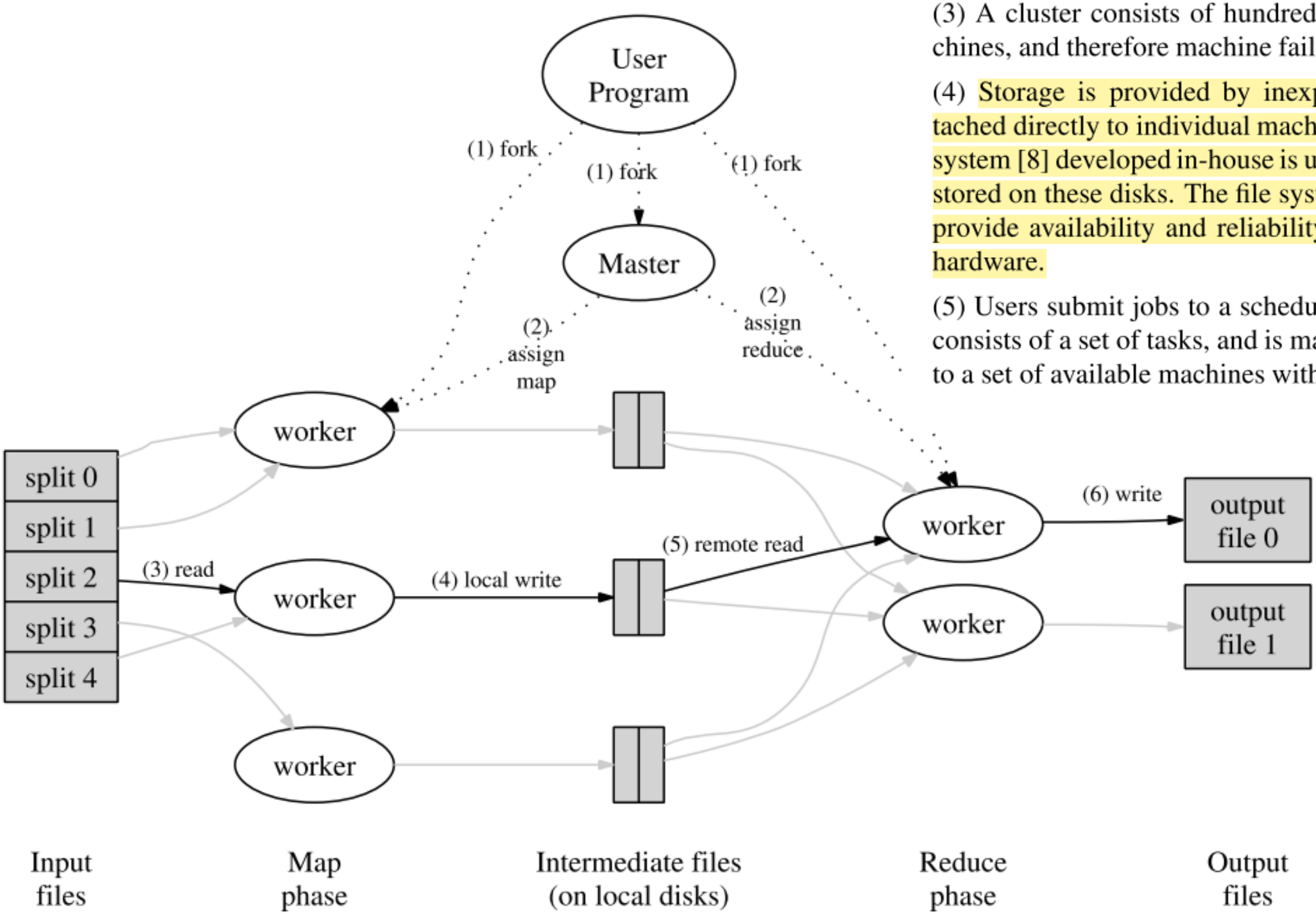
Distributed Systems

Spring Semester 2020

Lecture 4: Google File System

John Liagouris
liagos@bu.edu

MapReduce



(1) Machines are typically dual-processor x86 processors running Linux, with 2-4 GB of memory per machine.

(2) Commodity networking hardware is used – typically either 100 megabits/second or 1 gigabit/second at the machine level, but averaging considerably less in overall bisection bandwidth.

(3) A cluster consists of hundreds or thousands of machines, and therefore machine failures are common.

(4) Storage is provided by inexpensive IDE disks attached directly to individual machines. A distributed file system [8] developed in-house is used to manage the data stored on these disks. The file system uses replication to provide availability and reliability on top of unreliable hardware.

(5) Users submit jobs to a scheduling system. Each job consists of a set of tasks, and is mapped by the scheduler to a set of available machines within a cluster.

Google File System

[8] The Google File System; Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung SOSP 2003

Why are we reading this paper?

All major topics of this course show up in this paper: **scalability**, **fault-tolerance**, and **data consistency**

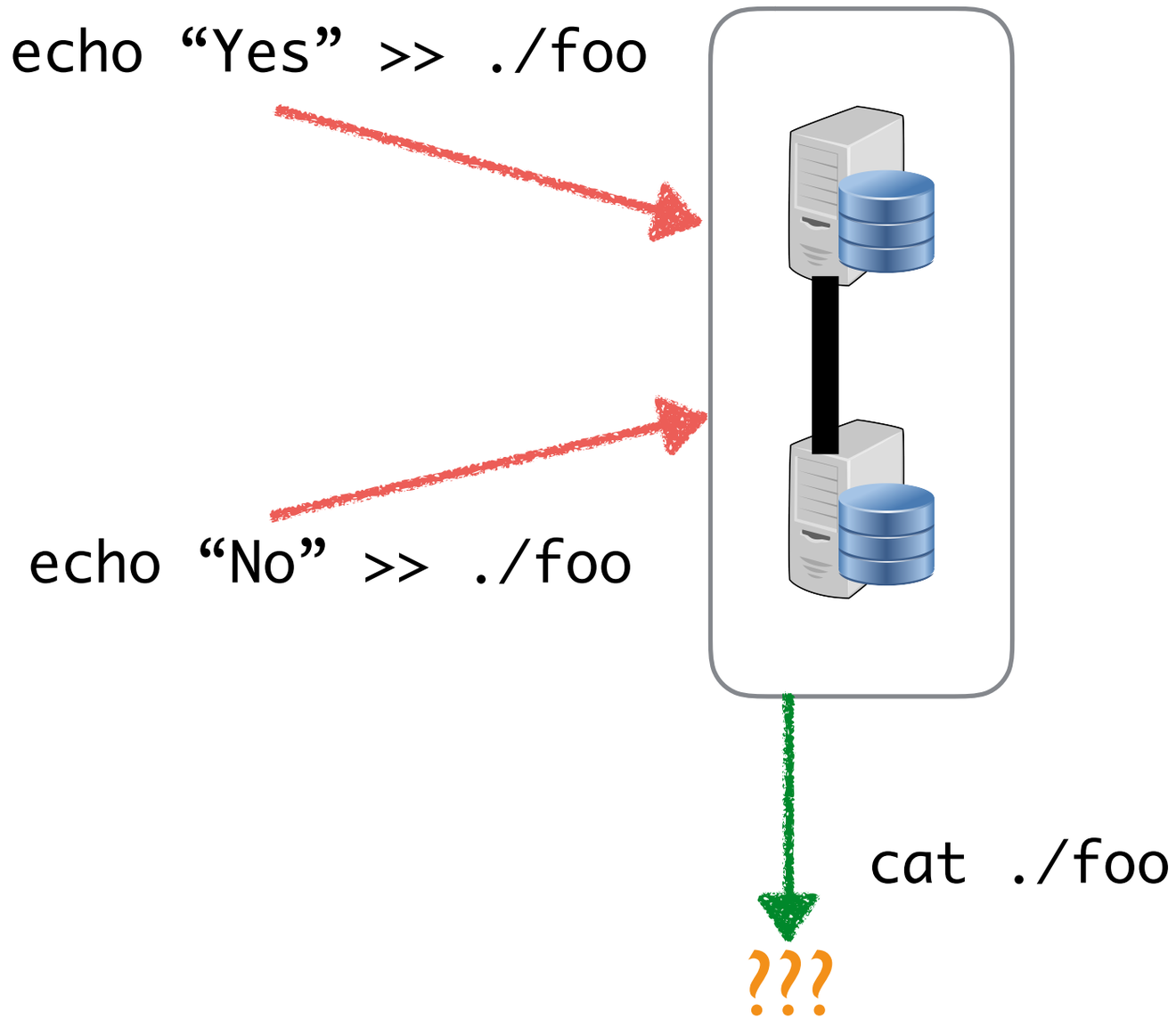
What is consistency?

A ***Correctness*** Condition

What is consistency?

Important when data is replicated and concurrently accessed by applications

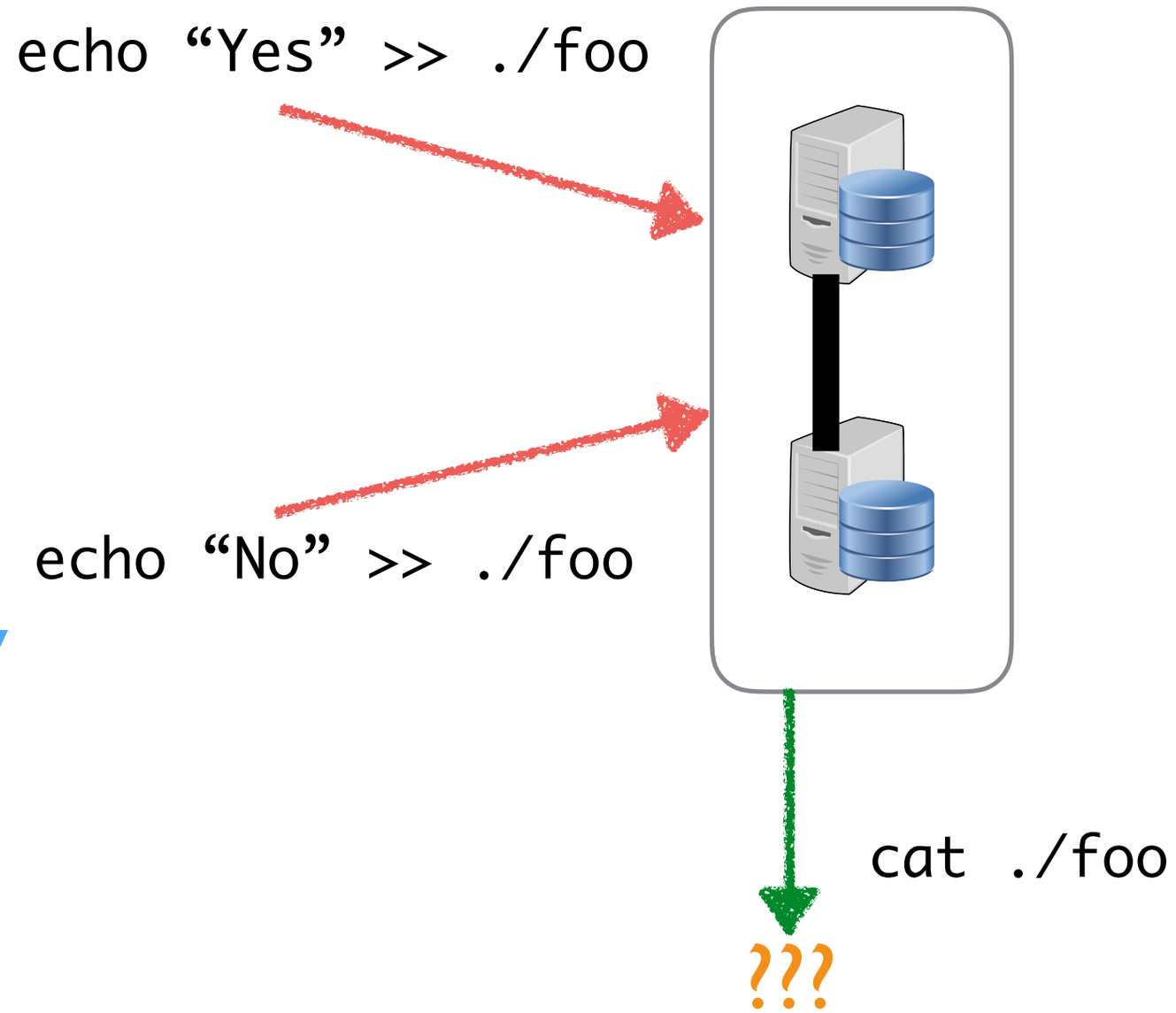
- If a **write** is performed, what value then will later **read** observe?
- What if two writes are performed concurrently?
- What if the reader is different from the writers?



Data’s **consistency model** defines **what is** correct

What is consistency?

- **Strong consistency**
  - read **always** returns the value of the *most recent* write
- **Weak consistency**
  - reads *may* return "stale" data, i.e., **not** the value of the *most recent* write



In a weak consistency model, *inconsistency* is… **OK**!

Long History of Consistency

Decades of development from architecture, operating systems, database, and distributed system communities:

- Concurrent processors with private caches accessing a shared memory
- Concurrent clients accessing a file system
- Concurrent transactions on distributed database

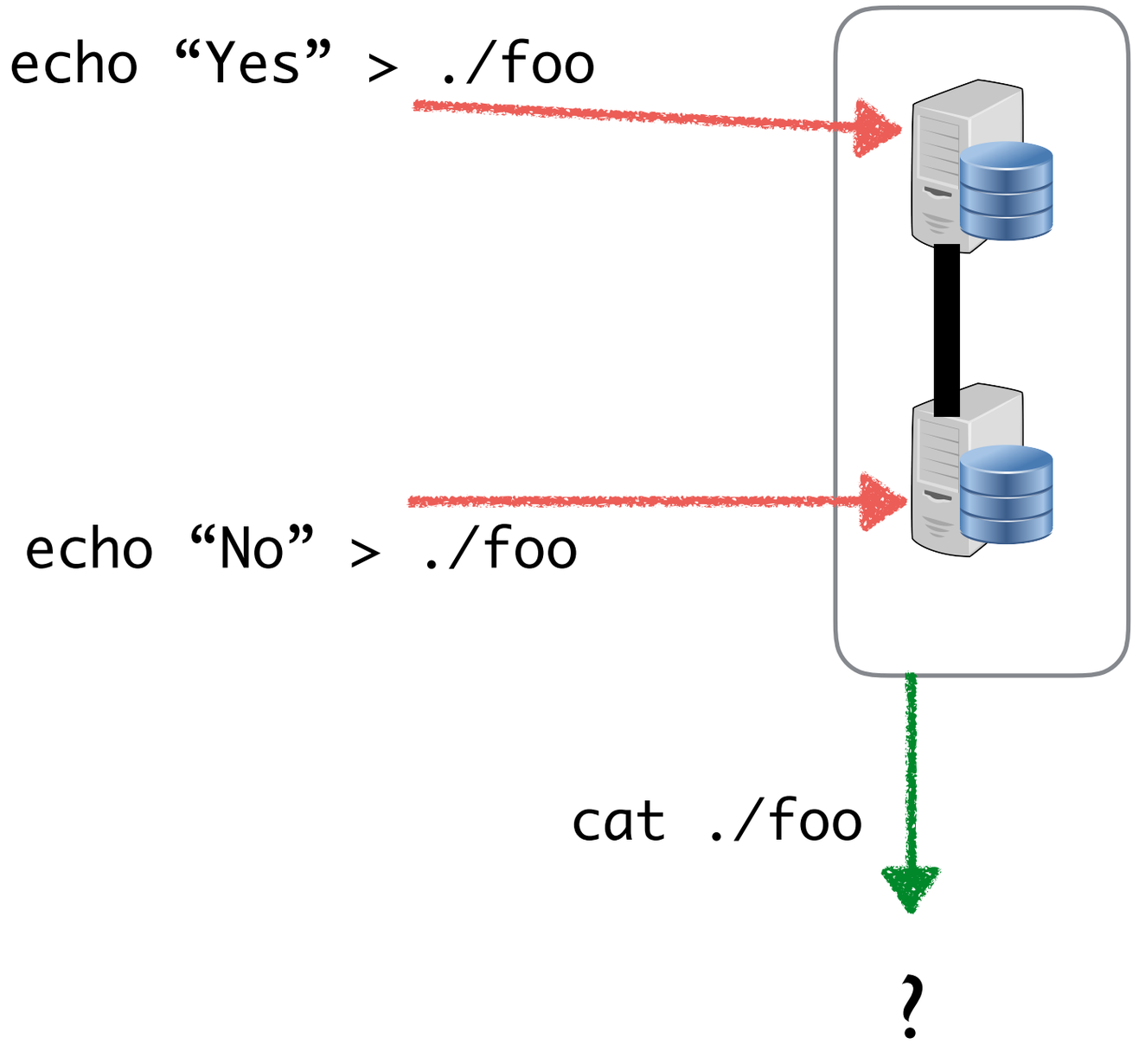
Today may be your first peak at consistency…
but it'll show up in <u>every other paper we read this term!</u>

Consistency Today?

- Important data maybe replicated to prevent it from being lost due to a failure (**durability)**
- Important services maybe replicated to handle increasing demand, or guarantee **availability** in the case of a failure
- Any mutations of a replicated value has the potential of introducing *inconsistencies*!

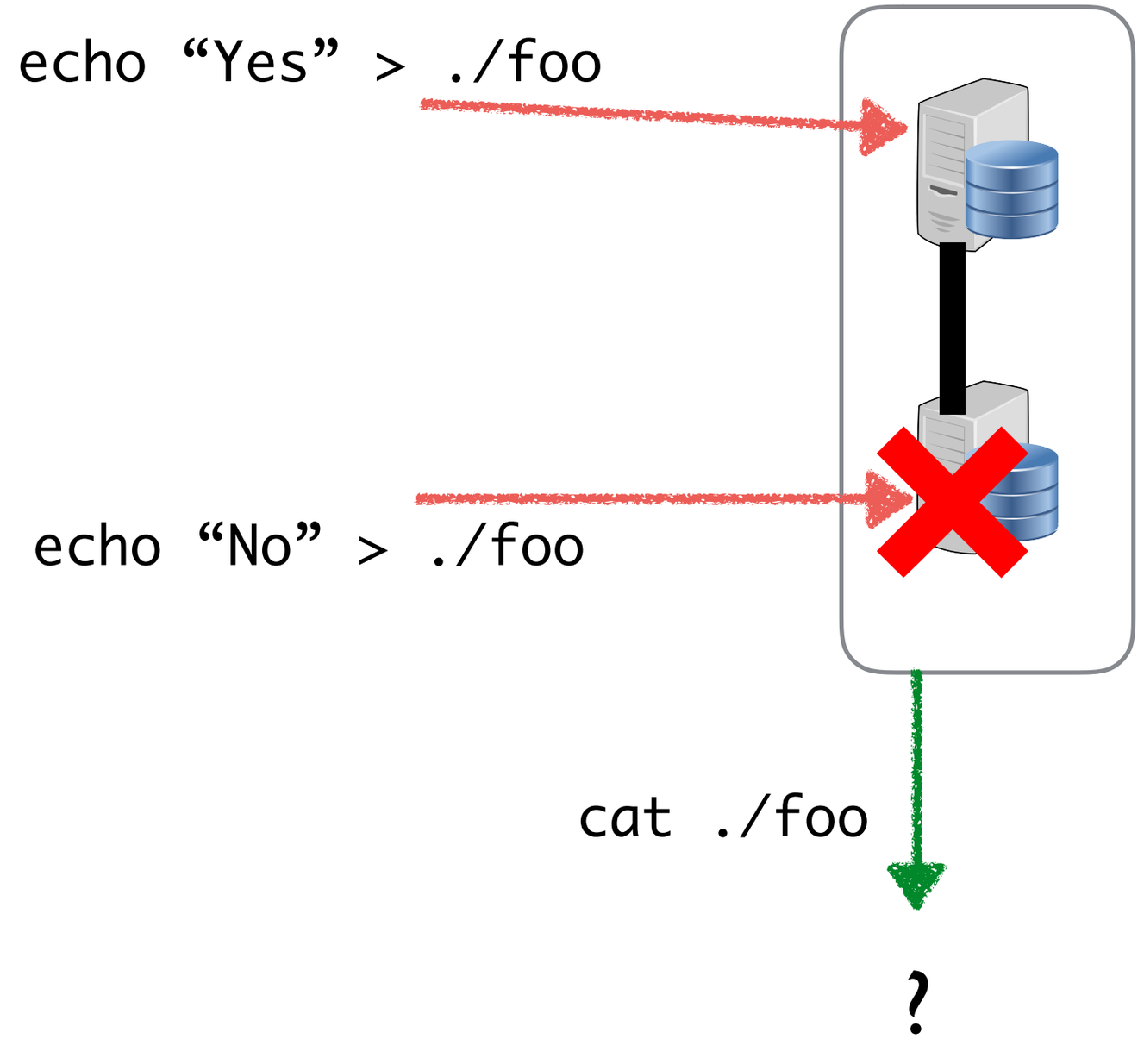
Sources of Inconsistency

CONCURRENCY!



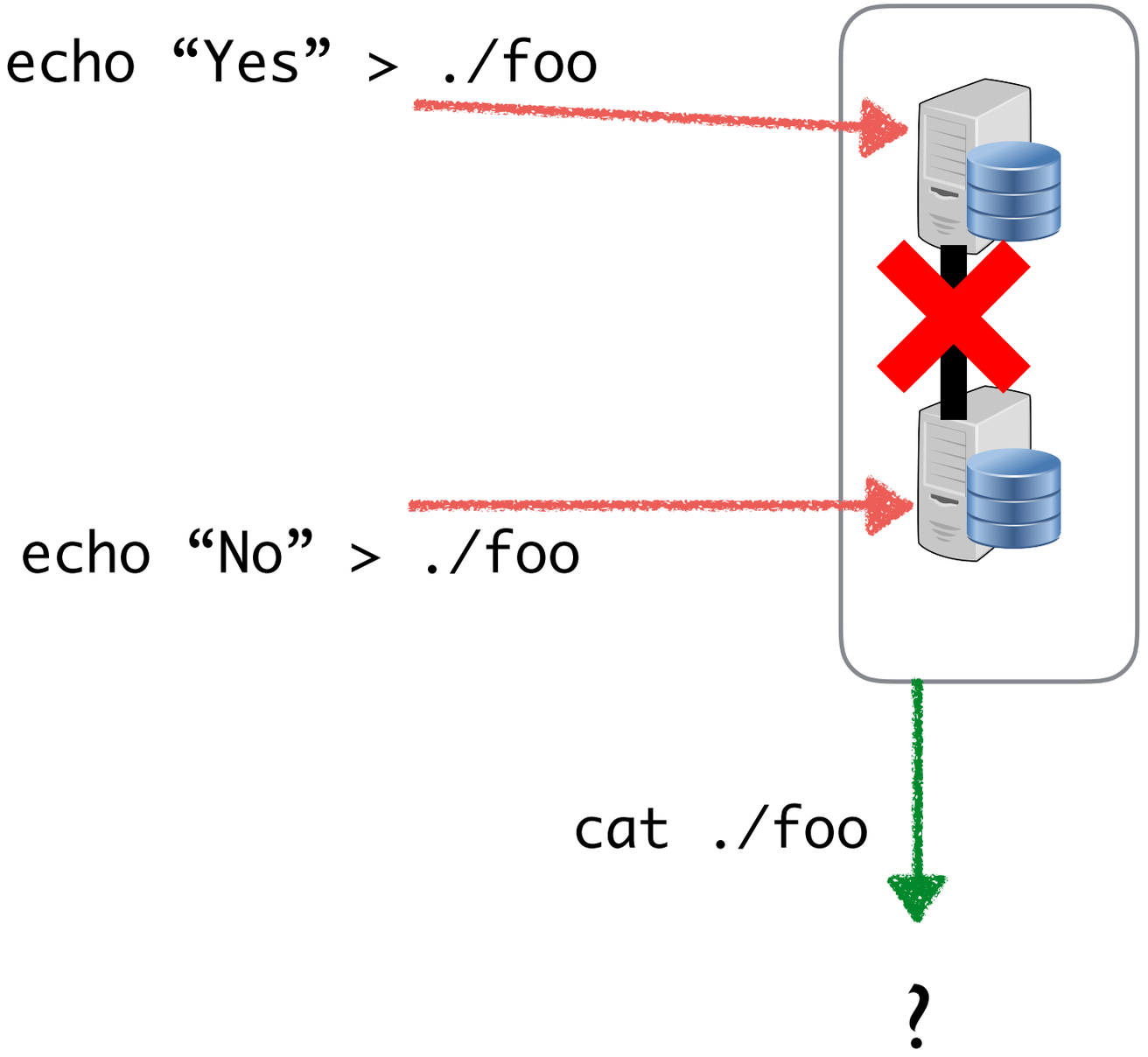
Sources of Inconsistency

MACHINE FAILURE!



Sources of Inconsistency

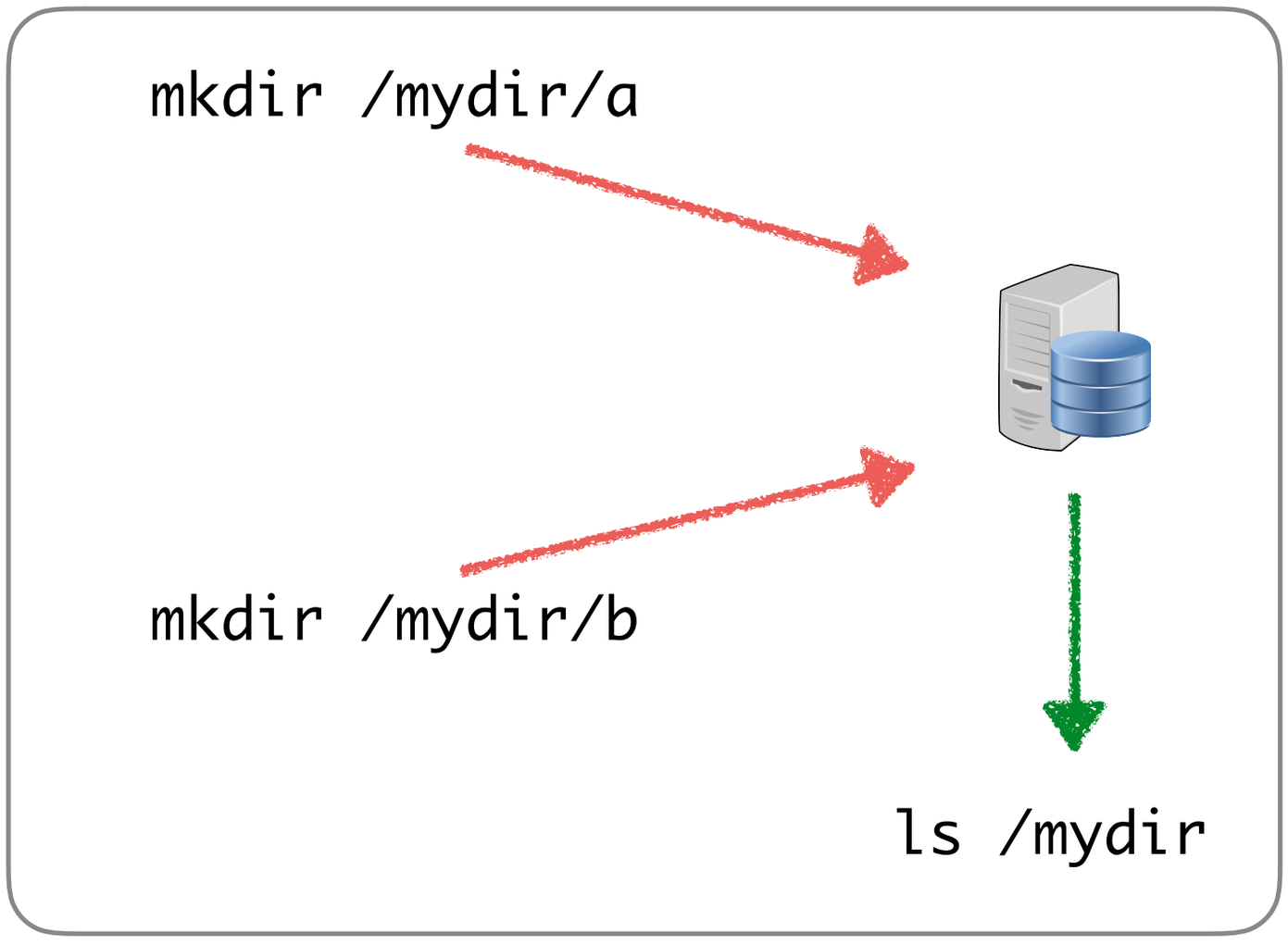
NETWORK FAILURE!



“Ideal” Correctness Guarantees

Replicated system **behaves** like a non-replicated system

- “Single copy” ***serialized*** access
- After a write, <u>all reads will always</u> see update value
- Concurrent writes to **same file** independently done in a *well defined order*



Achieving the“ideal” can often be… *unideal*

- Protocols require lots of communication!
  - Network communication is not free
  - Networks are unreliable!
- Underlying program is **BLOCKED** until the *correctness of the operation is fully assured*
  - Performance suffers, quality decreases. Sad!

An Inconvenient Truth

- Strong consistency is **GREAT** for the design of distributed operations

- Strong consistency is **TERRIBLE** for the performance of distributed operations

An Inconvenient Truth

- Weak consistency is **TERRIBLE** for the design of distributed operations

- Weak consistency is **GREAT** for the performance of distributed operations

In reality, many different "consistency models" exist employing various degrees of correctness guarantees

GFS not ideal by design

Paper illustrates a design struggle between:

- data correctness
- fault-tolerance
- aggregate performance / scalability
- simplicity of design, usability for developers

GFS goal

- Shared filesystem for Google developers
- Hundreds or thousands of (commodity) physical machines — i.e., CHEAP, disposable
- To enable storing massive data sets, sequential read and append-heavy workloads

The Files

- Multi-terabyte data sets
- Most of the files are large
- Authors suggest 1Million files x 100 MB = 100TB (in 2003)
- Majority of writes are append only

What does it store?

- Well.. authors don't actually say
- My best guesses (for 2003)…
  - System logs
  - Search indexes & databases
  - All cached HTML/images files the web
  - MapReduce-like sources and merged output

The Challenge

With enough scale, exceptions turn into expectations!

Assume a individual machine "fails" once per year

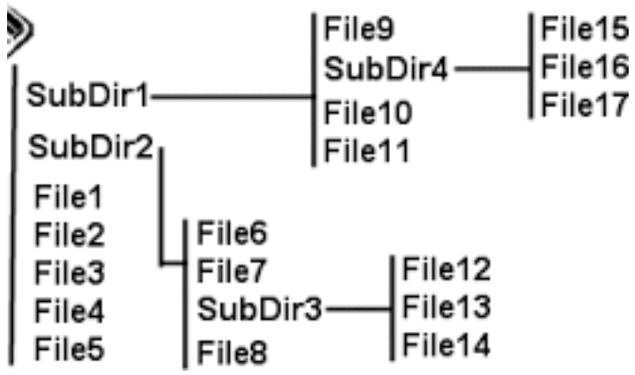
- 1000 machines, ~3 failures per-day
- 10,000 machines, ~1 failures per-hour

High-performance workloads

- Many concurrent readers and writers
- Huge sequential reads and writes
- Streaming data / use network efficiently

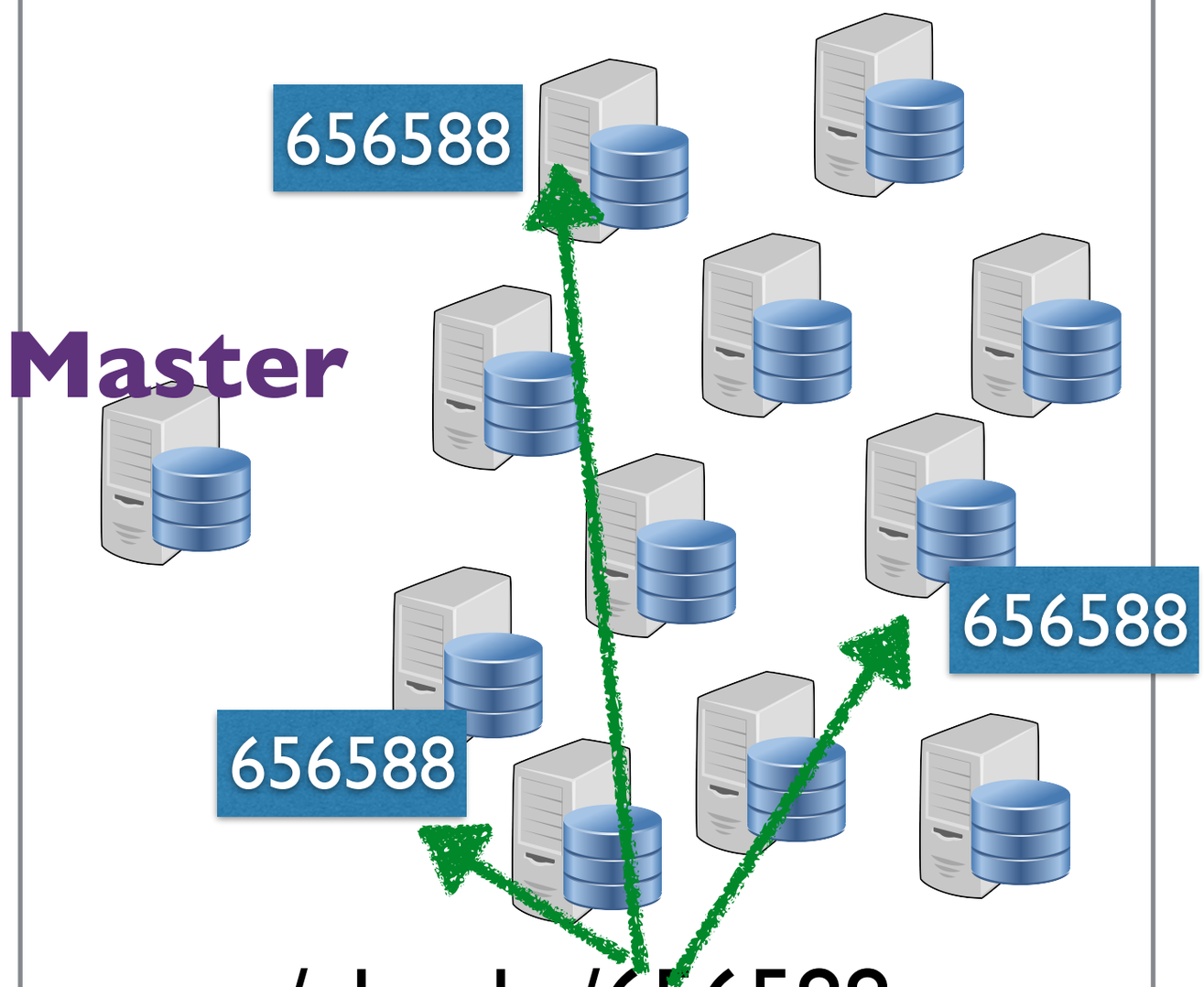
GFS Design

clients



Directories, files, names, open/read/write

100's of Linux *Chunkservers*

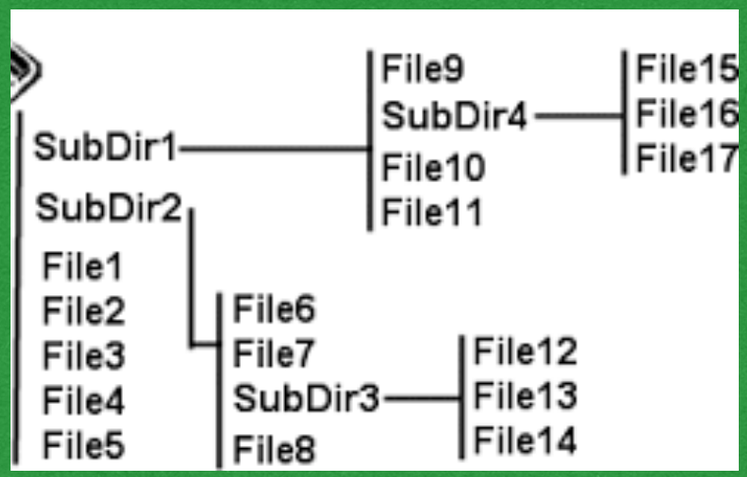


/chucks/656588
64MB chunks(3xreplicas)

GFS Design

Master Server

Directory Tree file:Chunks



/File1
656588
878788
999456
174520

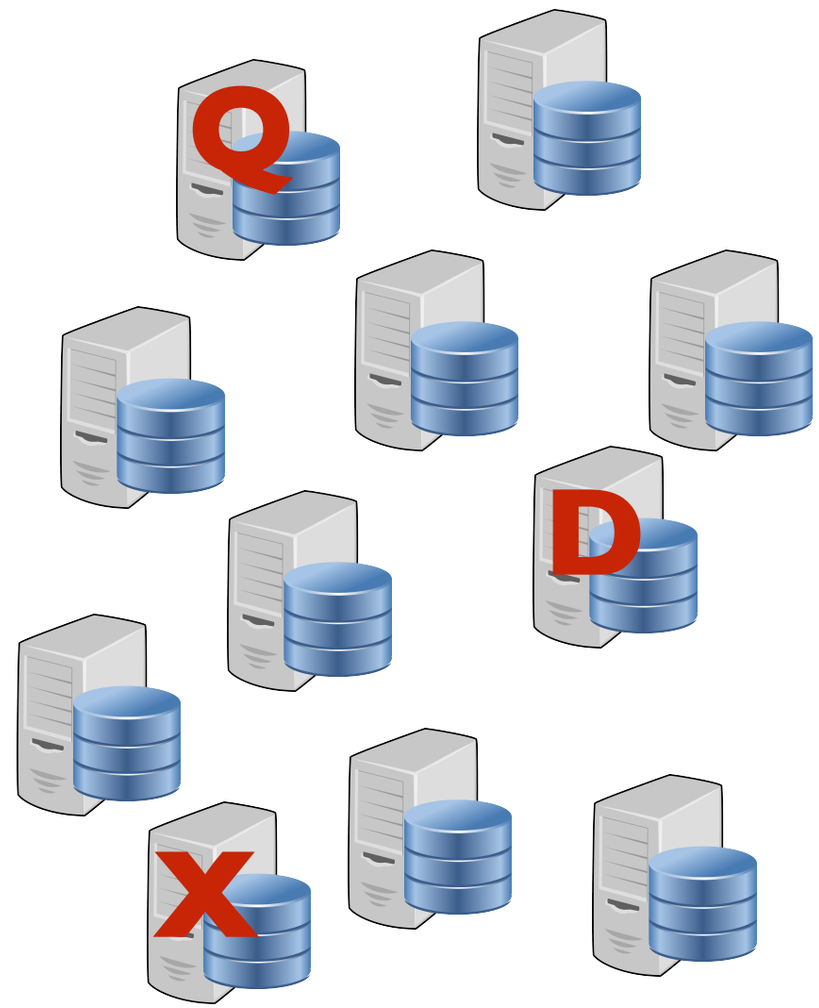
Master

Persistent

Chunk : Servers

Chuck	Servers
656588	Q,D,X
876768	P,V,R
768774	M,N,V
…	

Chunkservers

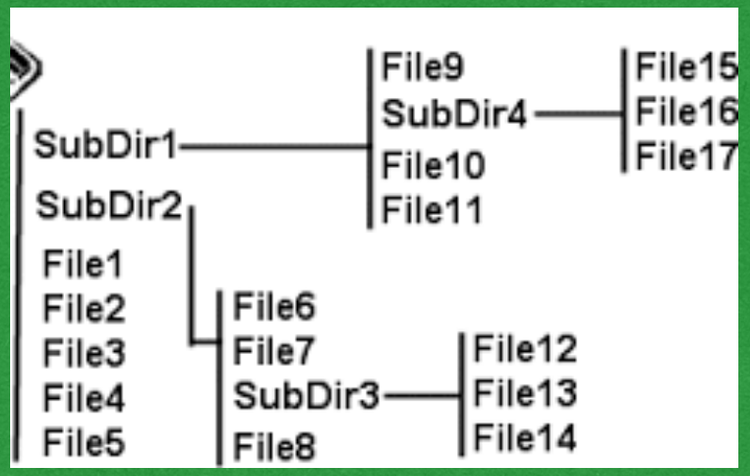


Basic Reads

read("/File1", 100, …)

Master

Directory Tree file:Chunks



/File1
656588
878788
999456
174520

Persistent

Chunk : Servers

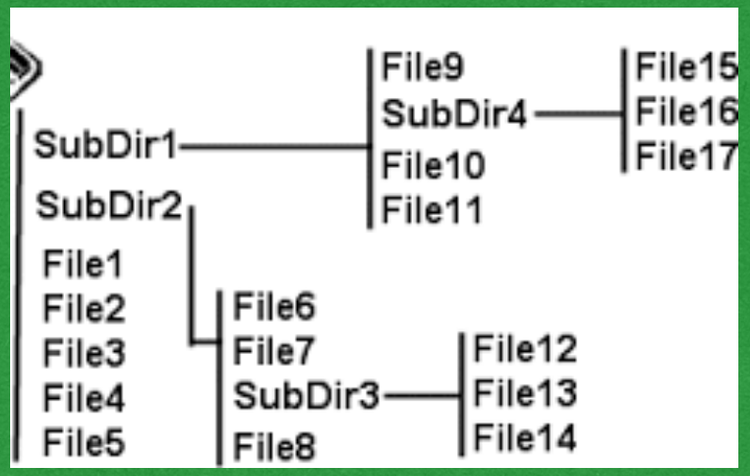
Chuck	Servers
656588	Q,D,X
876768	P,V,R
768774	M,N,V
…	

Basic Reads

656588: Q,D,X

Master

Directory Tree file:Chunks



/File1
656588
878788
999456
174520

Persistent

Chunk : Servers

Chuck	Servers
656588	Q,D,X
876768	P,V,R
768774	M,N,V
…	

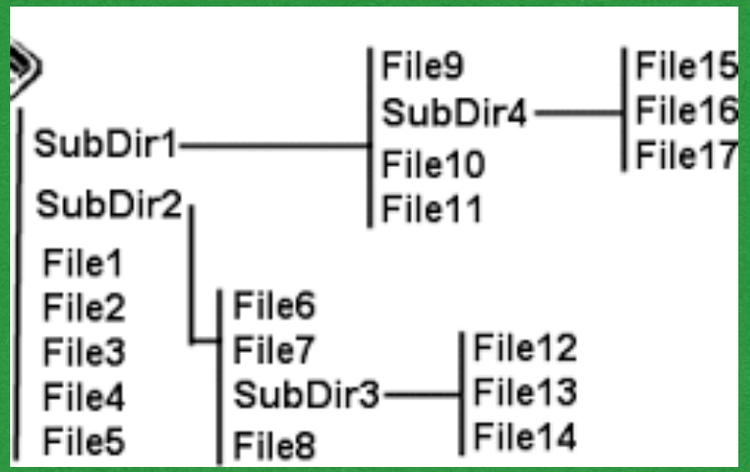
Basic Reads

Chunk to Server $

Chuck	Servers
656588	Q,D,X

Master

Directory Tree



file:Chunks

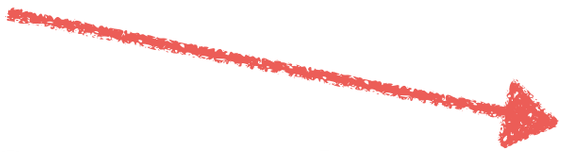
/File1
656588
878788
999456
174520

Persistent

Chunk : Servers

Chuck	Servers
656588	Q,D,X
876768	P,V,R
768774	M,N,V
…	

Basic Reads



D

Chunk to Server $

Chuck	Servers
656588	Q,D,X

656588

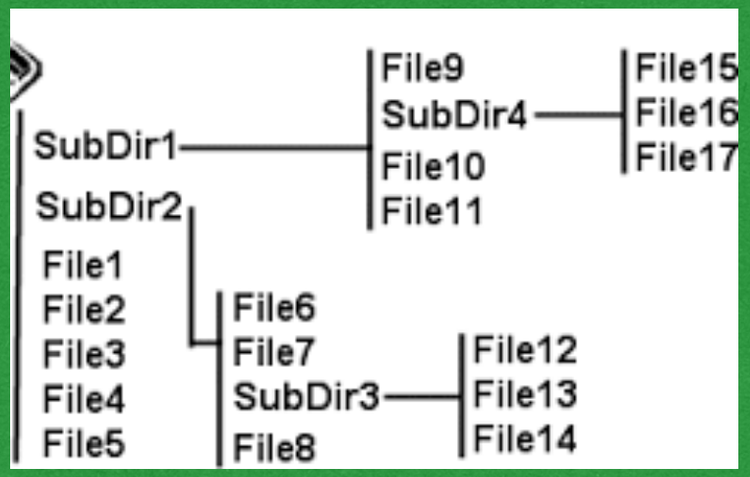
Asks “closest” server for the data

Basic Writes

Write("/File1", 100, …)

Master

Directory Tree file:Chunks



/File1
656588
878788
999456
174520

Persistent

Chunk : Servers

Chuck	Servers
656588	Q,D,X
876768	P,V,R
768774	M,N,V
…	

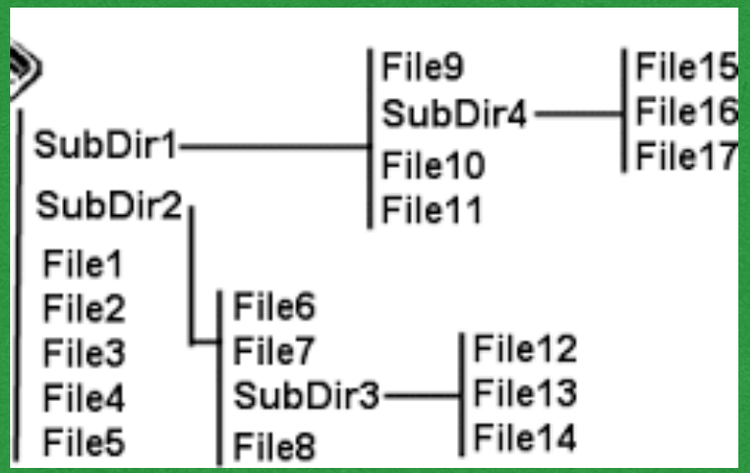
Basic Writes

656588: **Q***,D,X

Q is the data's ***primary***

Master

Directory Tree file:Chunks



/File1
656588
878788
999456
174520

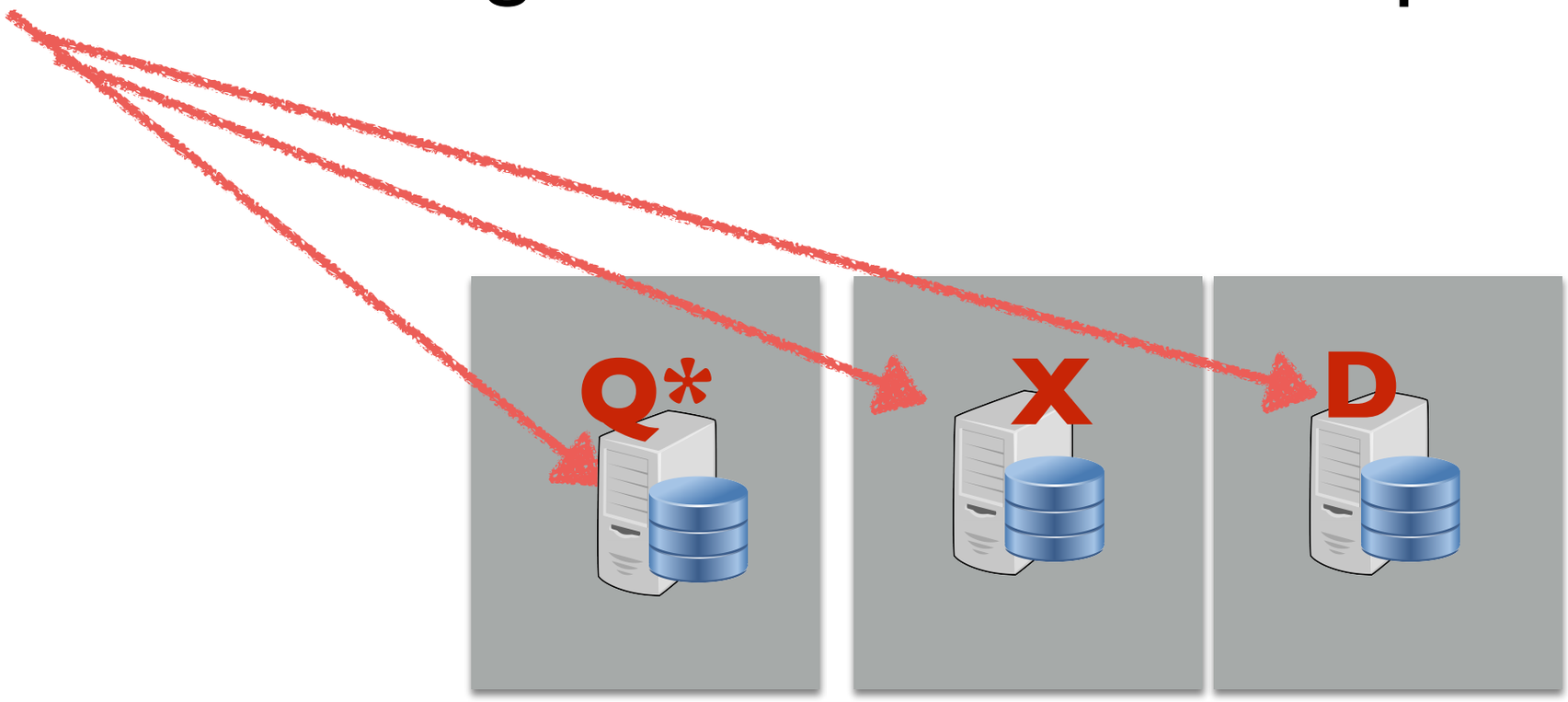
Persistent

Chunk : Servers

Chuck	Servers
656588	Q,D,X
876768	P,V,R
768774	M,N,V
…	

Basic Writes

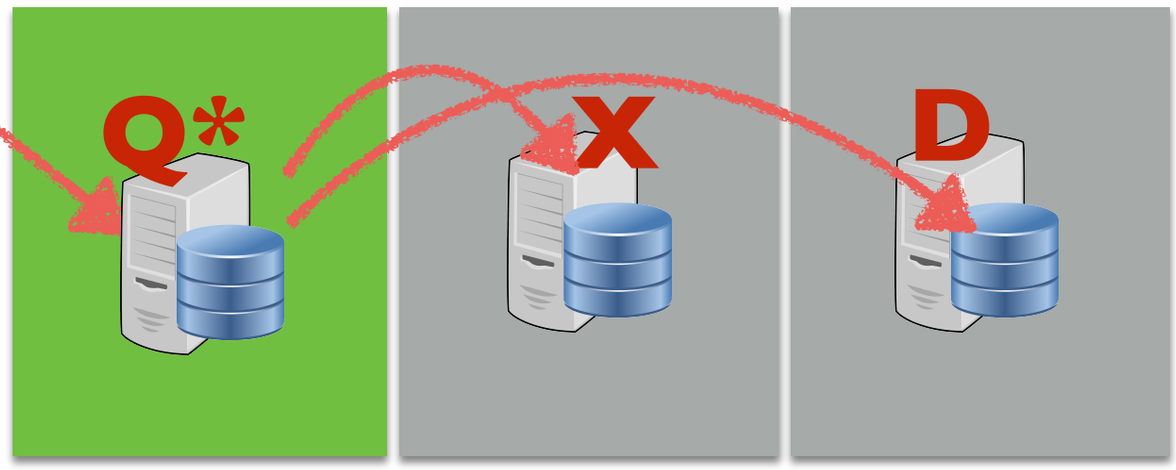
Client "stages" the write on all replicas



But, the write is not complete (no data *written*)

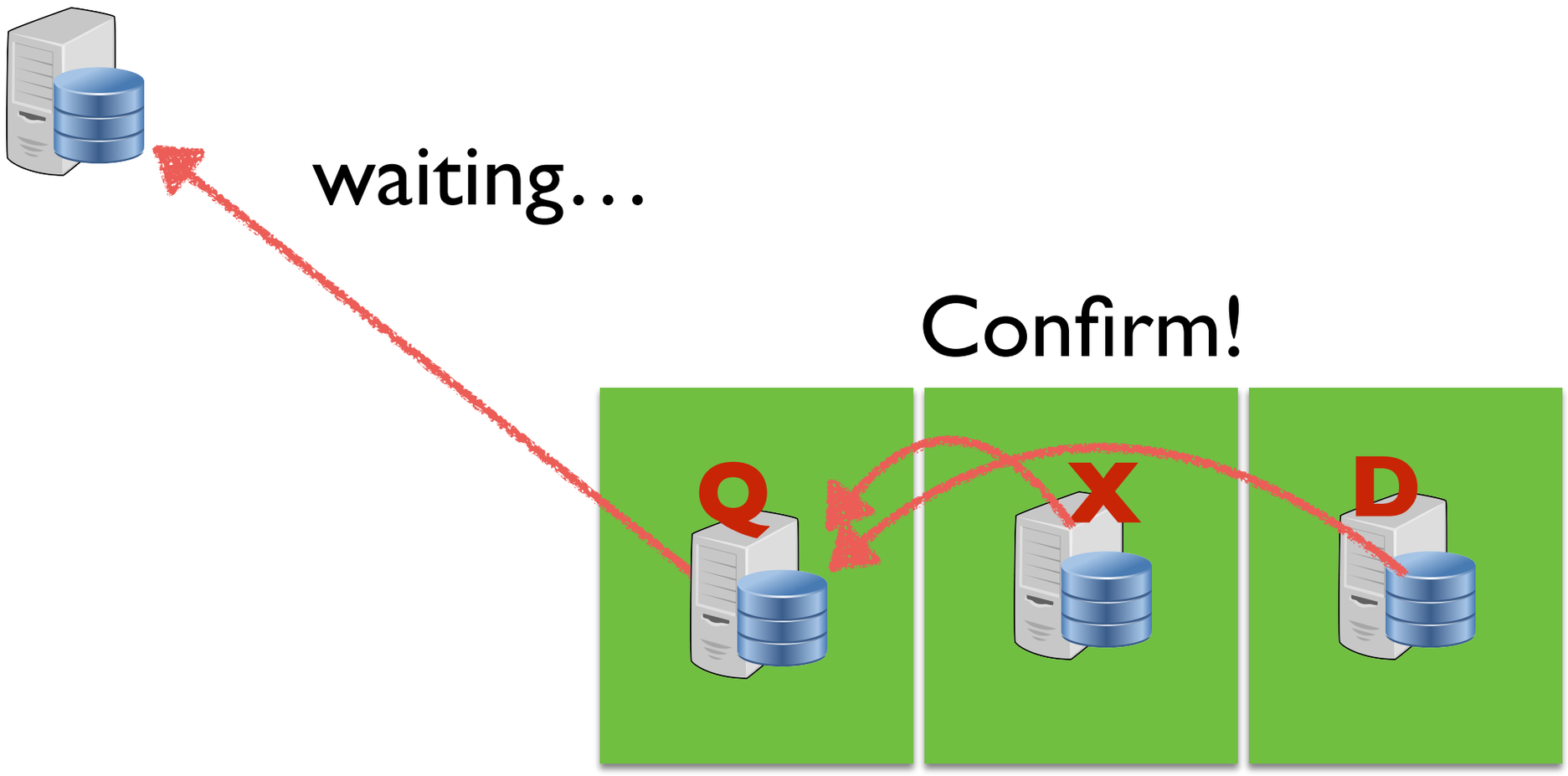
Basic Writes

Confirm write on primary



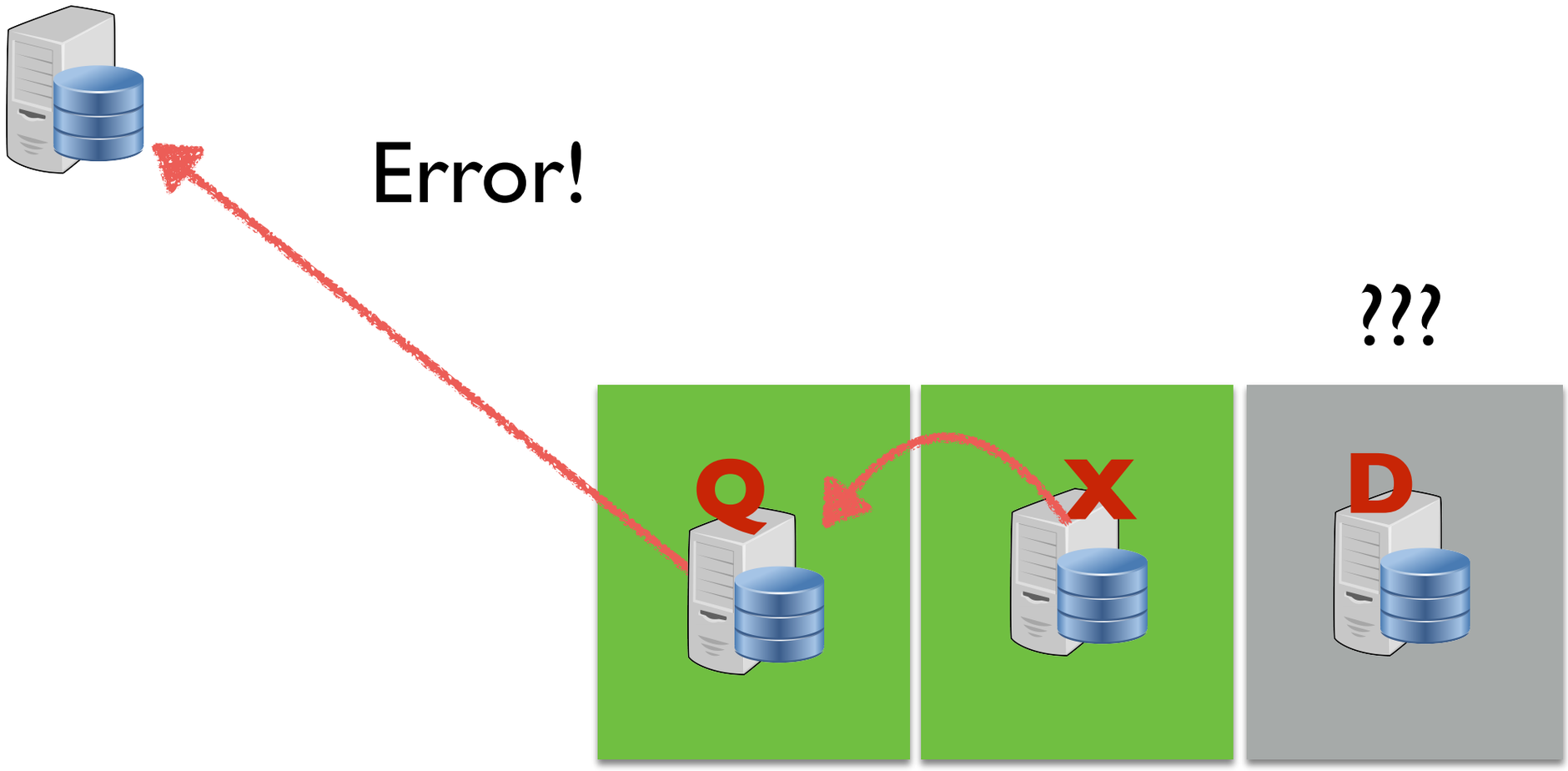
Primary job is to accept the request and chooses order of application (with concurrent requests) and applies change

Basic Writes



Primary confirms when the write is completed across all three replicas.

Basic Writes

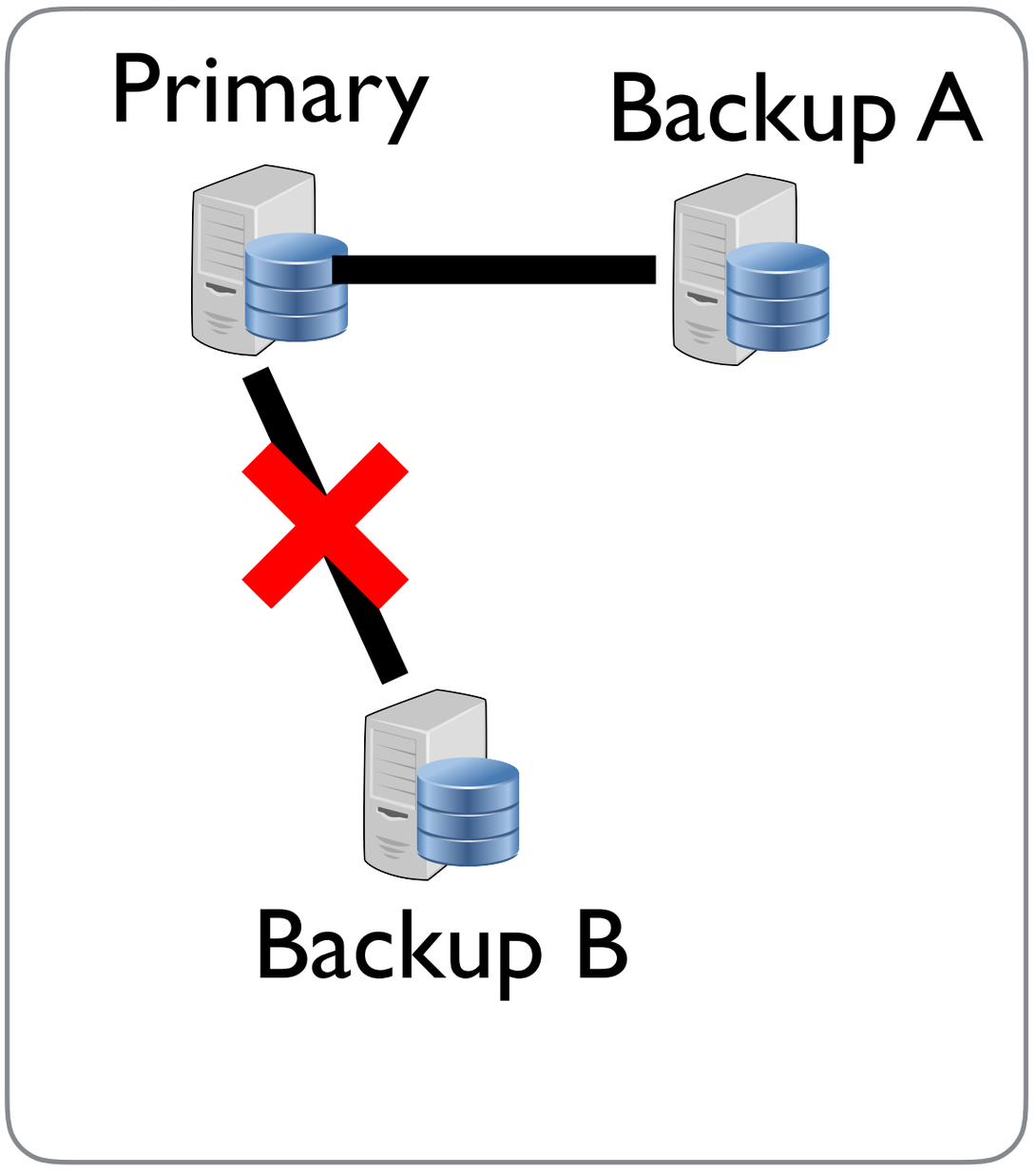


Else, error is reported to client. Client retries

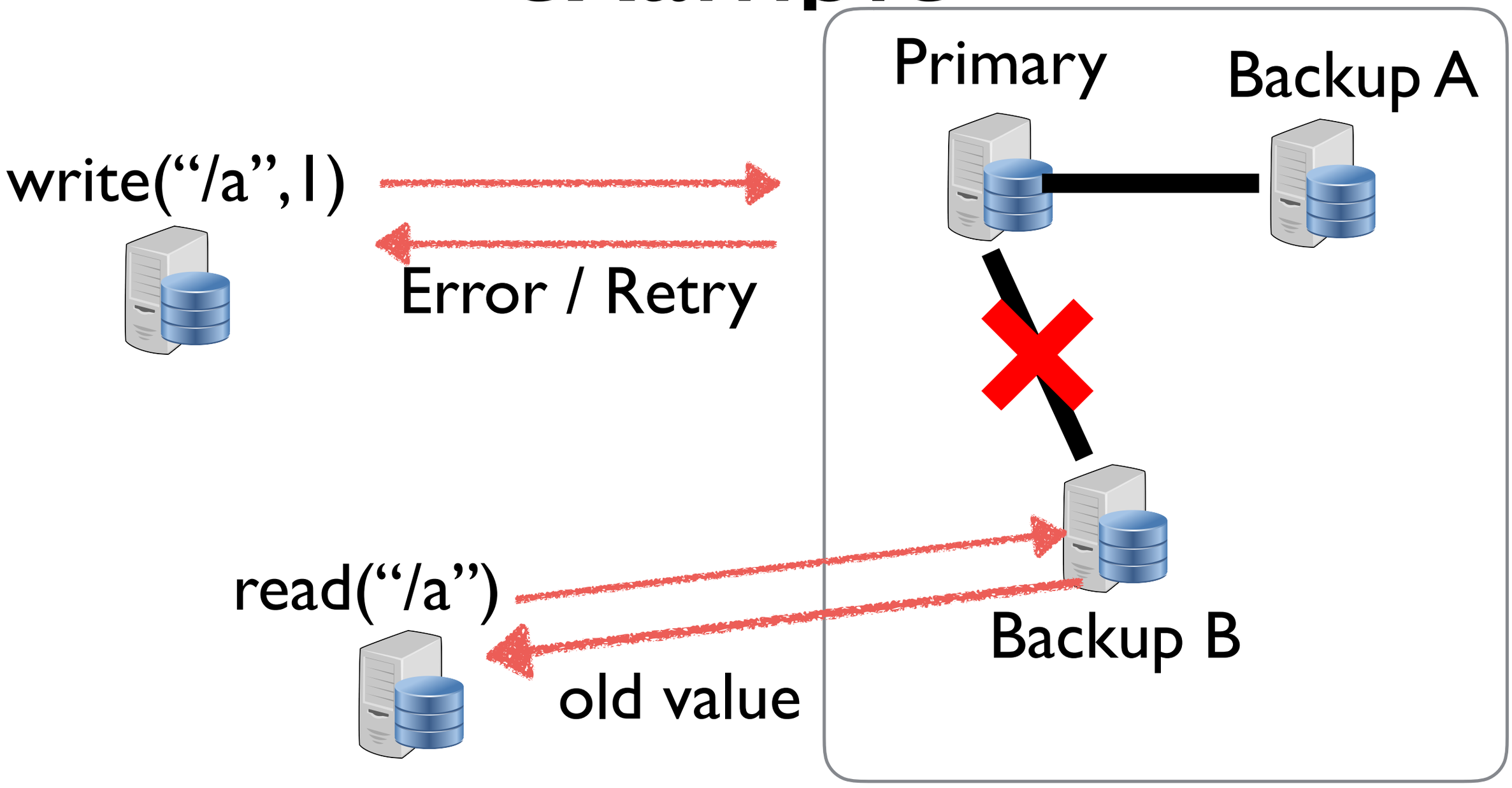
Network Partition GFS example

write("/a",1)

Error / Retry



Network Partition GFS example



Concurrent Writes/Appends

- Concurrent writes to same region of file
  - Write on a region result in a mix of those writes
    - Few apps do this — but same semantics on Linux
- Special support for ***"record appends"***
  - Guarantees **atomic** *at-least-once* appends
  - Primary coordinates — tell replicas *exact* offset
    - fail to client and retry — duplicates can arise

GFS Fault Tolerance

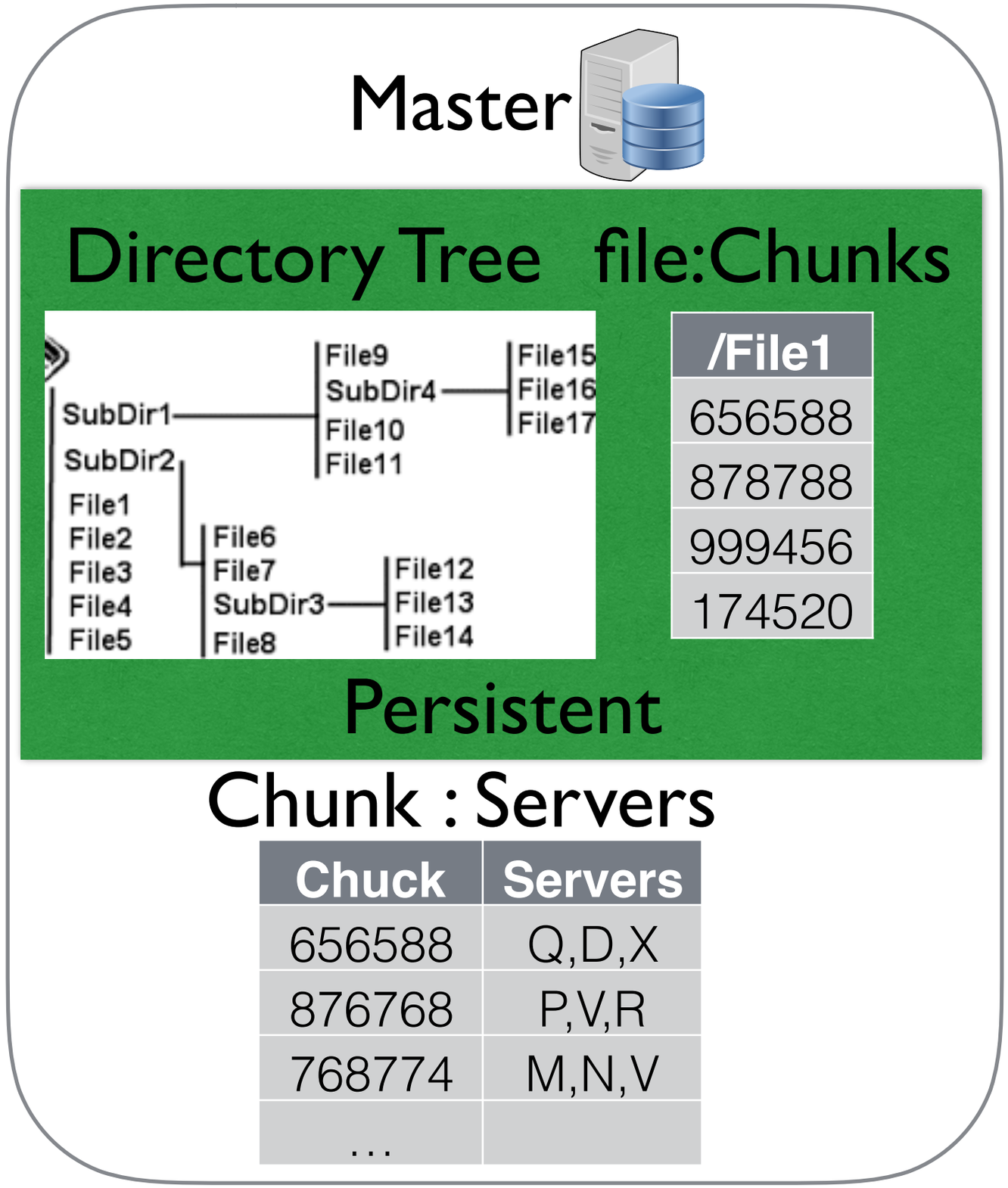
Two failure domains:

1. Master

2. Chunk Server

Fault Tolerance: Master

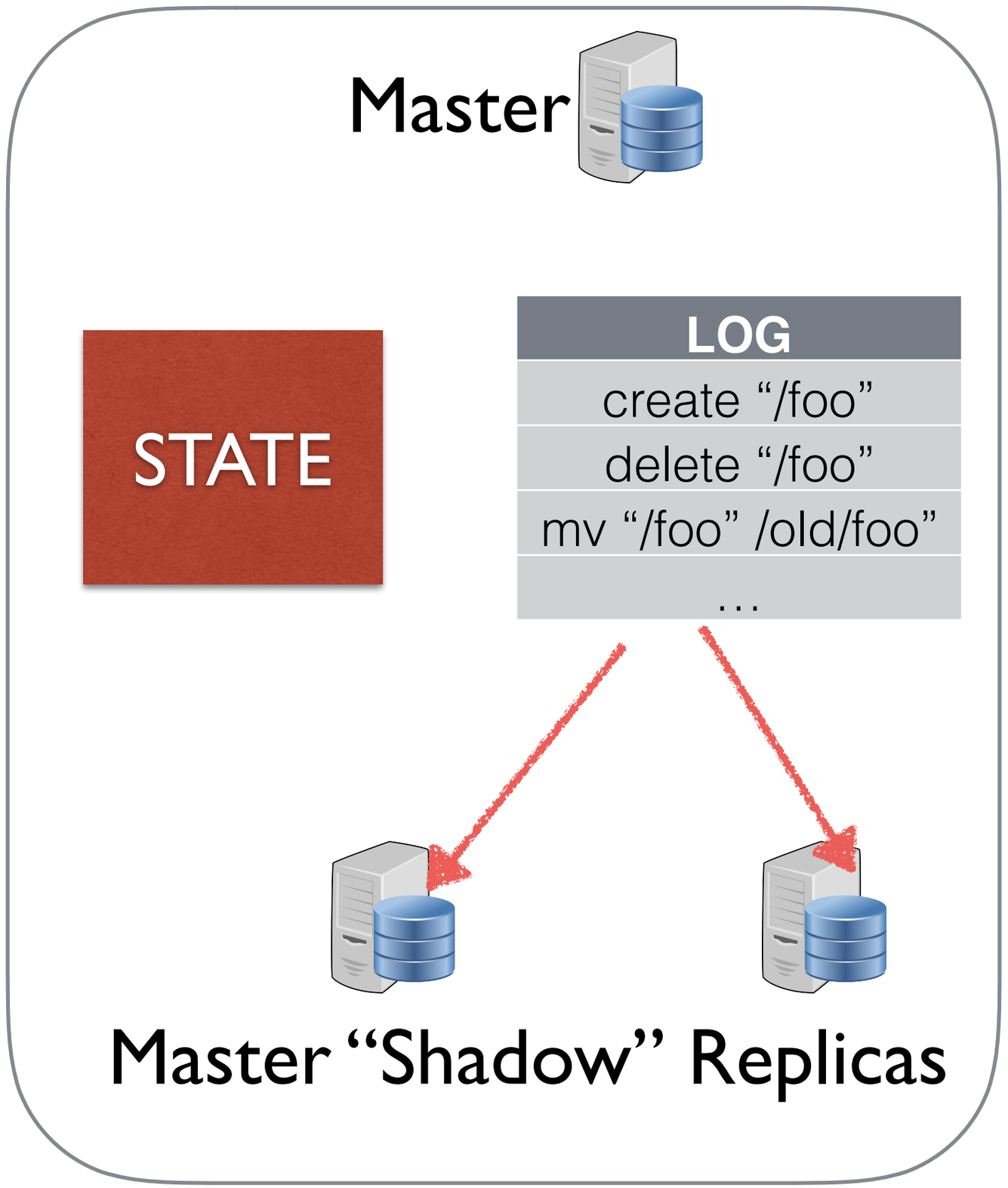
- Single master
- Clients talk to master
- Master orders all its operations



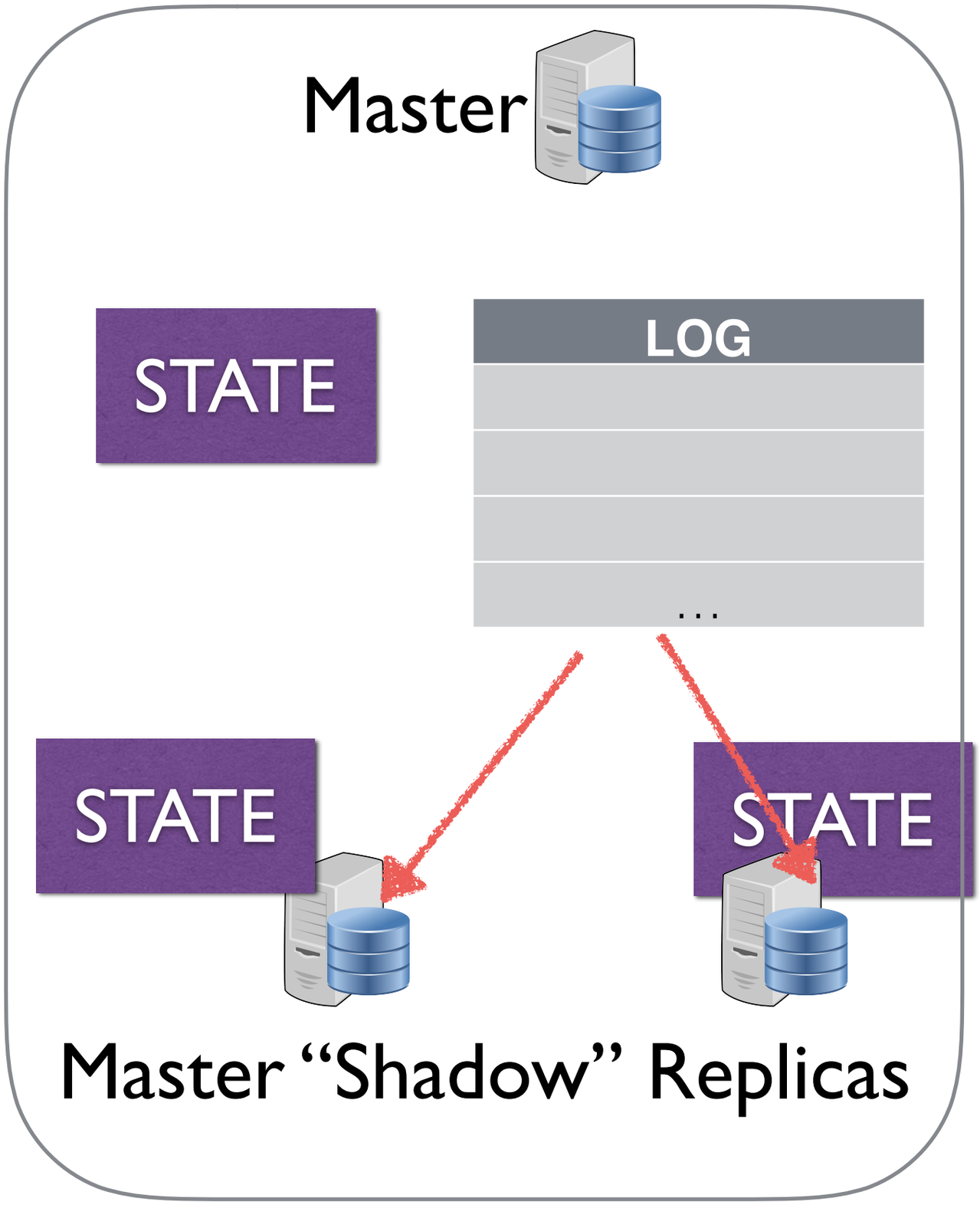
Fault Tolerance: Master

- Save history into an ordered log
- Log is replicated across several backups
- Clients operations that modify state return *after* recording changes in *logs*

Replication logs play a central role in most "strongly consistent" protocols (i.e., Lab #2!)

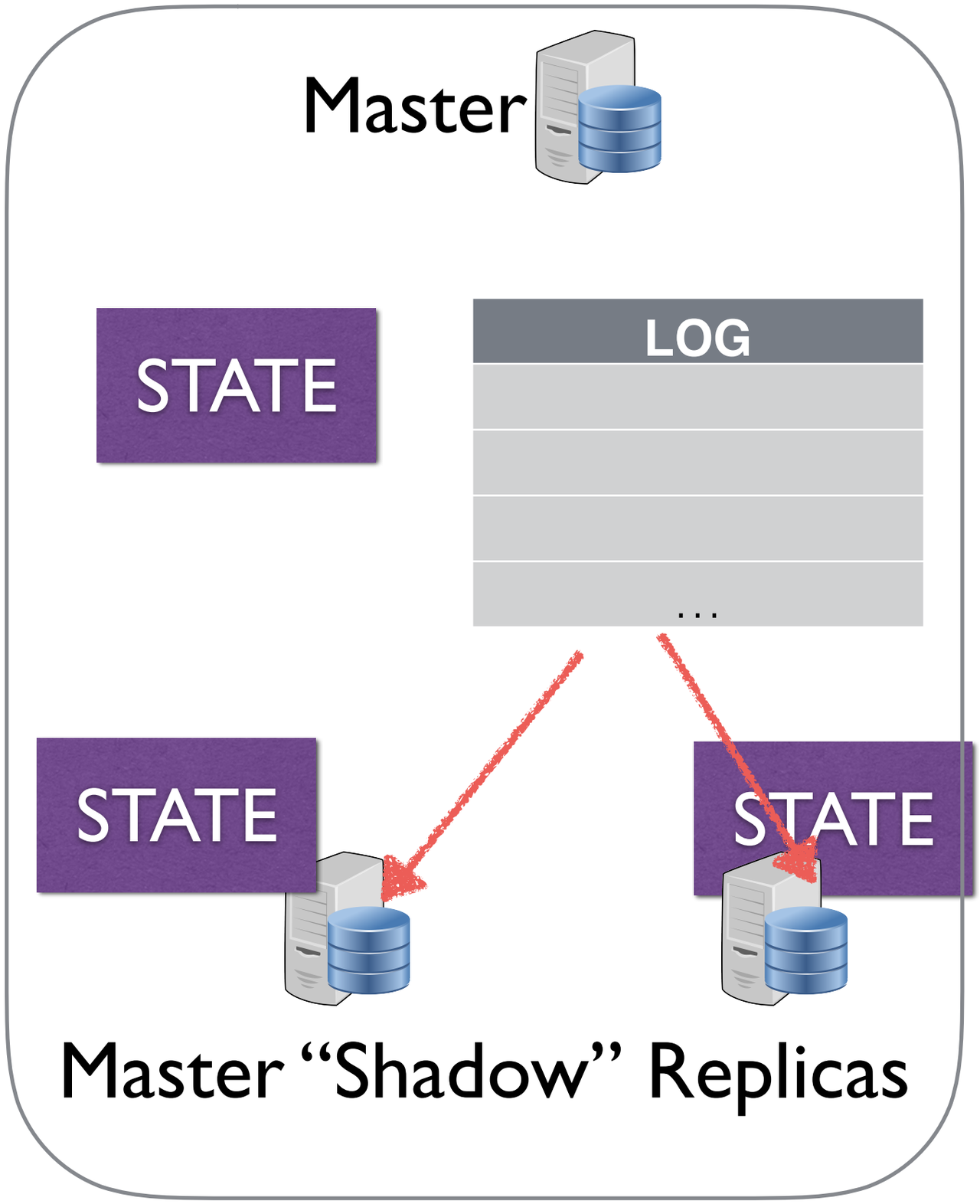


Fault Tolerance: Master



Fault Tolerance: Master

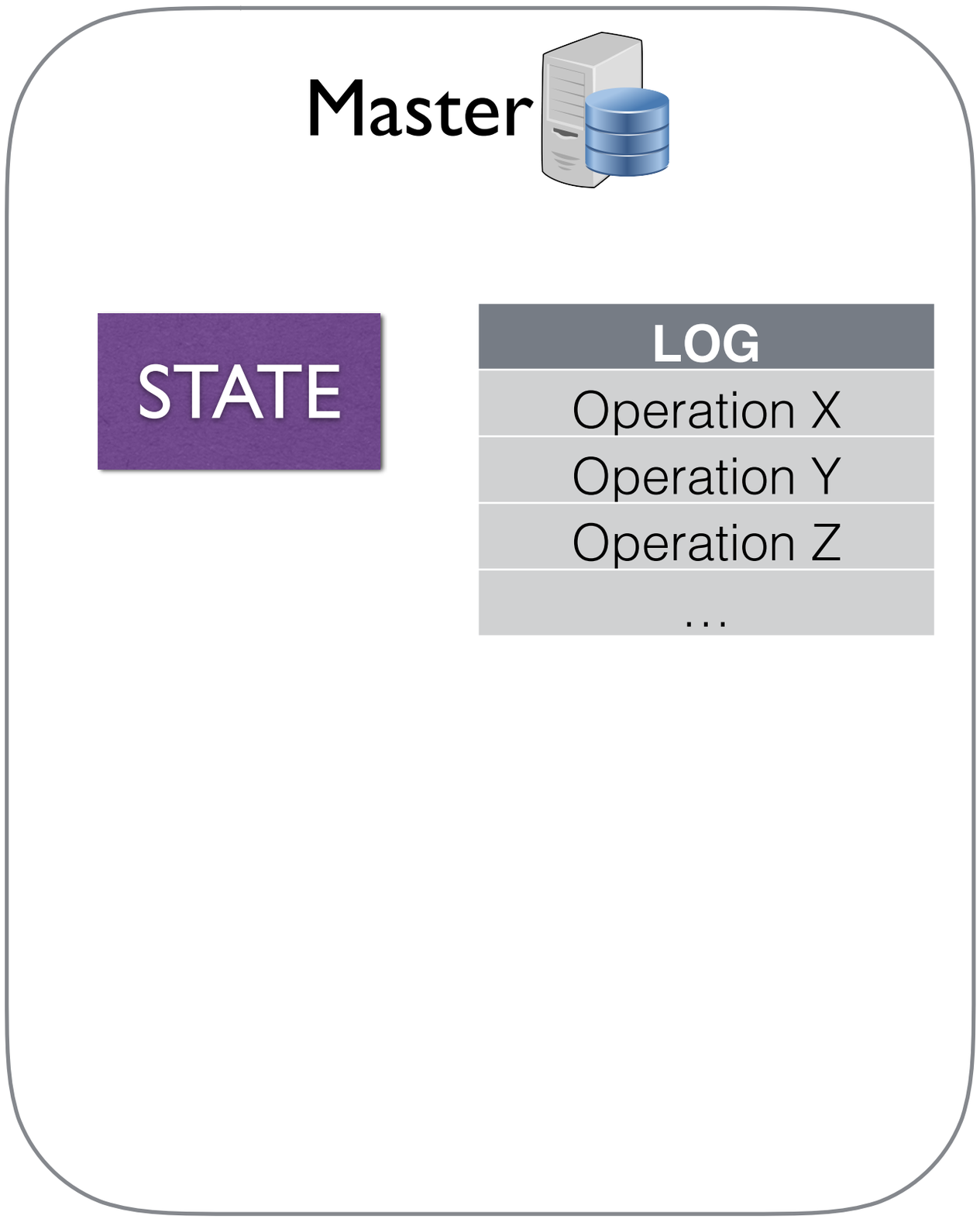
- Limit the size of the log with Checkpoints
- Make a checkpoint of the master state
- Remove all operations from log from before checkpoint
- Checkpoint is replicated to backups



Fault Tolerance: Master

Recovery

- Replay events from log, starting from last checkpoint
- Chunk location information is recreated by asking chunk servers



Fault Tolerance: Master

Master is single point of failure

- Recovery is "fast", because master state is small
- Service maybe unavailable for "short time"

Shadow masters (log replicas):

- State lags behind master
- Replay from the log that is replicated
- Can serve read-only — may return stale data