Can you see this? cool

GFS Google File System

=> Goals:
- -> [Scale] -> => for search, etc.
- -> where they focus on it, where they don't

=> Assumptions
- -> failure: common
- -> workload:
  - -> large, streaming I/O
  - -> writes: append (conc. append)
- -> well-defined semantics
- -> bandwidth, not

benefit: Apps

=> systems: H/W

no caching of data, don't worry about "small files"

latency

Interface:

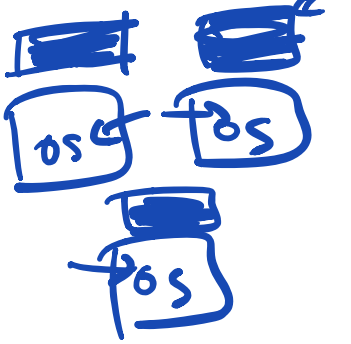
~ file system API
(but not identical)

e.g. append, snapshots, etc.

how to expose interface to clients?

→ (through OS : looks like local file system)

→ library : what?



why?

positives:

→ portability ← but not an issue here

→ easy to add new interfaces

→ easy to deploy

→ easy to develop?

negative:

→ doesn't integrate w/ existing apps

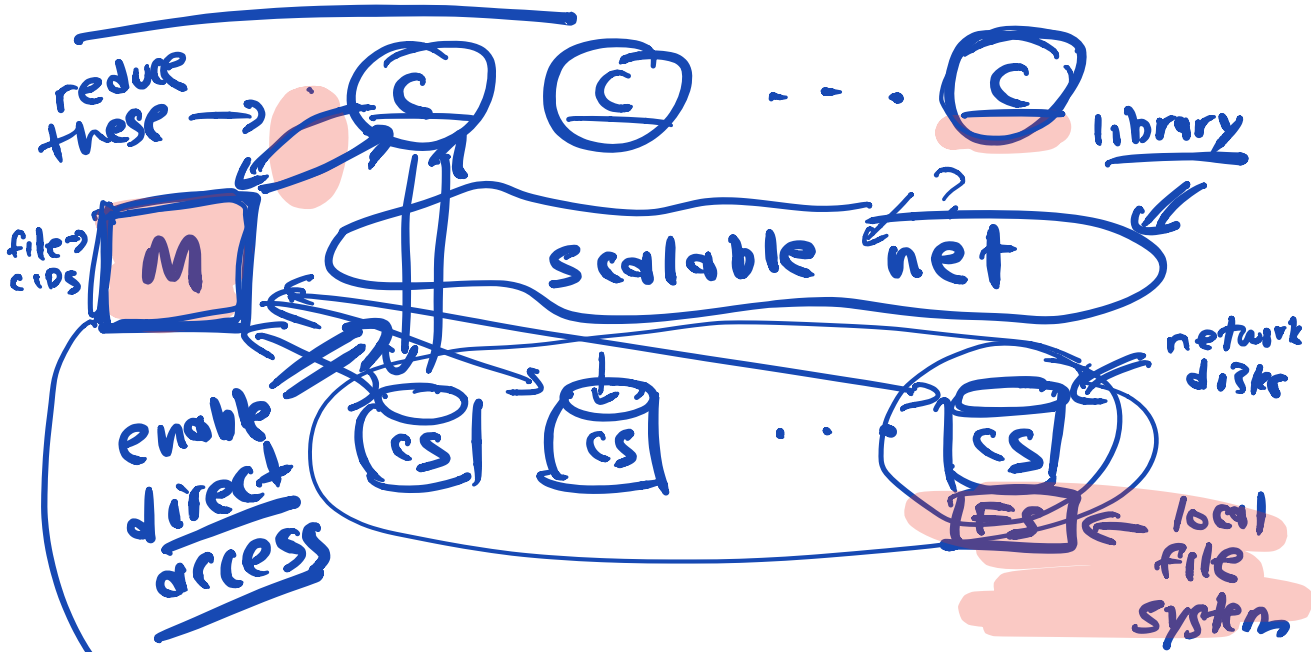
classic rm:

gfs:

gfsrm gfsls

→ poor integration w/ OS features (caching)

Architecture (high level)



→ centralized: surprise

why good?

→ simpler ⇐ (no replication)

→ centralized knowledge
→ information
makes many decisions easier to make

file:
just made up of chunks → 64MB
→ 64-bit unique ID (immutable)

Chunk size : 64MB (large)

→ positives:
less metadata → and client
→ reduces master memory load
→ less contact w/ master

→ negatives:
usual: internal fragmentation
here: OK because
→ most files: large
→ using local file for chunk:

lazily allocated
(4KB block size)

Metadata @ Master:

All in memory — lots of persistent

Types:

- file/chunk namespaces (64 bit UID)
- map: file → chunk ID
- (map: (CID → servers)) kept @ CS

comm ⟺ master
heartbeats

writes (to metadata)

⇒ op log → replicated file

(periodically checkpointed)

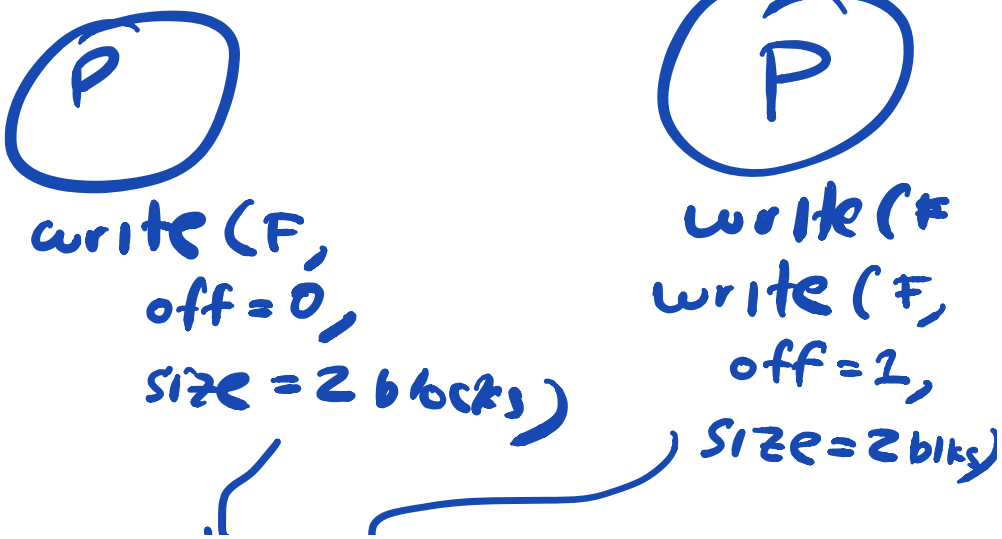
Consistency Model

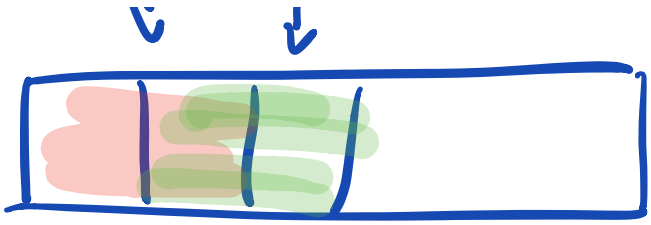
Files: replica

consistent:
all clients see same data
(regardless of replica)

[**defined:** (higher form of consistency)
consistent + clients
see result of
client mutation in
its entirety]

e.g. local file system





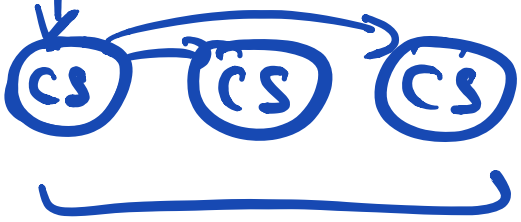
key: apps can deal w/ imperfection

easy ↓ Reads (Writes) ← hard

write : primary/backup

later: append

⇒



replication: 3

3-phase flow

1) C → M ← relevant meta, lease

Data Flow

2) C → [CS1, ..., CS3]

primary

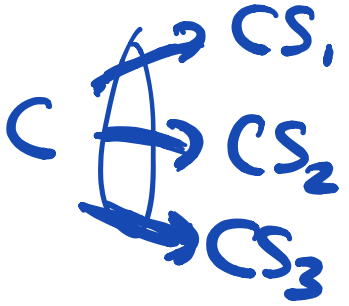
3) ordering :

CS primary : decides order

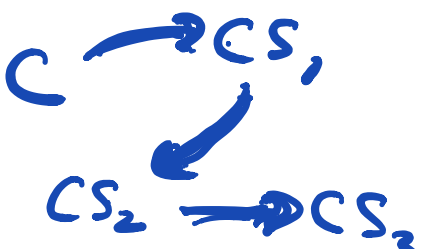
C → CS primary → CS sec.

ack ← CS sec

Data Flow: pipeline

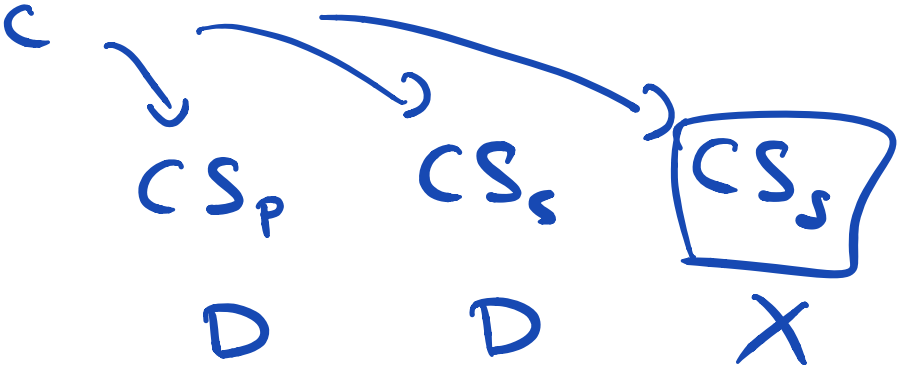


typical



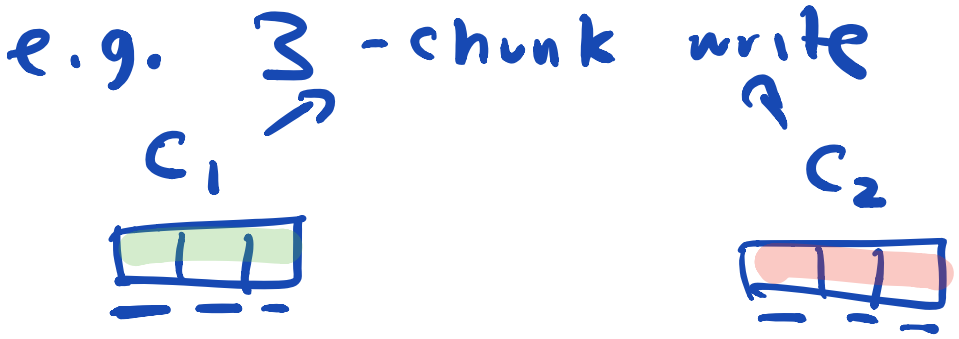
actual

writes: can fail



consistent, not defined

e.g. 3-chunk write



Append:

similar flow

key diffs:

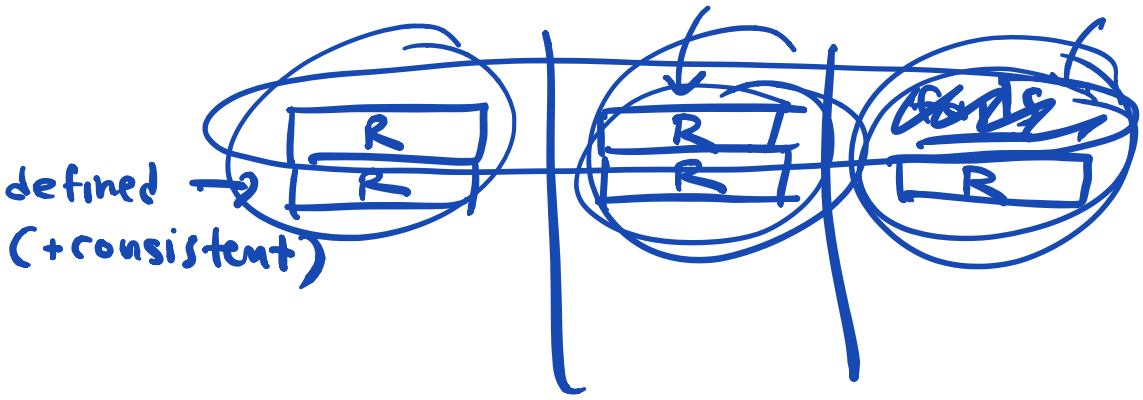
→ make sure record append is w/in 1 chunk

⇒ use padding

CS:

→ primary: pick offset

Problem: failure ⇒ client retry : append



CS_1 CS_2 CS_3