CS 744: GOOGLE FILE SYSTEM

Shivaram Venkataraman

Fall 2019

ANNOUNCEMENTS

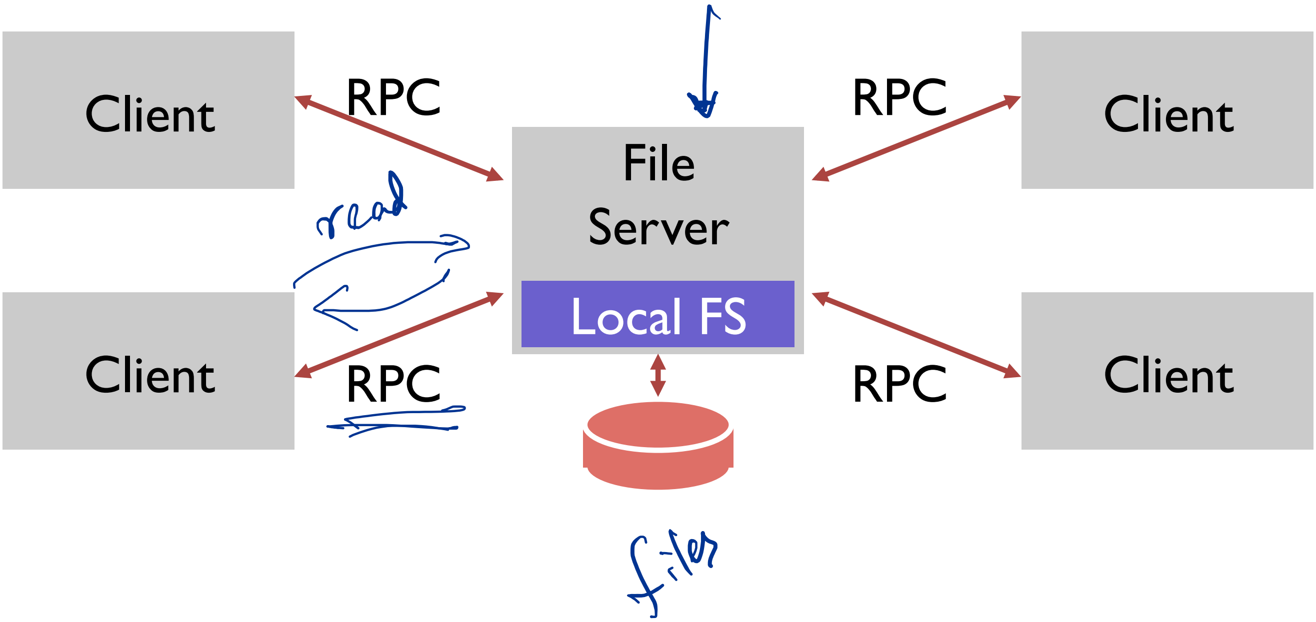
- Assignment 1 out later today
- Group submission form
- Anybody on the waitlist?

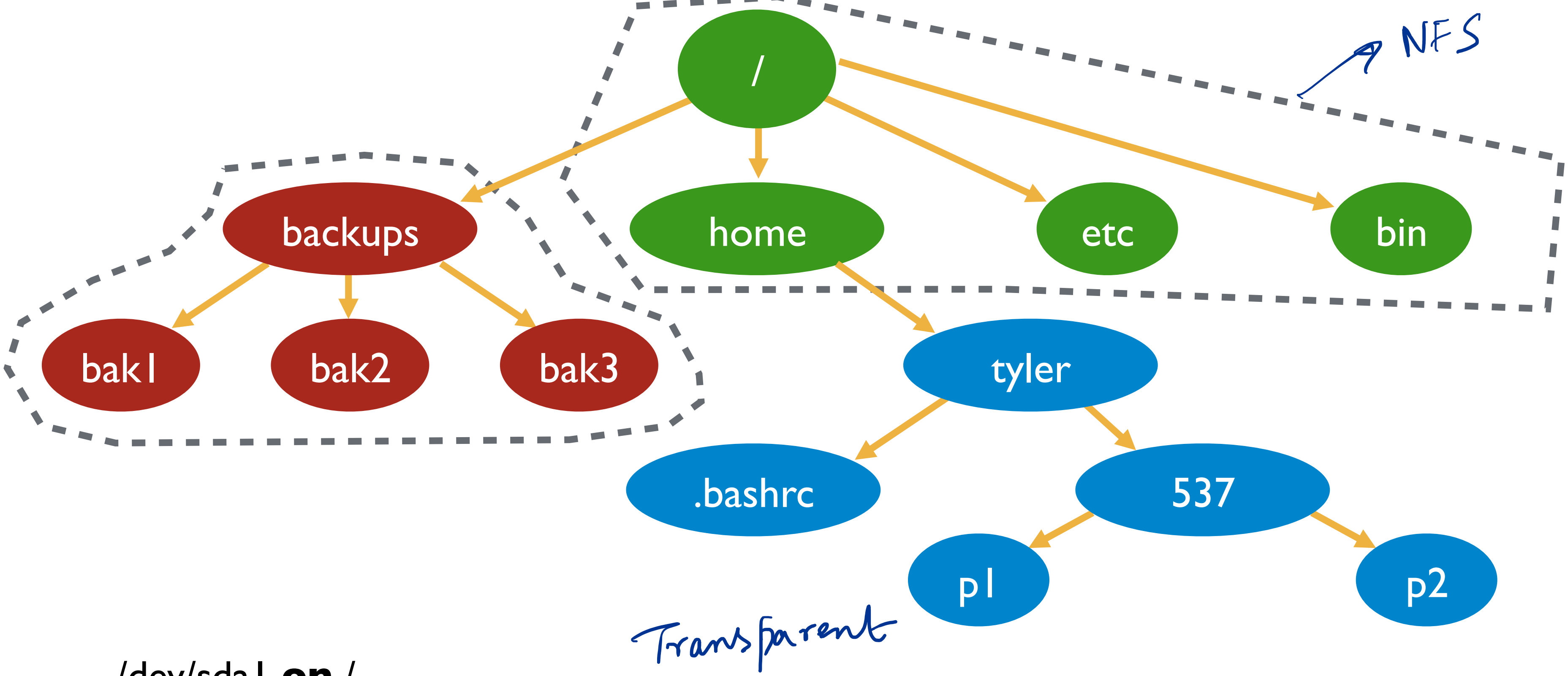
OUTLINE

1. Brief history
2. GFS
3. Discussion
4. What happened next?

HISTORY OF DISTRIBUTED FILE SYSTEMS

SUN NFS


Client
RPC
File
Server
Local FS
RPC
Client
read
Client
RPC
RPC
Client
files



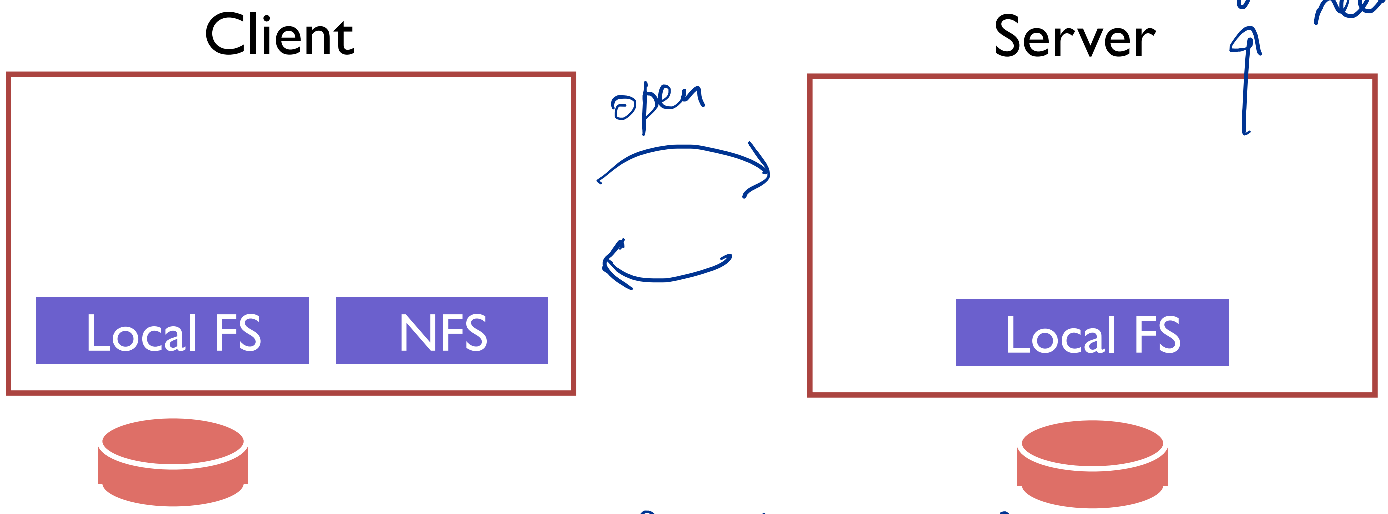
/dev/sda1 **on** /

/dev/sdb1 **on** /backups

NFS **on** /home

POSIX

FILE HANDLES

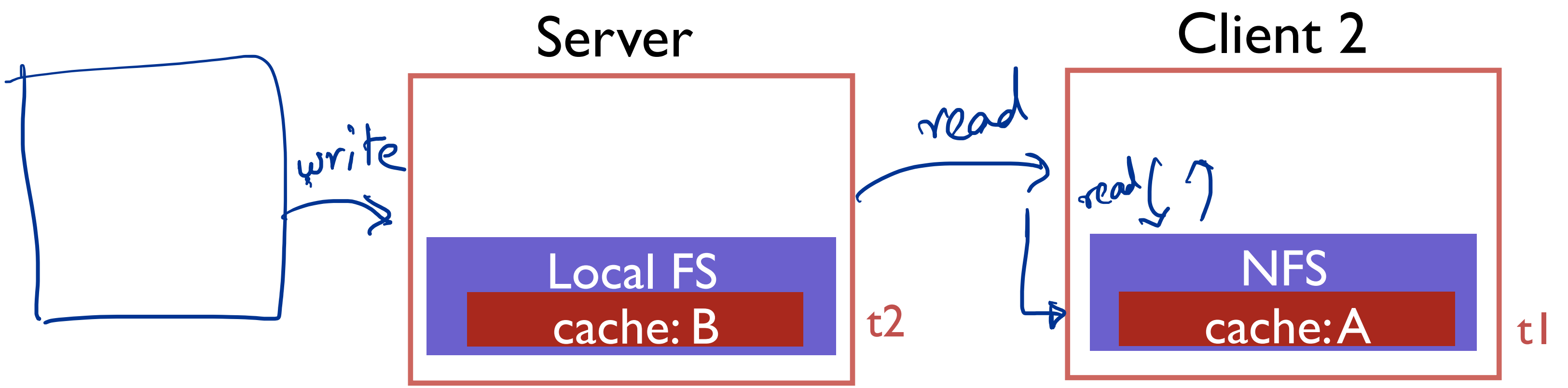


does not need to keep any state

fd = open("/tmp/foo")

Examples
- open
- read

CACHING



Client cache records time when data block was fetched (t1)

Before using data block, client does a STAT request to server

- get's last modified timestamp for this file (t2) (not block…)
- compare to cache timestamp
- refetch data block if changed since timestamp (t2 > t1)

check last updated timestamp

NFS FEATURES

~ 1985 or 80

NFS handles client and server crashes very well; robust APIs that are:

- stateless: servers don't remember clients
- idempotent: doing things twice never hurts

Caching is hard, especially with crashes

Problems:

– Consistency model is odd (client may not see updates until 3s after file closed)
– Scalability limitations as more clients call stat() on server

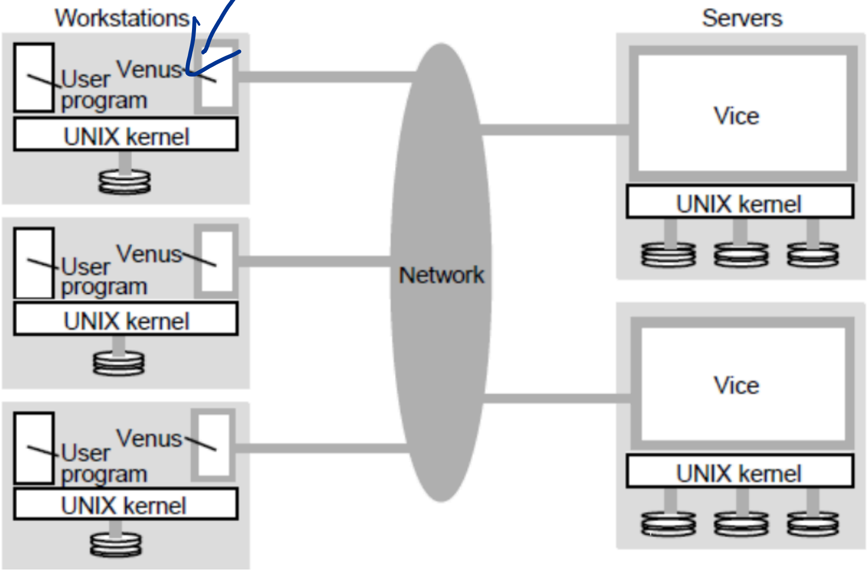
ANDREW FILE SYSTEM

POSIX API

local disk

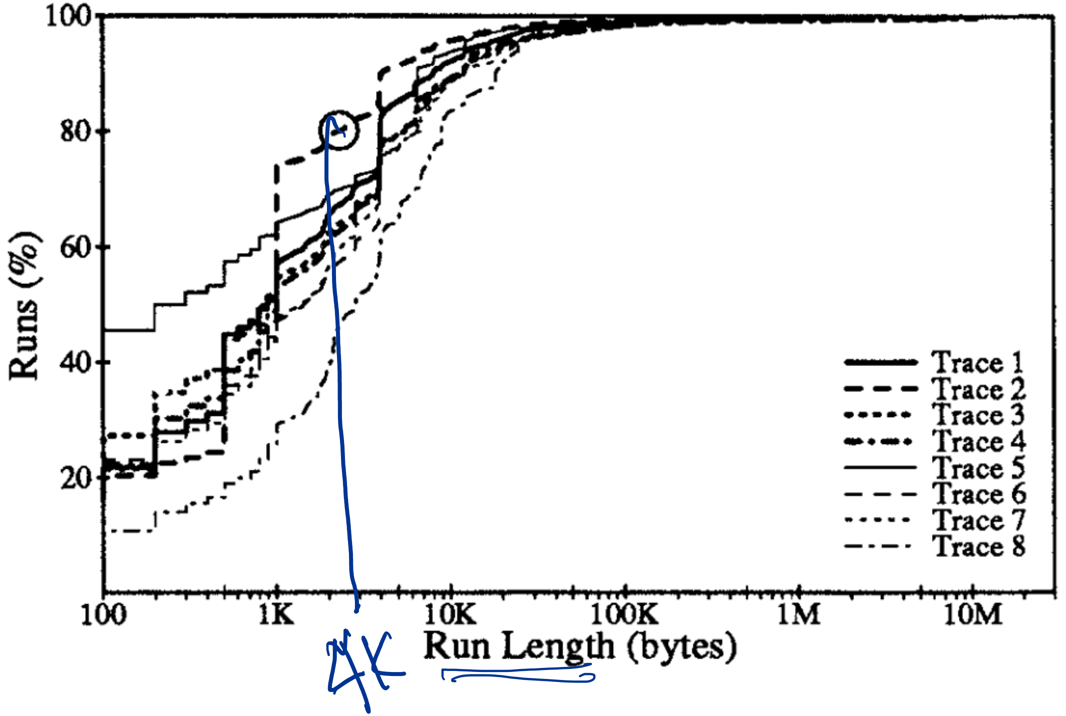
open("/tmp/foo")

Architecture



- Design for scale
- Whole-file caching
- Callbacks from server

WORKLOAD PATTERNS (1991)



Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout

WORKLOAD PATTERNS (1991)

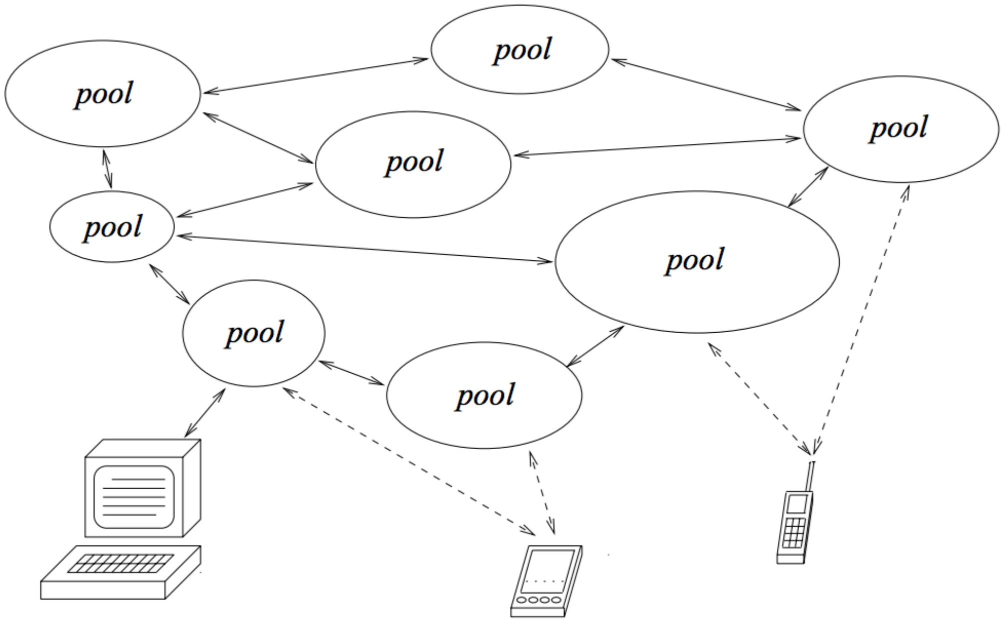
File Usage	Type of Transfer	Accesses (%)		Bytes (%)	
Read-only	Whole-file	78	(64-91)	89	(46-96)
	Other sequential	19	(7-33)	5	(2-29)
	Random	3	(1-5)	7	(2-37)
Write-only	Whole-file	67	(50-79)	69	(56-76)
	Other sequential	29	(18-47)	19	(4-27)
	Random	4	(2-8)	11	(4-41)
Read/write	Whole-file	0	(0-0)	0	(0-0)
	Other sequential	0	(0-0)	0	(0-0)
	Random	100	(100-100)	100	(100-100)

OCEANSTORE/PAST

Wide area storage systems

Fully decentralized

Built on distributed hash tables (DHT)



GFS: WHY ?

Co-design with apps

Fault tolerance

bunch of machines can fail

Workload pattern changed

- File size
Access size

> Before

- Not too many random writes

→ No caching

↳ Overhead / Scalability limits from caching?

Components with failures

Files are huge !

GFS: WHY ?

Applications are different

GFS: WORKLOAD ASSUMPTIONS

"Modest" number of large files

Two kinds of reads: Large Streaming and small random

Writes: Many large, sequential writes. No random

High bandwidth more important than low latency

GFS: DESIGN

- Single Master for metadata

- Chunkservers for storing data

- No POSIX API !
- No Caches!

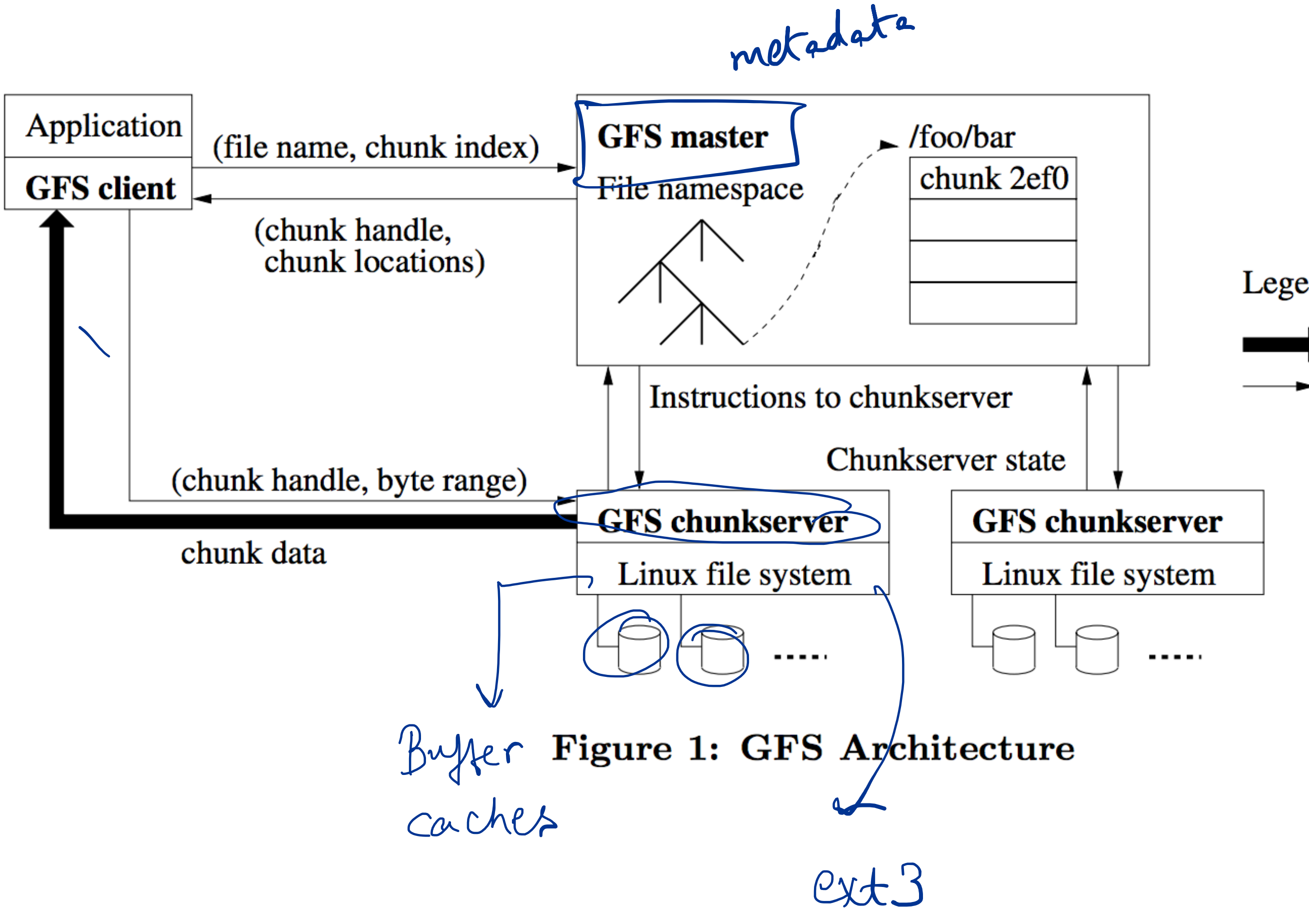


Figure 1: GFS Architecture

CHUNK SIZE TRADE-OFFS

64 MB

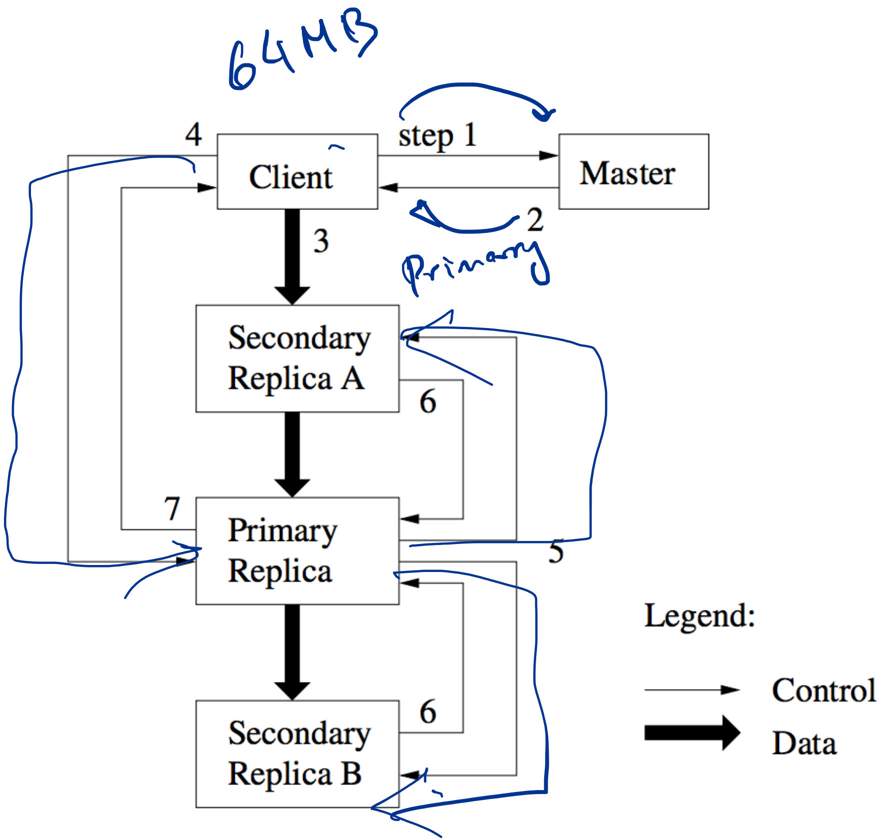
Client → Master → cache chunk location
amortize

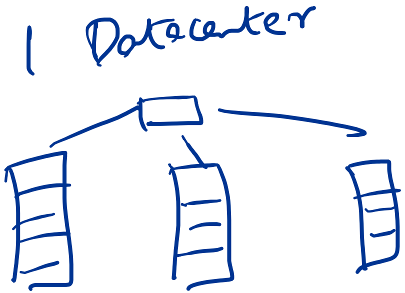
Client → Chunkserver → more client
same chunk

even if change is small → update chunk

Metadata → larger chunk size
⇒ lesser metadata

GFS: REPLICATION





- 3-way replication to handle faults
- Primary replica for each chunk
- Chain replication (consistency)

- Dataflow: Pipelining, network-aware

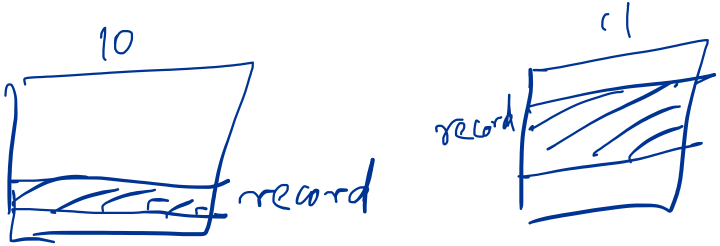
RECORD APPENDS

Write Client specifies the offset

Record Append GFS chooses offset

Consistency

- At-least once
- Atomic entire record as a sequence
- Application level

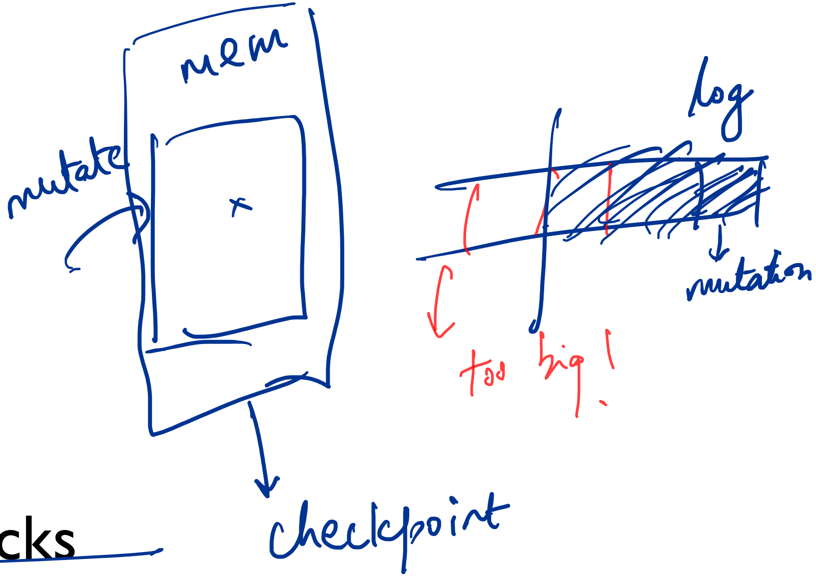


MASTER OPERATIONS

- No "directory" inode! Simplifies locking
- Replica placement considerations
    - Cross racks for FT
    - Balance writes → balancing empty chunks
    - Disk Utilization
- Implementing deletes
    - Lazy deletes using rename
    - Background process
    - recover, manage load, low latency

FAULT TOLERANCE

- Chunk replication with 3 replicas
- Master
    - Replication of log, checkpoint
    - Shadow master
- Data integrity using checksum blocks



Chunk locations → "accurate"

DISCUSSION

GFS SOCIAL NETWORK

You are building a new social networking application. The operations you will need to perform are

(a) add a new friend id for a given user

(b) generate a histogram of number of friends per user.

How will you do this using GFS as your storage system ?

Graph in-memory
→ checkpoint

→ Histogram checkpointed

1 file

− < a adds b >

records

"edge-list"

+ Append

− Access might require scanning whole file

+

File per user

u3
u4

u1 u2

+ Easy histogram

− Random writes delete

− Traversal

− Small files

100 Mbps ≈ 12.5 MB/s

GFS EVAL

≈ 70 MB/s Disk

List your takeaways from "Figure 3: Aggregate Throughputs”

60 - 100 MB/s

30

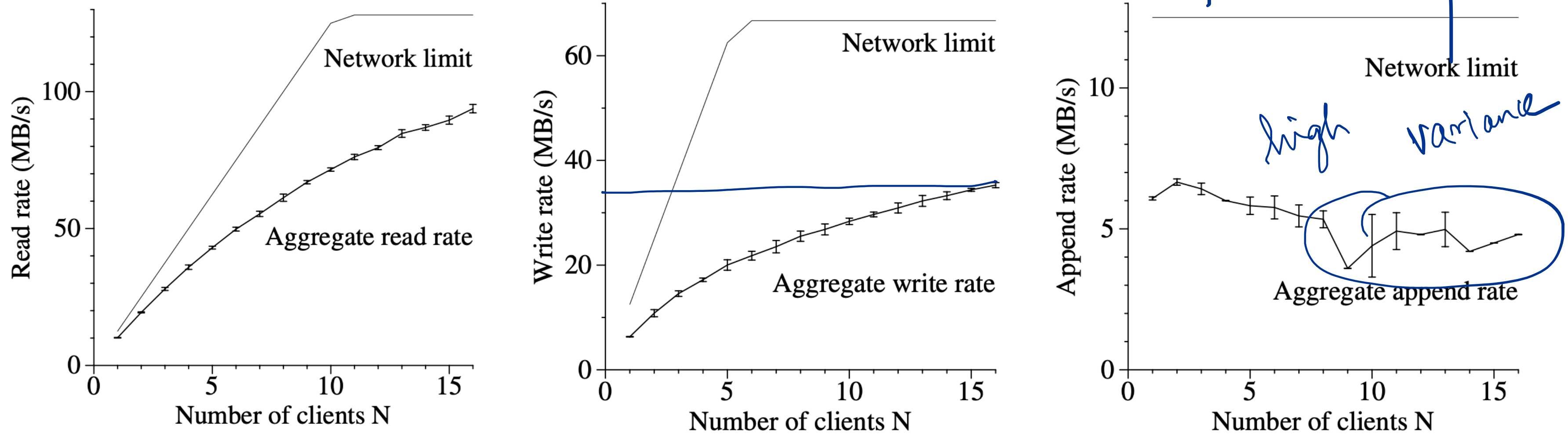
N files

1 file

5-6 MB/s

N clients

last chunk



(a) Reads (b) Writes (c) Record appends

GFS SCALE

The evaluation (Table 2) shows clusters with up to 180 TB of data. What part of the design would need to change if we instead had 180 PB of data?

64 MB chunks ⇒ Metadata explosion!

↓

Chunk size larger

WHAT HAPPENED NEXT

Cluster-Level Storage @ Google

How we use *Colossus* to improve storage efficiency

Denis Serenyi
Senior Staff Software Engineer
dserenyi@google.com

Keynote at PDSW-DISCS 2017: 2nd Joint International Workshop On Parallel Data Storage & Data Intensive Scalable Computing Systems

GFS EVOLUTION

Motivation:

- GFS Master
  - One machine not large enough for large FS
  - Single bottleneck for metadata operations (data path offloaded)
  - Fault tolerant, but not HA

- Lack of predictable performance
  - No guarantees of latency
  - (GFS problems: one slow chunkserver -> slow writes)

GFS EVOLUTION

GFS master replaced by Colossus

Metadata stored in BigTable

Recursive structure ? If Metadata is ~1/10000 the size of data

100 PB data → 10 TB metadata

10TB metadata → 1GB metametadata

1GB metametadata → 100KB meta...

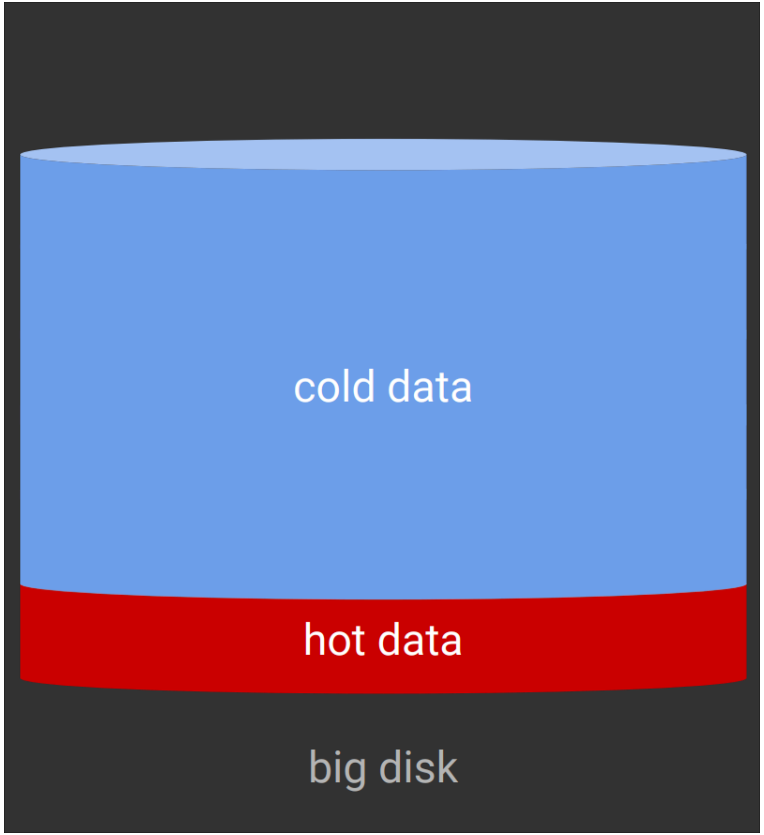
GFS EVOLUTION

Need for Efficient Storage

Rebalance old, cold data

Distributes newly written data evenly across disk

Manage both SSD and hard disks



HETEROGENEOUS STORAGE

amazon web services™ | S3

DynamoDB

F4: Facebook

redis

Blob stores

Key Value Stores

NEXT STEPS

- Assignment 1 out tonight!
- Next week: MapReduce, Spark