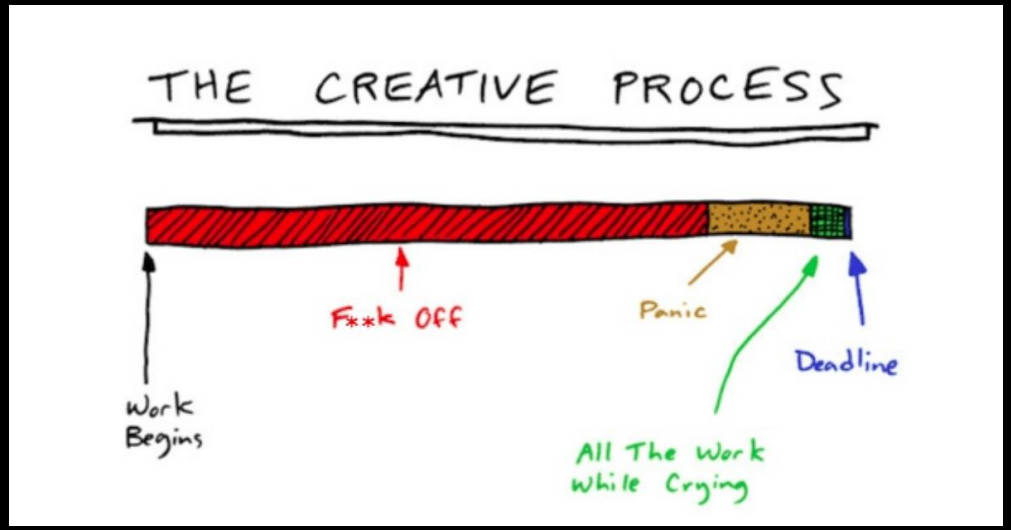
1. Watch https://pdos.csail.mit.edu/6.824/video/3.html
2. Read https://pdos.csail.mit.edu/6.824/papers/gfs.pdf
3. Read https://pdos.csail.mit.edu/6.824/notes/l-gfs.txt
4. Continue working on Lab 1

Meetup 2 Discussion:

1. Randy Pausch's Last Lecture
2. If you want to know more about Communicating sequential processes
3. Remote Procedure Call has several implementations:
    1. XML RPC
    2. gRPC implemented from Google using Google's opensource high performance structure data serializer Protocol Buffers
    3. gPRC in .NET Core
    4. Implementing gRPC in Go
4. Go Concurrency Patterns
5. If you want to know more about Closure: JavaScript's Closure

Recordings (YouTube Videos)

Lab 1 Due next Saturday!!



6.824

Lecture 3 & GFS

The Big Storage System trend

“The storage turns out to be a key abstraction”

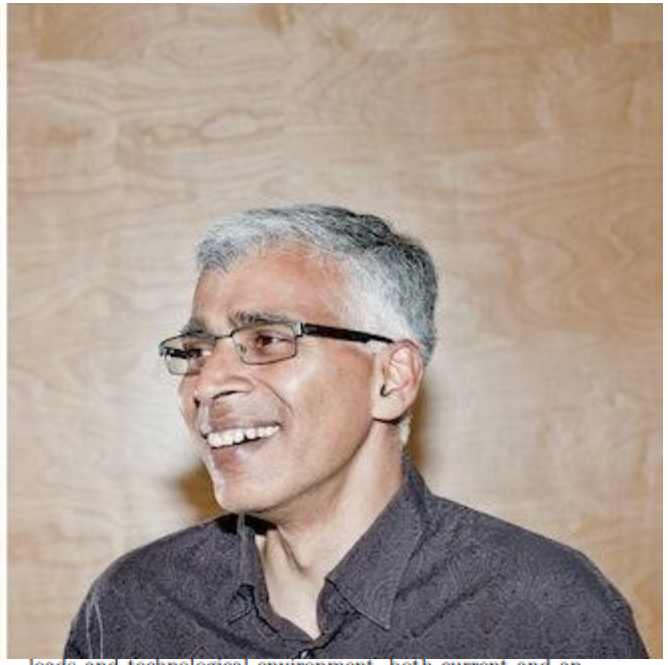
“A simple storage interface is incredible useful”

The Google File System is a successful real-world design.

The Performance and Consistency tradeoff (revisited)

1. High performance -> shared data over many servers
2. Many distributed servers -> constant faults as a norm
3. Fault tolerance -> replication
4. Replication -> potential inconsistencies
5. Better consistency -> low performance ☹

The Google File System



tributed file systems, our design has been driven by observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system assumptions. This has led us to reexamine traditional choices and explore radically different design points.

The file system has successfully met our storage needs. It is widely deployed within Google as the storage platform for the generation and processing of data used by our service as well as research and development efforts that require large data sets. The largest cluster to date provides hundreds of terabytes of storage across thousands of disks on over a thousand machines, and it is concurrently accessed by hundreds of clients.

In this paper, we present file system interface extensions designed to support distributed applications, discuss many aspects of our design, and report measurements from both micro-benchmarks and real world use.

Categories and Subject Descriptors

D [**4**]: 3—*Distributed file systems*

General Terms

Design, reliability, performance, measurement

Keywords

Fault tolerance, scalability, data storage, clustered storage

*The authors can be reached at the following addresses: {sanjay,hgobioff,shuntak}@google.com

loads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions. We have reexamined traditional choices and explored radically different points in the design space.

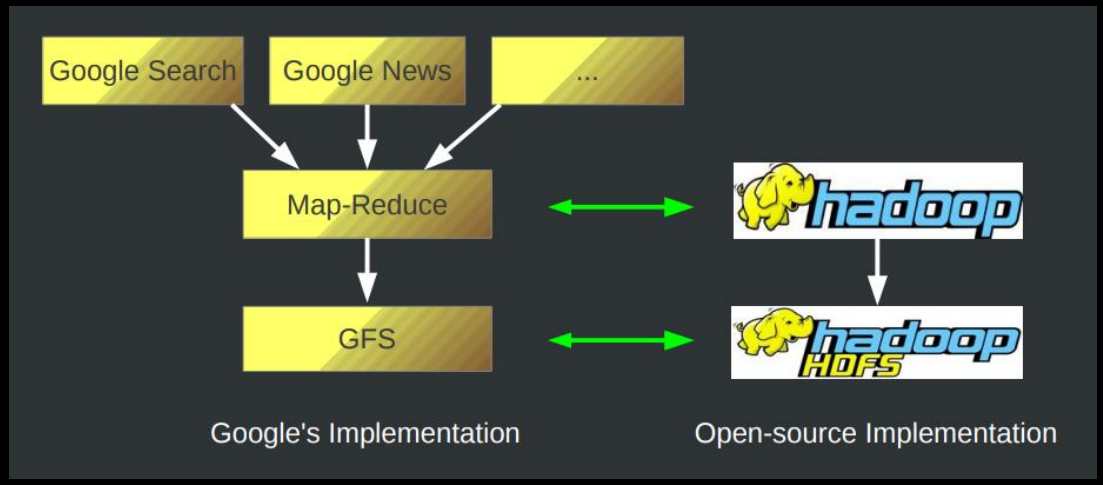
First, component failures are the norm rather than the exception. The file system consists of hundreds or even thousands of storage machines built from inexpensive commodity parts and is accessed by a comparable number of client machines. The quantity and quality of the components virtually guarantee that some are not functional at any given time and some will not recover from their current failures. We have seen problems caused by application bugs, operating system bugs, human errors, and the failures of disks, memory, connectors, networking, and power supplies. Therefore, constant monitoring, error detection, fault tolerance, and automatic recovery must be integral to the system.

Second, files are huge by traditional standards. Multi-GB files are common. Each file typically contains many application objects such as web documents. When we are regularly working with fast growing data sets of many TBs comprising billions of objects, it is unwieldy to manage billions of approximately KB-sized files even when the file system could support it. As a result, design assumptions and parameters such as I/O operation and block sizes have to be revisited.

Third, most files are mutated by appending new data rather than overwriting existing data. Random writes within a file are practically non-existent. Once written, the files are only read, and often only sequentially. A variety of data share these characteristics. Some may constitute large

Introduction

- The GFS is a scalable distributed file system for large distributed data-intensive applications. It provides fault tolerance while running on inexpensive commodity hardware (15,000 PCs), and it delivers high aggregate performance to a large number of clients.
- Google used it to store the entire Web.



Motivation

“We have designed and implemented the Google File System (GFS) to meet the rapidly growing demands of Google’s data processing needs. GFS shares many of the same goals as previous distributed file systems such as performance, scalability, reliability, and availability. However, its design has been driven by key observations of our application workloads and technological environment, both current and anticipated, that reflect a marked departure from some earlier file system design assumptions.”

The heretic view of GFS

1. Component failures are the norm rather than exception
    1. Constant monitoring, error detection, fault tolerance, automatic recovery are much needed
2. Files are huge (multi-GB)
3. Most files are mutated by appending new data rather than overwriting exiting data
4. Codesigning the application and the file system API benefits the overall system by increasing flexibility

Interface

1. “We support the usual operations to create, delete, open, close, read, and write files”
2. Snapshot
3. Record append

Record append allows multiple clients to append data to the same file concurrently while guaranteeing the atomicity of each individual client’s append.

Design Overview Key Points

1. Single master (M) (operation logs and checkpoints are replicated) and multiple chunkservers (CS) and multiple clients (C).
2. File -> 64 MB chunks (a normal linux chunk is 4KB)
    1. It reduces clients' need to interact with the M because reads and writes on the same chunk require only one initial request to the M for chunk location information.
    2. A client is more likely to perform operations on a given chunk. Persistent TCP connection to the CS over a long time
    3. Reduces the size of the metadata stored on the M

Design Overview Key Points

3. Lazy space allocation to counter internal fragmentation (physical allocation of space is delay ALAP)
4. CS replicated on 3
5. Chunk -> replicated and spread over CS (potential consistency issues?)
6. M's data
    1. In memory: (mainly metadata)
        1. File name -> chunk handlers (64-bit pointer)
        2. Chunk handler -> version number (NV), list of CS (V), primary (V), lease time(V)
    2. On disk:
        1. Operation Logs (namespaces, file-to-chunk mapping, mutations)
        2. Checkpoint (compact B-tree, to reduce the size of logs)

Design Overview Key Points

7. CS's data: store chunks on Linux file system
8. M deals with metadata, CS deals with data
9. Where do M know the chunk handler -> list of CS mapping?
    1. Asks CS about its chunks at M startup and whenever a CS joins the cluster
10. M periodically scans its states in background
    1. Chunk garbage collection (GC), CS failure handling, chunk migration
11. M sends HeartBeat msg to CSs for lease management and chunk locations.

Design Overview

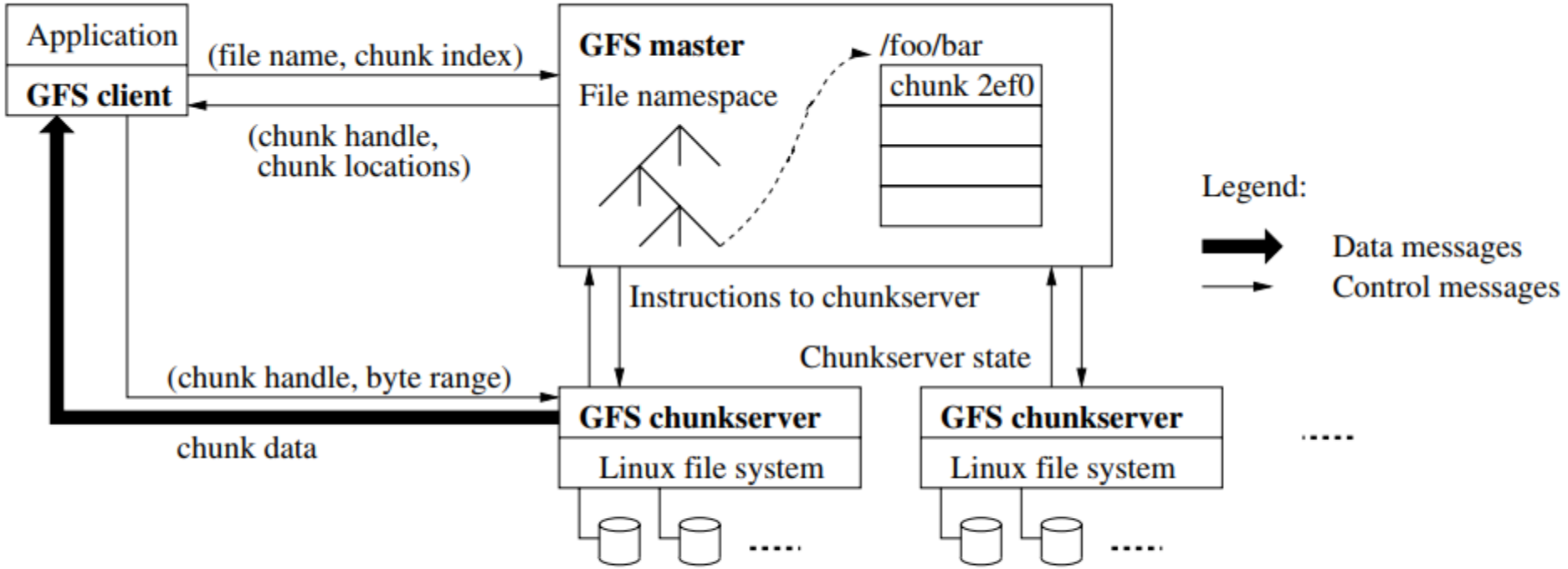


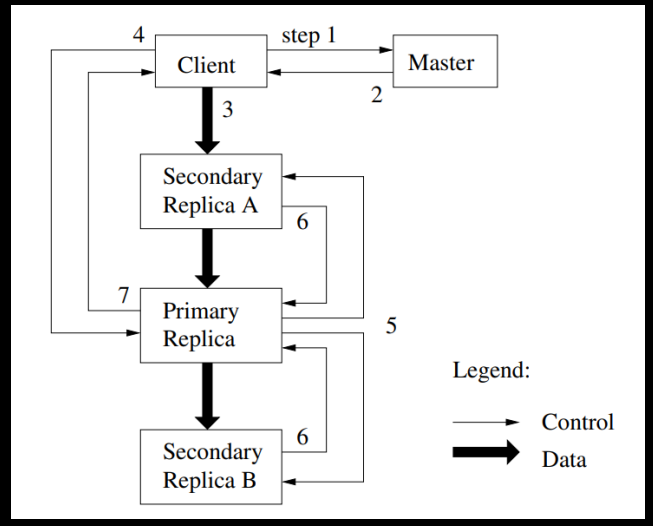
Figure 1: GFS Architecture

Design: READ (easy)

1. C Send(filename, offset) to M (if not cached)
2. M lookup(filename, offset) for chunk handler (64-bit address)
3. M returns list of CS (only with the latest chunk version number (CVN))
4. C caches handler + list of CS
5. C Send(chunk handler, offset) to nearest CS
6. CS Read(chunk handler, offset) on LFS -> data

Design: MUTATIONS (hard!!)

- Data mutations may be *writes* or record appends, or snapshot
- Record append: data to be appended *atomically at least once,* in the presence of concurrent mutations, but at an offset of M's choosing
- Each mutation is performed at all the chunk's replicas



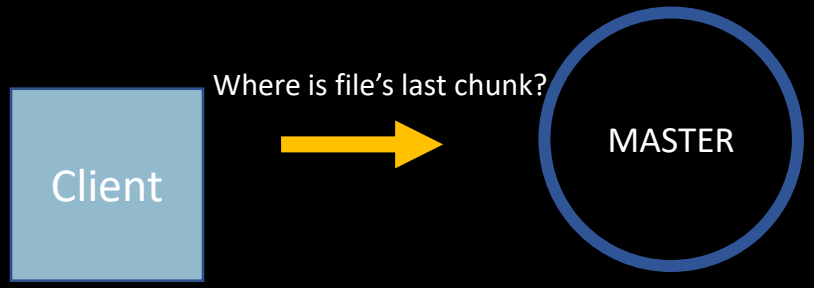
Lease mechanism

The M grants a chunk lease to one of the replicas, call it primary (P). P picks a serial order for all mutations to the chunk. All replicas follow this order when applying mutations. Thus, the global mutation order is defined first by the lease grant order chosen by M, and within a lease by the serial numbers assigned by P

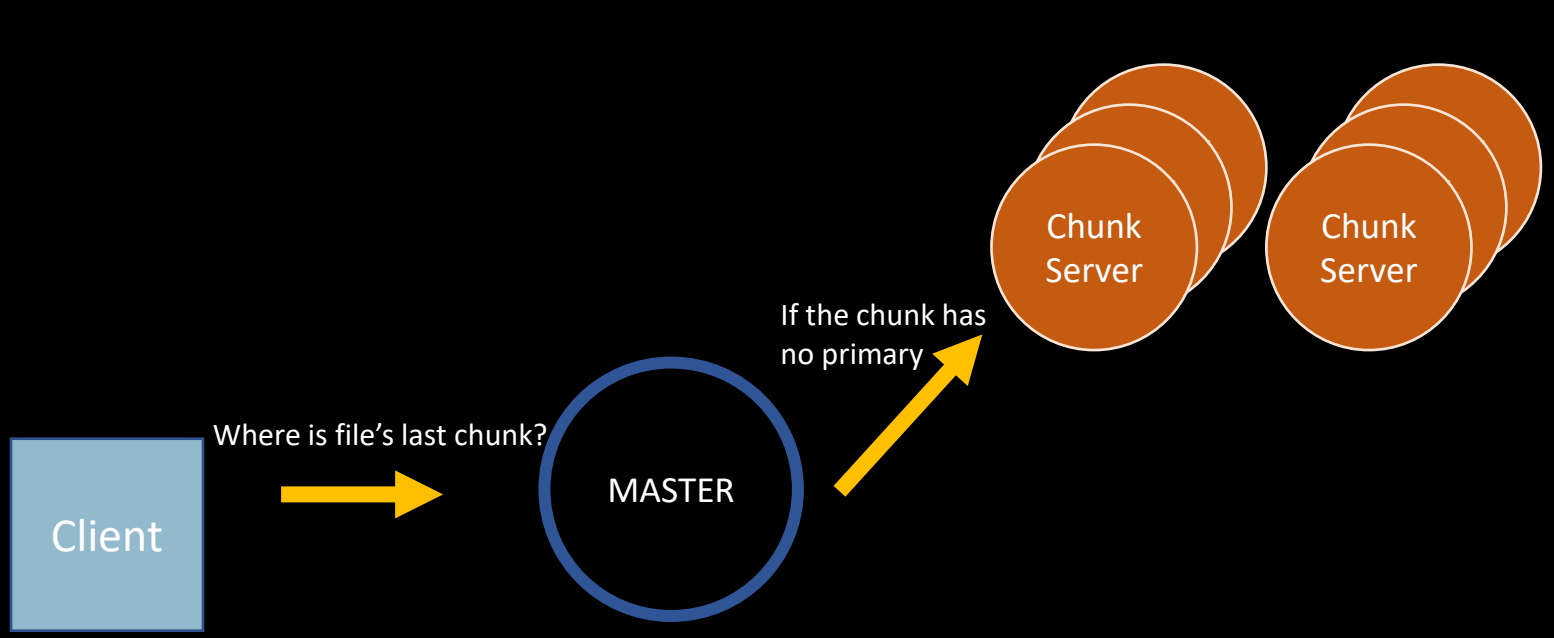
A lease has an initial timeout of 60s.

As long as the chunk is being mutated, P can request extensions from the M indefinitely. These extension requests and grants are piggybacked on the Heartbeat Messages regularly exchanged between the M and all CS. Even if the master loses communication with ha primary, it can safely grant a new lease to another replica after the old lease expires

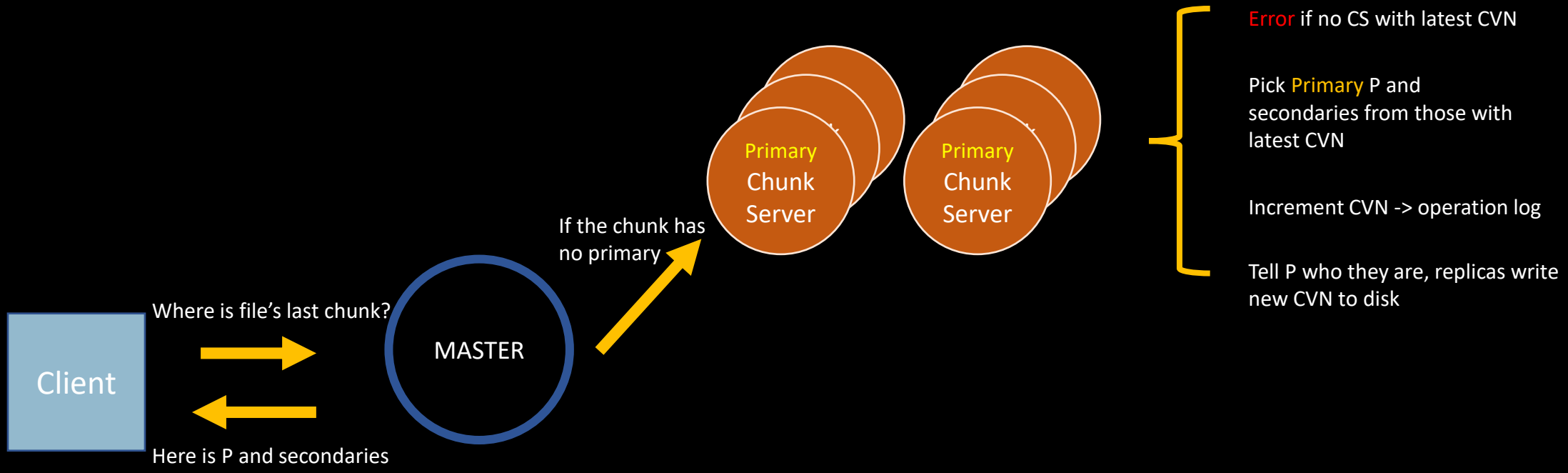
Data Mutation step 1



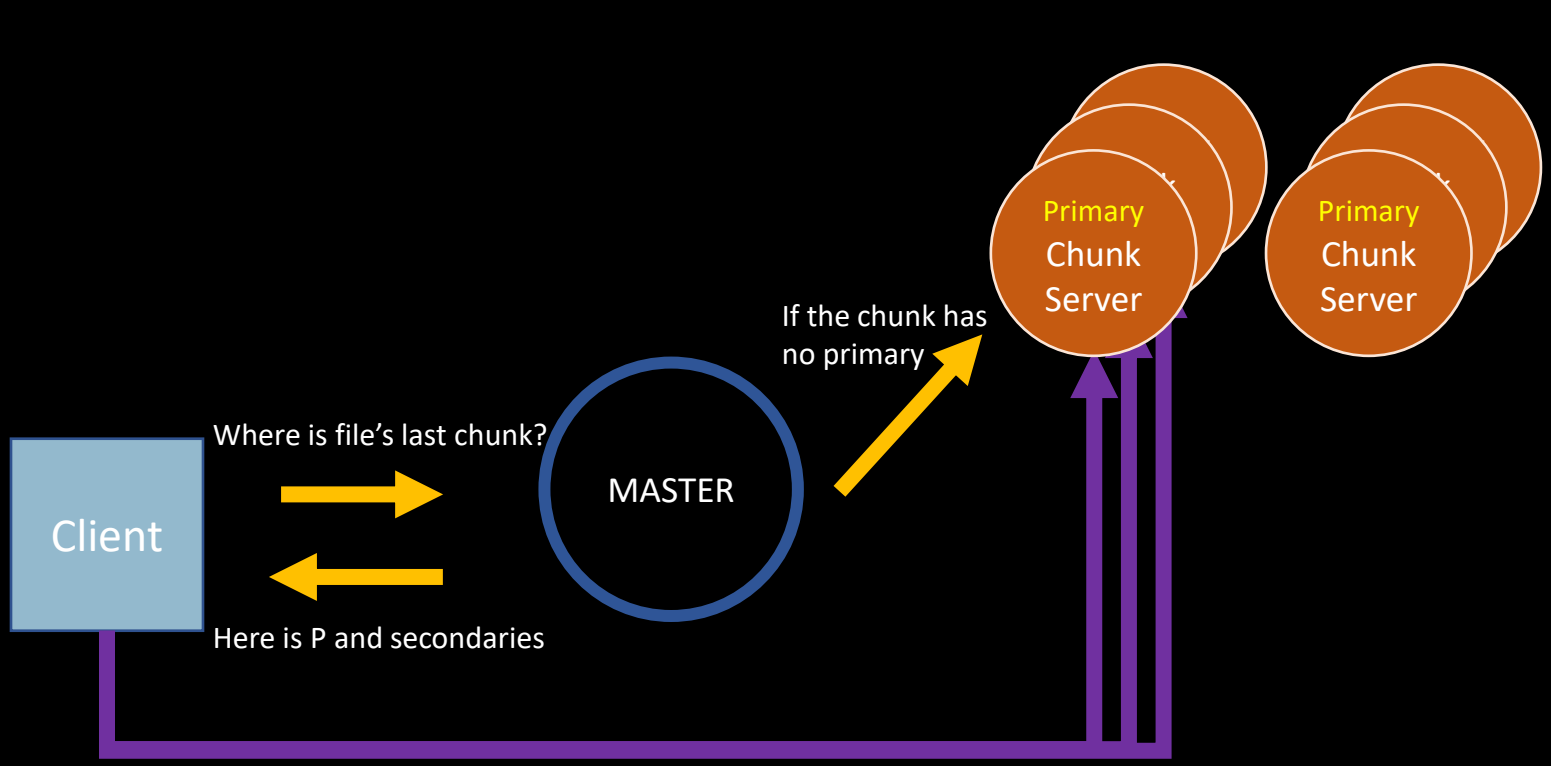
Data Mutation step 2



Data Mutation step 3



Data Mutation step 4 (Data flow vs Control flow)



Error if no CS with latest CVN

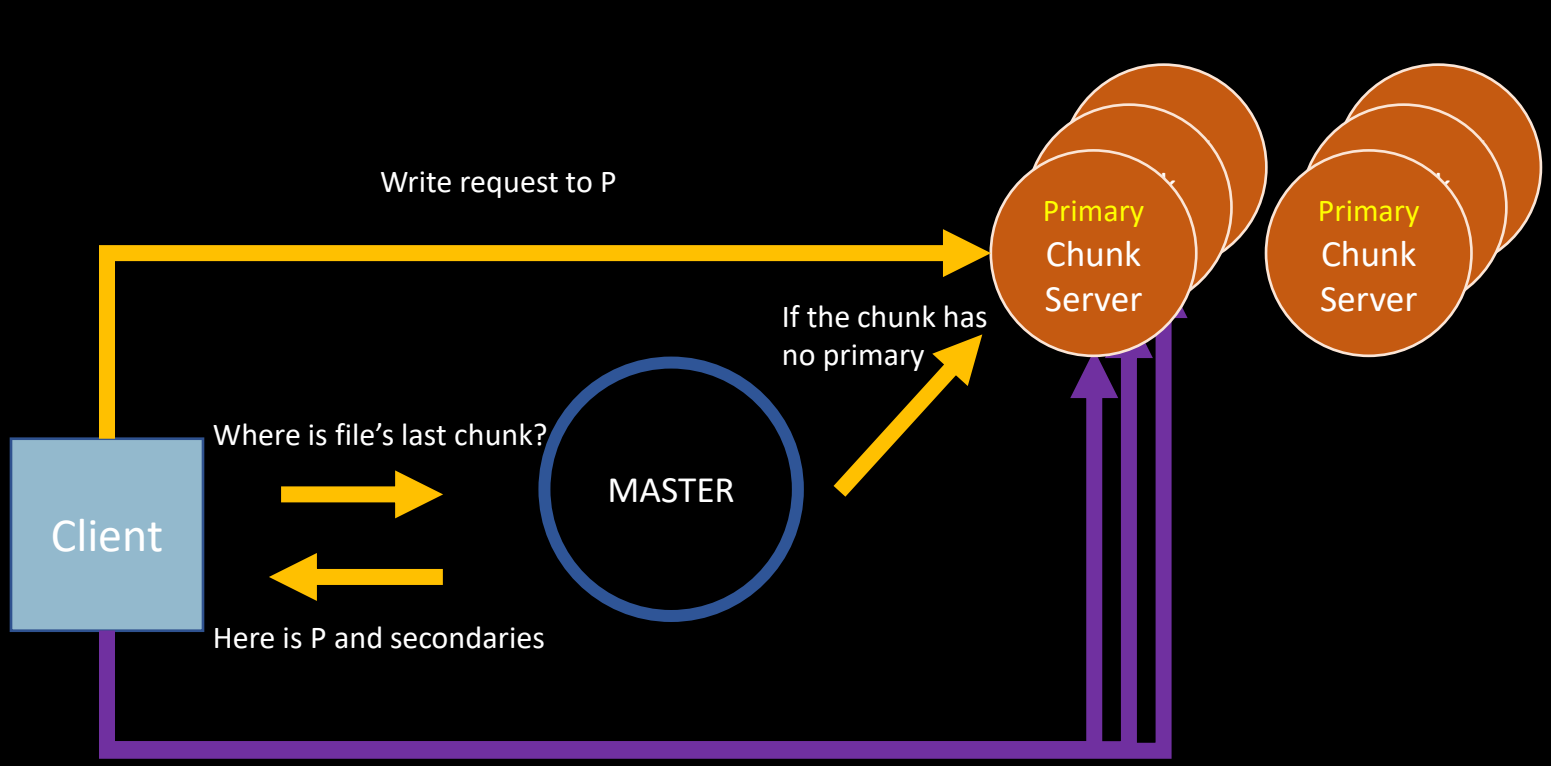
Pick Primary P and secondaries from those with latest CVN

Increment CVN -> operation log

Tell P who they are, replicas write new CVN to disk

C sends data to all CS replicas, in any order. Each CS store the data in LRU buffer cache until the data is used or aged out. Wait until all replicas have ack receiving the data

Data Mutation step 5 (Data flow vs Control flow)



Error if no CS with latest CVN

Pick Primary P and secondaries from those with latest CVN

Increment CVN -> operation log

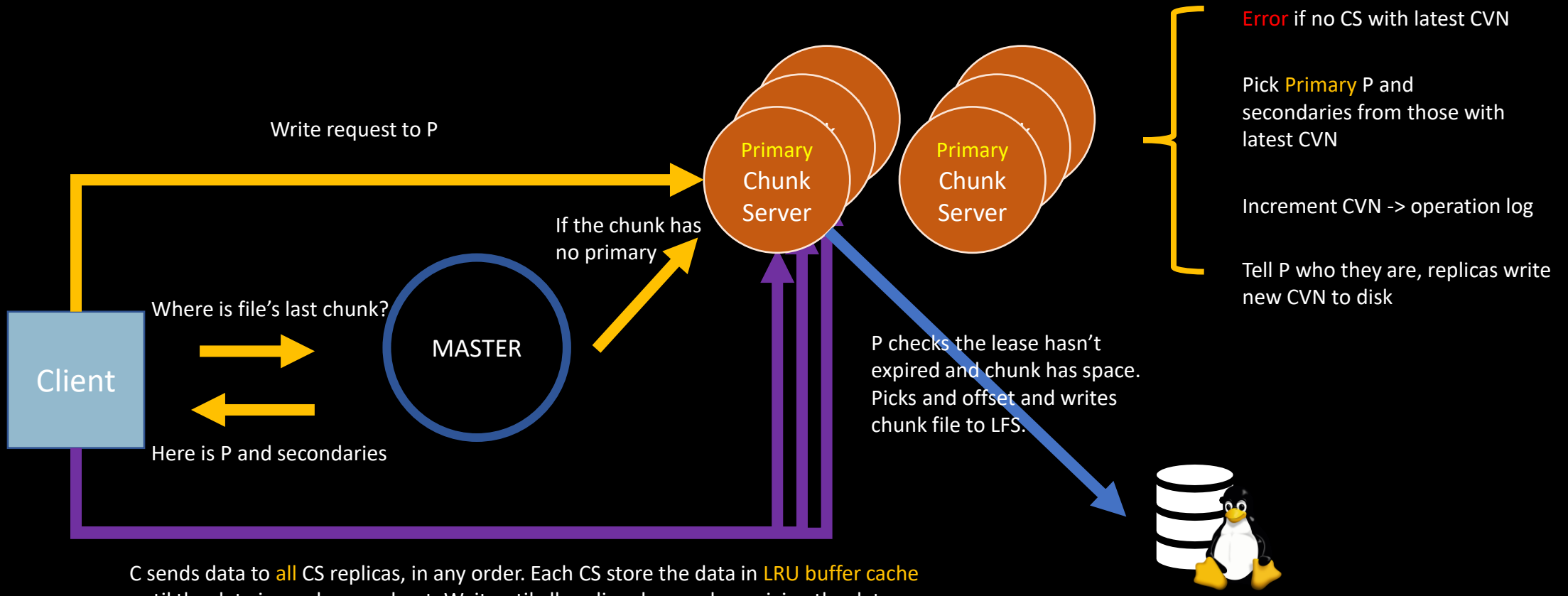
Tell P who they are, replicas write new CVN to disk

C sends data to all CS replicas, in any order. Each CS store the data in LRU buffer cache until the data is used or aged out. Wait until all replicas have ack receiving the data

S form a chain
Chain respects network topology
Allows fast replication

“By decoupling the data flow from the control flow, we can improve performance by scheduling the expensive data flow based on the network topology regardless of which CS is the P”

Data Mutation step 6 (Data flow vs Control flow)

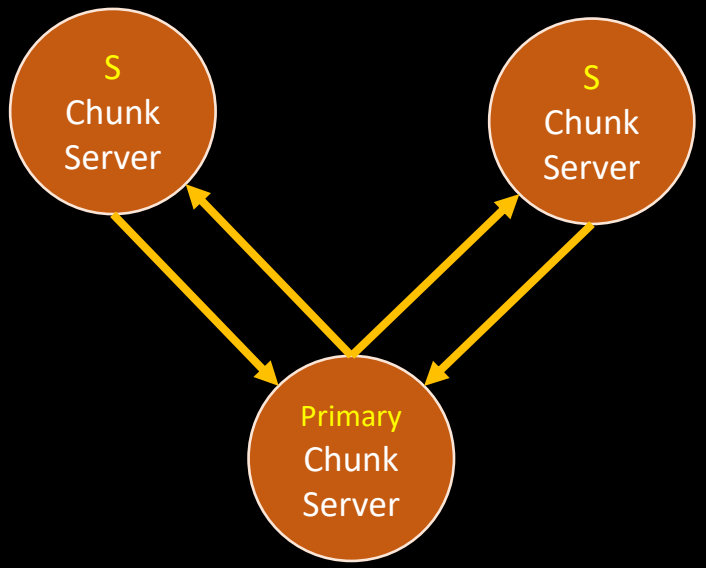


C sends data to all CS replicas, in any order. Each CS store the data in LRU buffer cache until the data is used or aged out. Wait until all replicas have ack receiving the data

S form a chain
Chain respects network topology
Allows fast replication

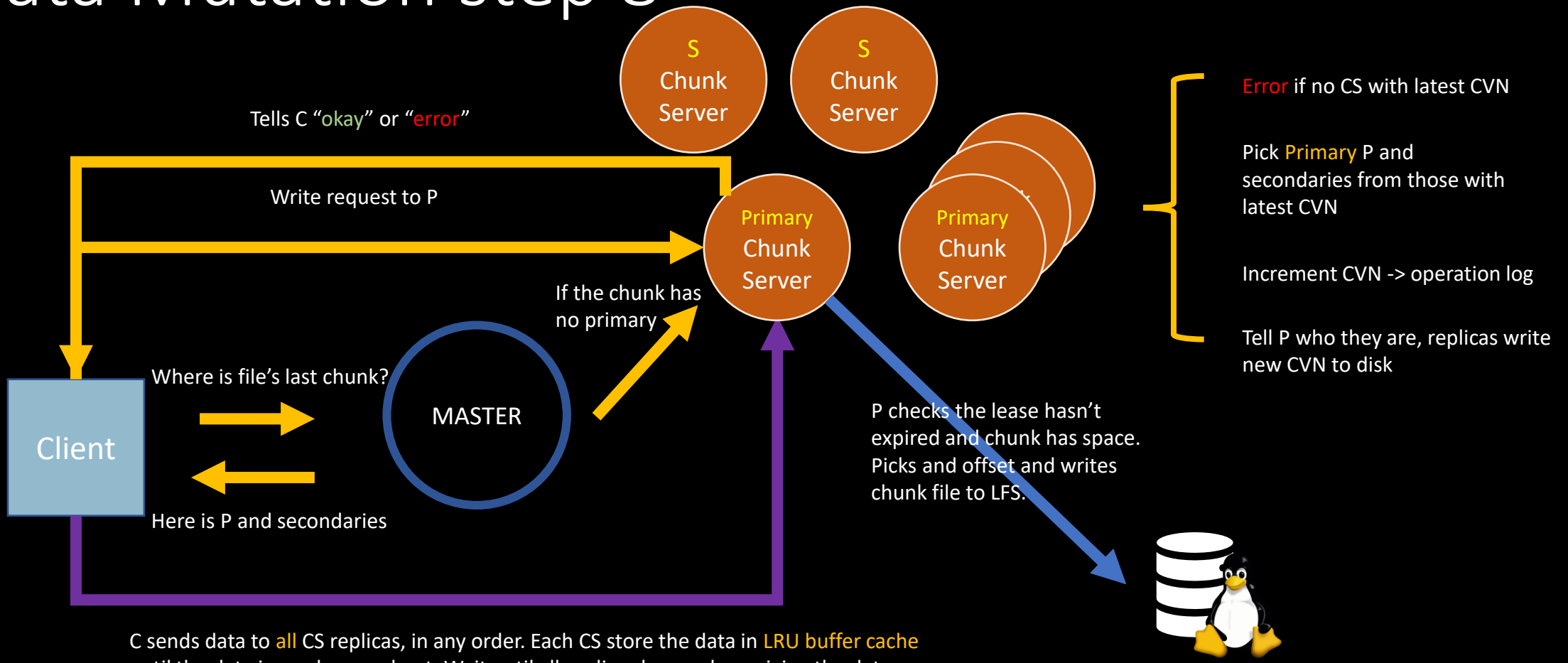
Data Mutation step 7

The S replicas all reply to the P indicating that they have completed the operation



The P forwards the write request to all S replicas. Each S replica applies mutations in the same serial number order assigned by P

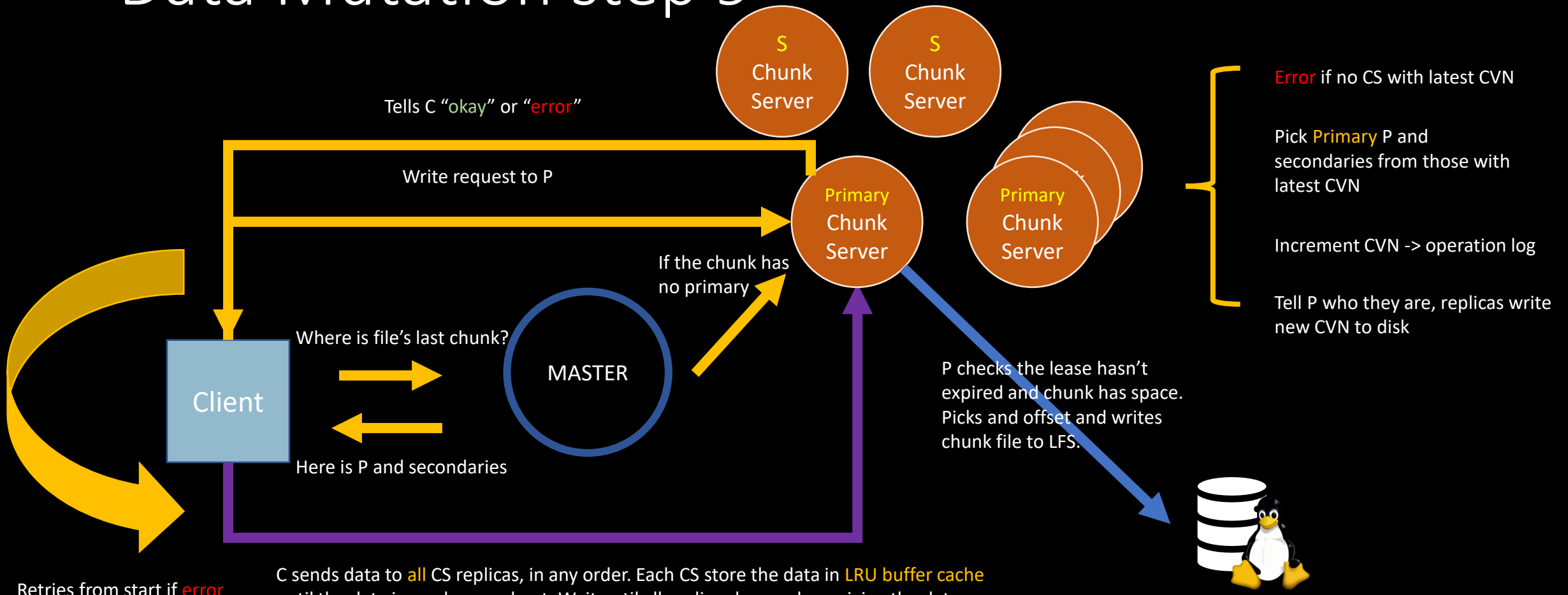
Data Mutation step 8



C sends data to all CS replicas, in any order. Each CS store the data in LRU buffer cache until the data is used or aged out. Wait until all replicas have ack receiving the data

S form a chain
Chain respects network topology
Allows fast replication

Data Mutation step 9



C sends data to all CS replicas, in any order. Each CS store the data in LRU buffer cache until the data is used or aged out. Wait until all replicas have ack receiving the data

S form a chain
Chain respects network topology
Allows fast replication

Record Append Algorithm

- Example: merging results from multiple machines in one file. Using file as producer-consumer queue.

1. Application originates record append request.
2. C sends request to M
3. M responds chunk handle, P CS + S CS locations
4. C push write data to all locations
5. P checks if record fits in specific chunk
    1. No: then P pads the chunk, tells S to do the same, and informs the C. C then retries the append with the next chunk
    2. Yes: P appends the record, tell S to do, receives responses from S, sends “okay” or “error” to C.

Consistency Model

- Consistency trades for High aggregate performance
- Definition:
  - A file is consistent if all clients will always see the same data, regardless which replicas they read from.
  - A file region is defined after a file data mutation if it is consistent and clients will see what the mutation writes in its entirety.

Guarantees by GFS

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

- File namespace mutations are atomic (e.g. creation) and are handled exclusively by M
- Successful mutation without interference -> defined
- Concurrent successful mutations -> undefined by consistent
- Failed mutation -> inconsistent (and undefined)
  - Different clients may see different data at different times

"In retrospect, that turned out to be a lot more painful than anyone expected."

------ Quinlan

An example of stale replica: Append A

A	A	A

An example of stale replica: Append B

B	B	
A	A	A

Send error to C

An example of stale replica: Append C

C	C	C
B	B	
A	A	A

An example of stale replica: Client retries

B	B	B
C	C	C
B	B	
A	A	A

Retry to append B

An example of stale replica: Client dies

D		D
B	B	B
C	C	C
B	B	
A	A	A

After C appends D and receive an error, C dies immediately. Now some of the replicas may never see D.

Consistency Model: what can we expect?

If the primary P tells a client C that a record append succeeded, then any reader that subsequently opens the file and scans it will see the appended record somewhere.

Weak consistency for chunk operations

A failed mutation leaves chunks inconsistent.

Split Brain

Problem:

Suppose S_1 is the P for a chunk, and the network between M and S_1 fails. M will notice and designate some other server as P, say S_2. Since S_1 didn't actually fail, are there now two P for the same chunk?

Solution:

The lease mechanism prevents this. M grants S_1 a 60 second lease to be P. S_1 knows to stop being P when its lease expires. M will wait until 60s passed and then grant a lease to S_2. So S_2 won't start acting as primary until after S_1 stops.

What I skipped

- Snapshot
- Network topology
- Namespace Management and locking
  - Allows concurrent mutations in the same directory
- Chunk creation, re-replication, rebalancing (very cool!)
- Garbage Collection rather than eager deletion
- Read-only shadow masters in the case of master failure
- Check summing for data integrity
- Measurement

Summary

- The good
  - Global cluster file system as universal infrastructure
  - Separation of naming (M) from storage (CS)
  - sharding for parallel throughput
  - Huge chunks to reduce overheads
  - Prevents split-brain scenario
- The ugly
  - Single master performance
  - CS not efficient for small files (BigTable is the solution)
  - Consistency too weak?

Fault Tolerance

Questions from the course notes:
https://pdos.csail.mit.edu/6.824/notes/l-gfs.txt

Takeways: Retrospective Interview with GFS engineer

MCKUSICK In retrospect, how would you handle this differently?
QUINLAN I think it makes more sense to have a single writer per file.
MCKUSICK All right, but what happens when you have multiple people wanting to append to a log?
QUINLAN You serialize the writes through a single process that can ensure the replicas are consistent.