SE3331-1 (2023 Fall)

Distributed file system

NFS & GFS

Yubin Xia

IPADS, Shanghai Jiao Tong University

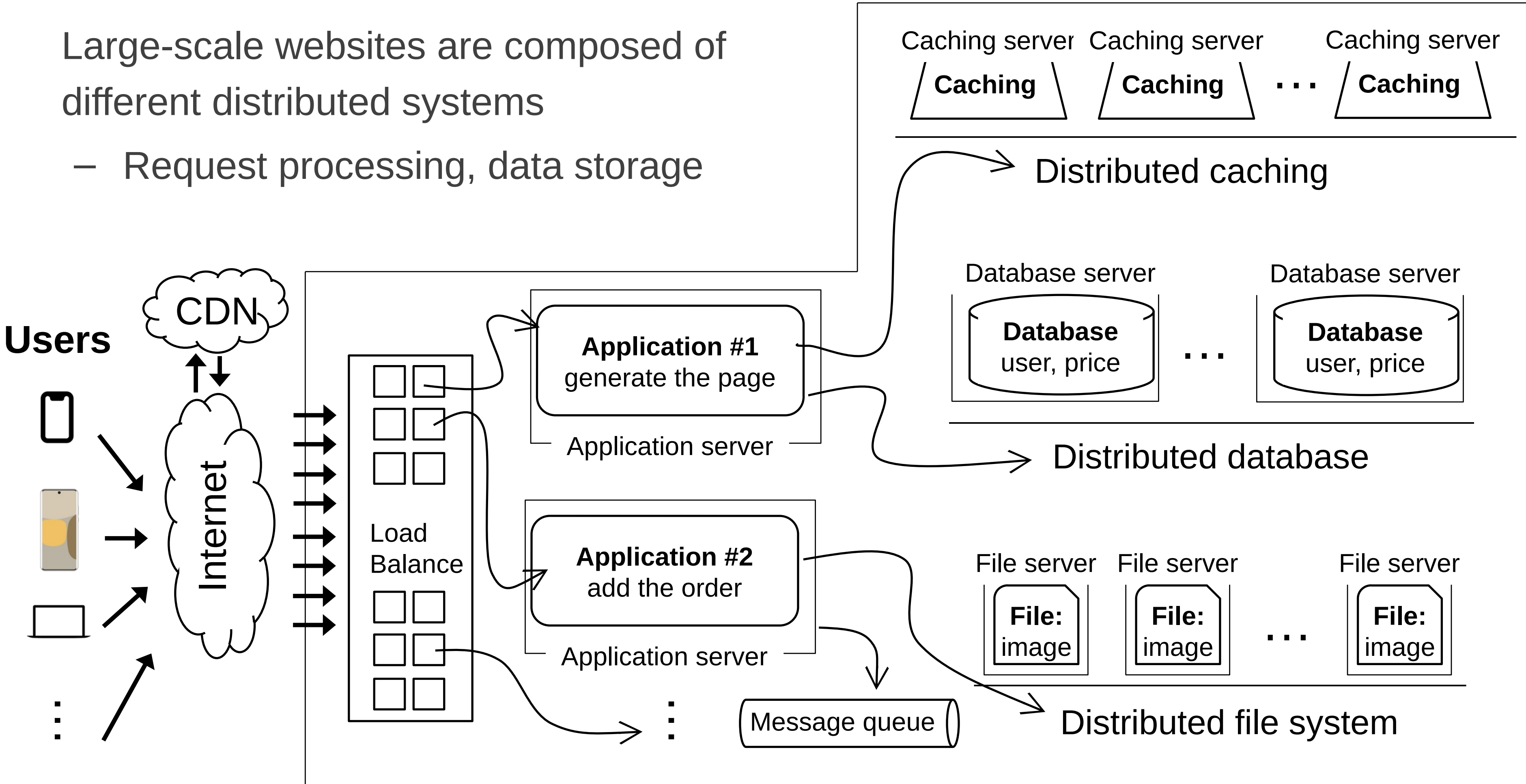
https://www.sjtu.edu.cn

Credits: Rong Chen@IPADS

Large-scale website so-far

Large-scale websites are composed of different distributed systems

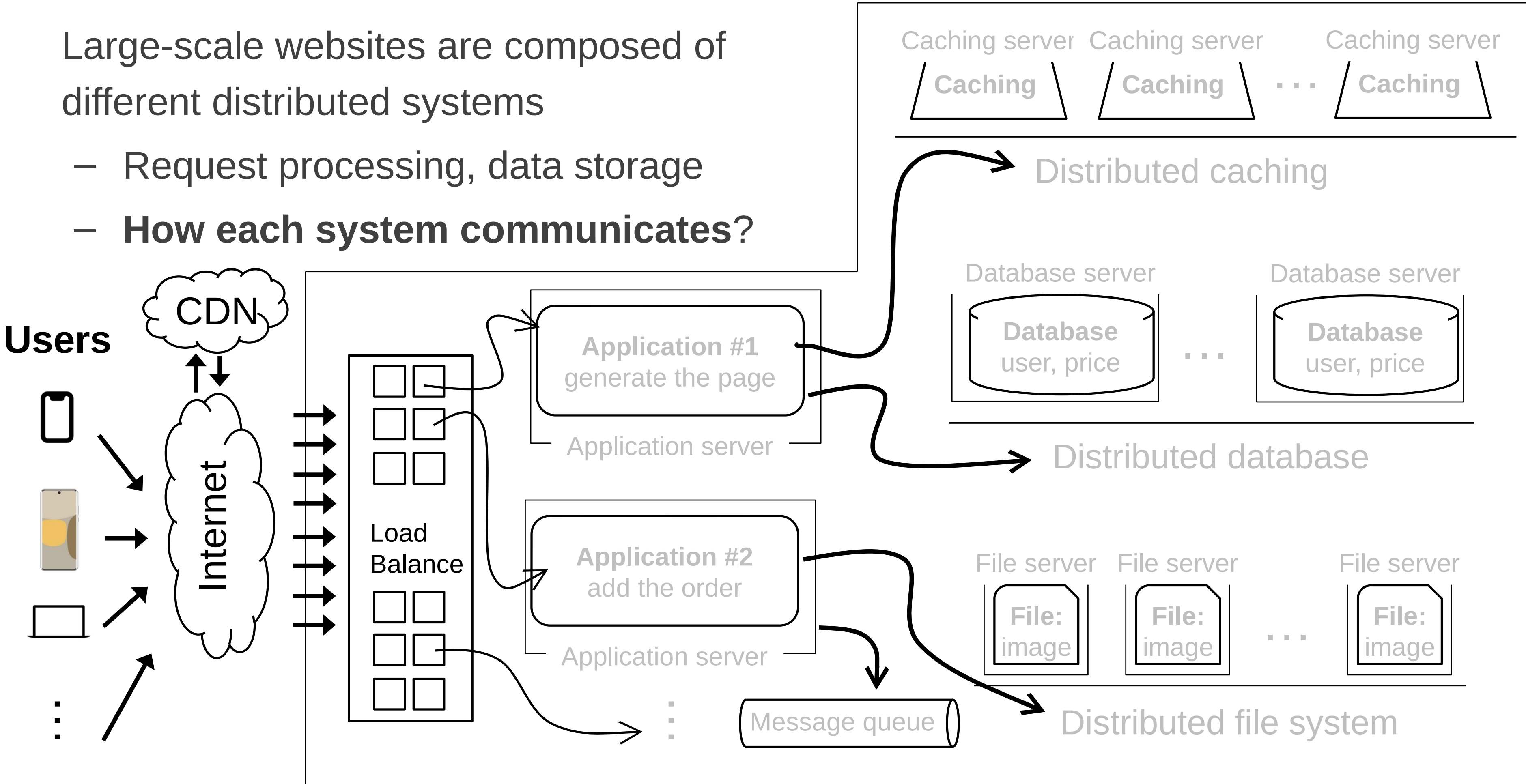
- Request processing, data storage



Large-scale website so-far

Large-scale websites are composed of different distributed systems

- Request processing, data storage
- **How each system communicates**?



Last lecture: Remote Procedure Call (RPC)

Allow a procedure to execute in another address space without coding the details for the remote interaction

RPC History

- Idea goes back in 1976

gRPC

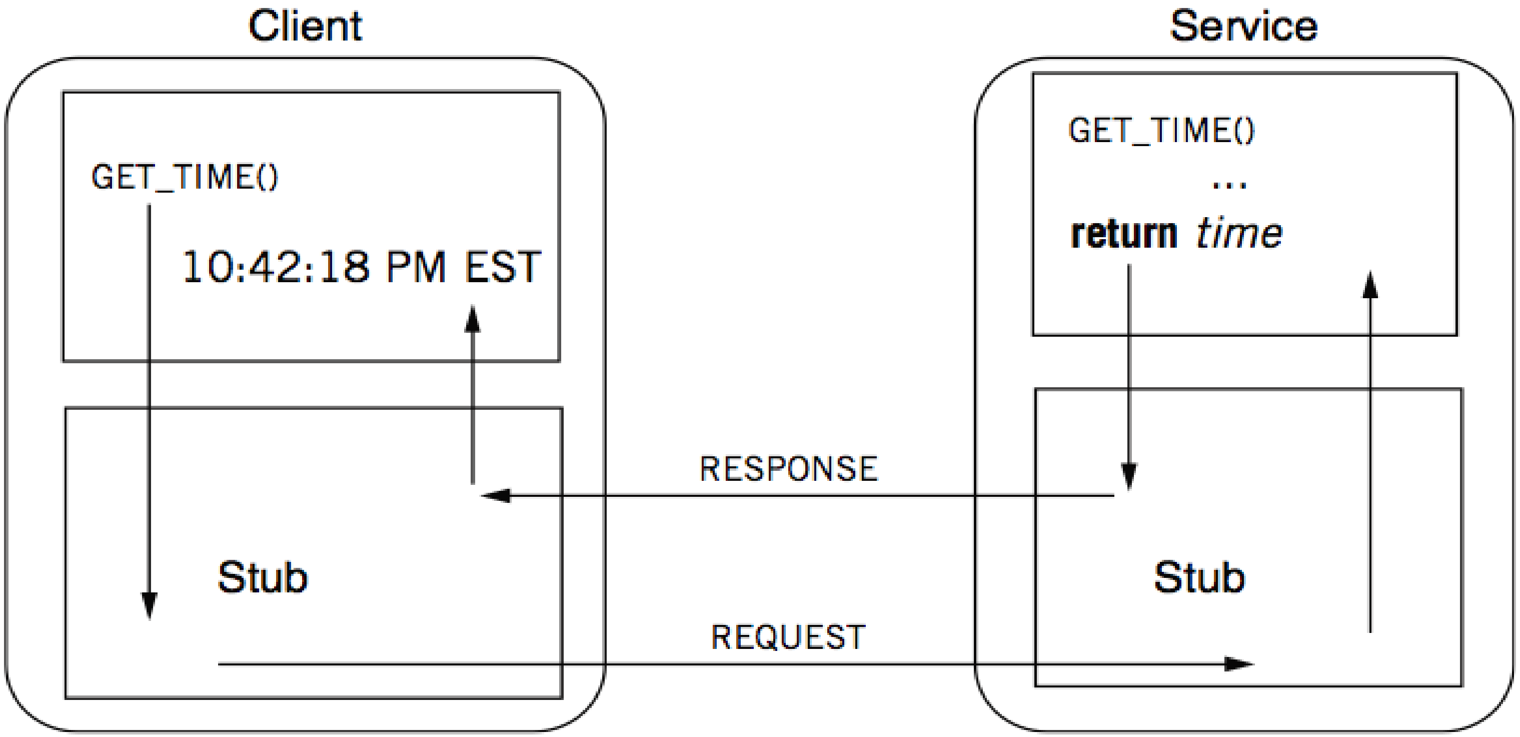
bRPC

RPC uses **stubs** to avoid handling argument **encoding/decoding** and send/receiving messages, message transports, etc.

How RPC handles **failures**?

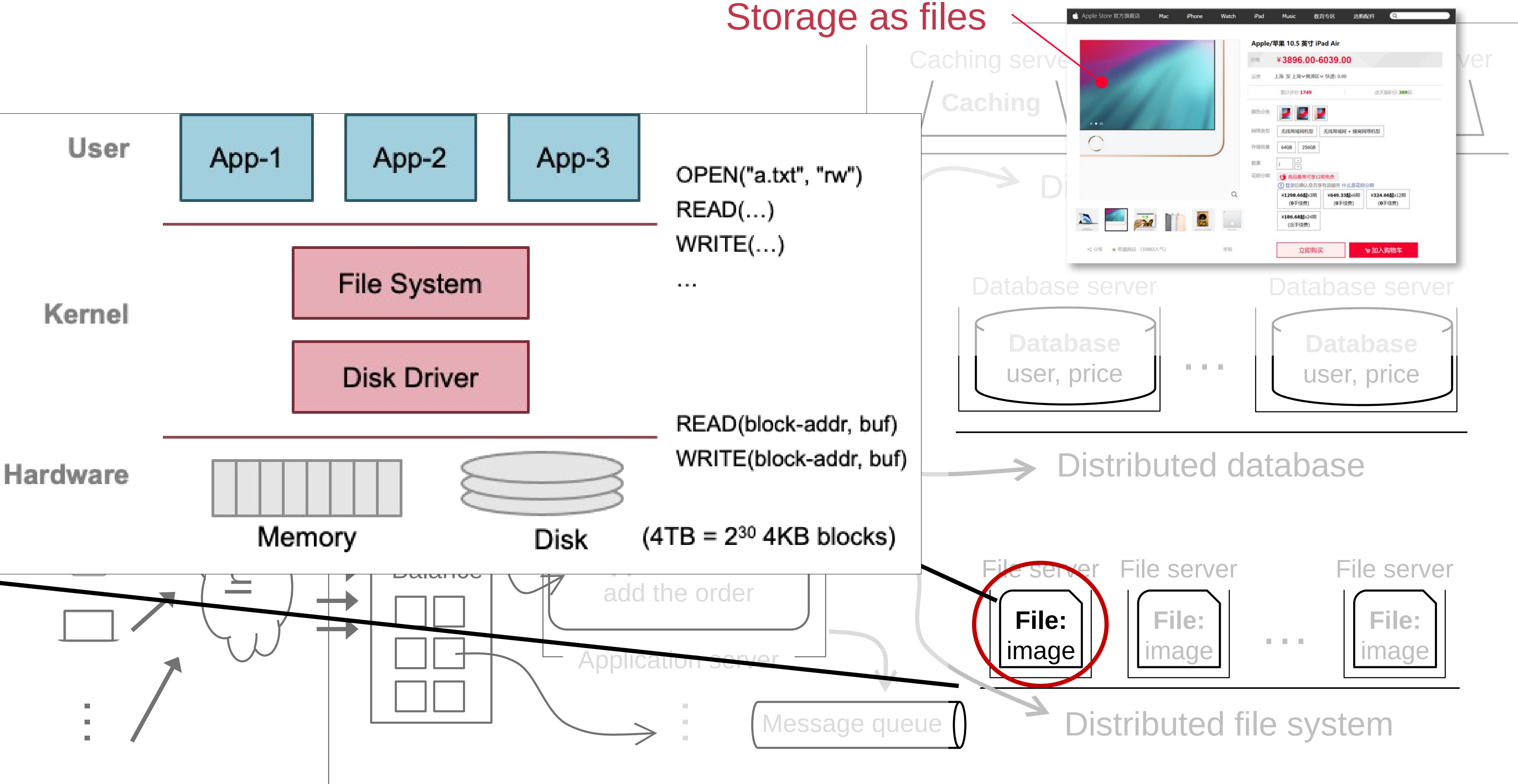
- Depends on the **semantic**: at-most-once, at-least-once & exactly-once

Review: RPC -- a complete calling process



Previous lecture: single-machine file system

Storage as files



OPEN("a.txt", "rw")
READ(…)
WRITE(…)
…

READ(block-addr, buf)
WRITE(block-addr, buf)

(4TB = 2^{30} 4KB blocks)

Apple Store 官方旗舰店 Mac iPhone Watch iPad Music 教育专区 选购配件

Apple/苹果 10.5 英寸 iPad Air

价格 ¥3896.00-6039.00

运费 上海 至 上海 黄浦区 快递: 0.00

累计评价 1749 送天猫积分 389起

颜色分类

网络类型 无线局域网机型 无线局域网 + 蜂窝网络机型

存储容量 64GB 256GB

数量 1

花呗分期 商品最高可享12期免息

登录后确认是否享有该服务 什么是花呗分期

¥1298.66起x3期 (0手续费) ¥649.33起x6期 (0手续费) ¥324.66起x12期 (0手续费)

¥186.68起x24期 (含手续费)

分享 ★ 收藏商品 (10883人气) 举报

立即购买 加入购物车

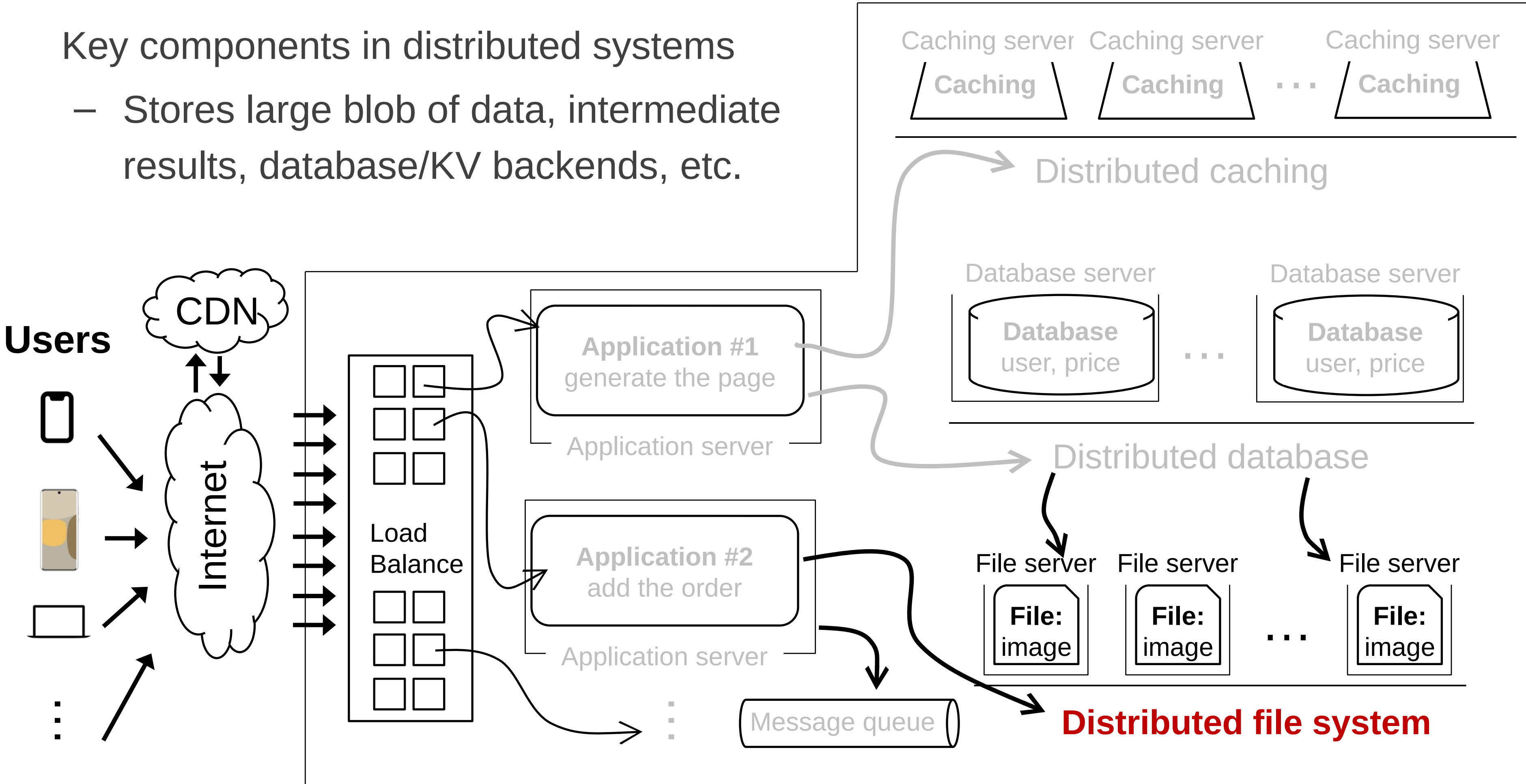
With the help of RPC

We can now build distributed file system!

Distributed file system

Key components in distributed systems

- Stores large blob of data, intermediate results, database/KV backends, etc.



Ways for accessing remote files

Many familiar ways, **FTP**, **telnet**, …

- Explicit access (knowing it is a distributed file system)
- User-directed connection to access remote resource

There is also transparent approach

- E.g., NFS, GFS, etc.
- Applications access the remote files like a local file

Distributed File Service Types

Upload/Download model

- Read file: copy file from server to client
- Write file: copy file from client to server

Advantage

- Simple

Problem

- **Wasteful** “what if client needs small pieces?”
- **Problematic** “what if client does not have enough space?”
- **Consistency** “what if others modify the same file?”

Distributed File Service Types

Remote access model

- File service provide functional interface with RPC (`create, delete, read, write`, etc.)

Advantage

- Client gets only what's needed
- Server can manage a consistent view of file system

Problem

- Possible server and network problem (e.g., congestion)
    - Servers are accessed for duration of file access
    - Same data may be requested repeatedly

NFS with RPC

NFS: Network File System

Design Goals (by Sun, 1980s, designed for workstations)

- Any machine can be a client or a server
- Support **diskless** workstations
- Support **Heterogeneous** deployment
  - Different HW, OS, underlying file system
- Access **transparency**
  - Use remote access model
- Recovery from **failure**
  - Stateless, UDP, client retries
- High **performance**
  - Use caching and read-ahead

RPC used in NFS

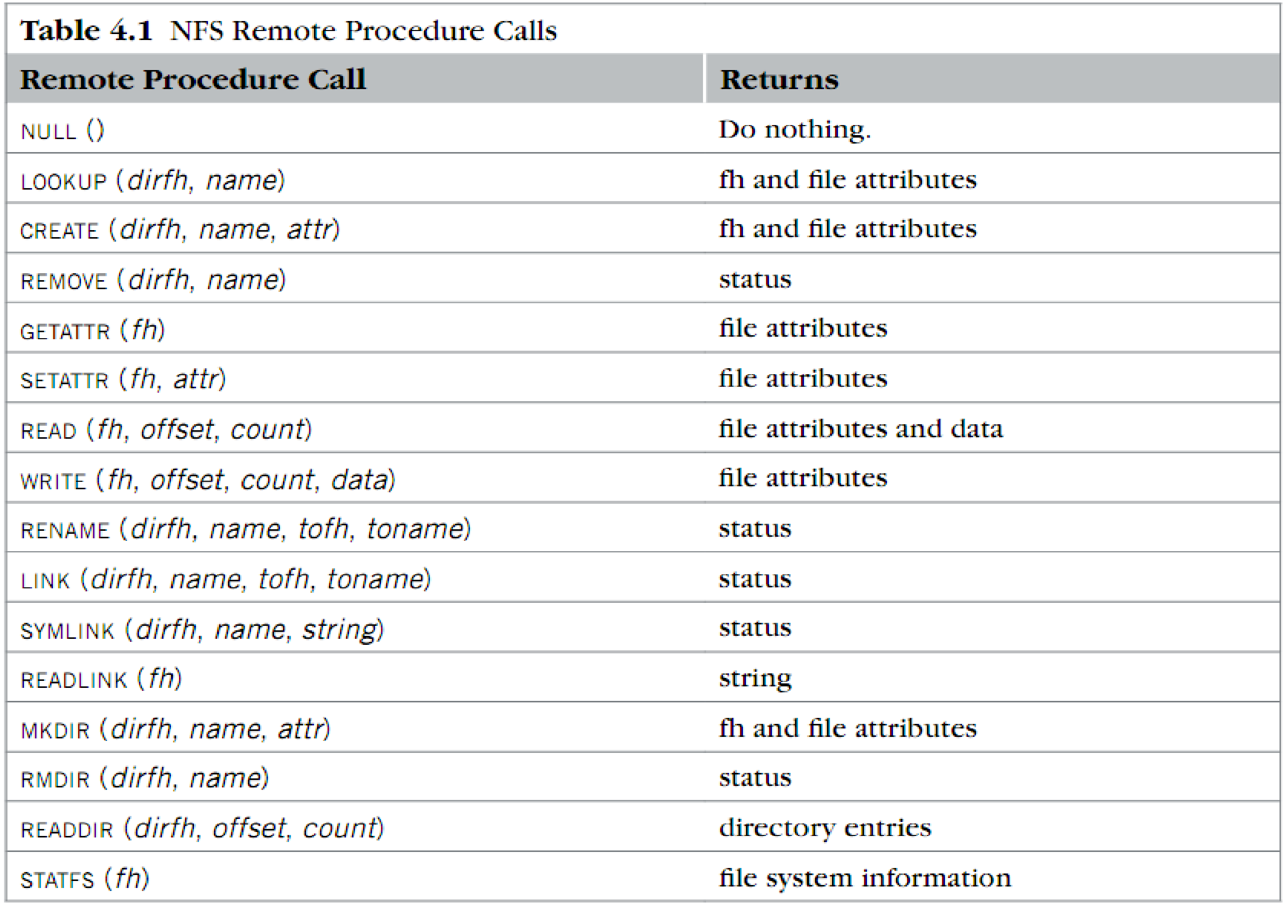


Table 4.1 NFS Remote Procedure Calls

Remote Procedure Call	Returns
NULL ()	Do nothing.
LOOKUP (*dirfh, name*)	fh and file attributes
CREATE (*dirfh, name, attr*)	fh and file attributes
REMOVE (*dirfh, name*)	status
GETATTR (*fh*)	file attributes
SETATTR (*fh, attr*)	file attributes
READ (*fh, offset, count*)	file attributes and data
WRITE (*fh, offset, count, data*)	file attributes
RENAME (*dirfh, name, tofh, toname*)	status
LINK (*dirfh, name, tofh, toname*)	status
SYMLINK (*dirfh, name, string*)	status
READLINK (*fh*)	string
MKDIR (*dirfh, name, attr*)	fh and file attributes
RMDIR (*dirfh, name*)	status
READDIR (*dirfh, offset, count*)	directory entries
STATFS (*fh*)	file system information

Where is OPEN and CLOSE?

NFS Protocols: Mount

Protocol:

- Request access to exported directory tree
    - Requests **permission** to access contents

Client: parses **pathname**
contacts server for file **handle**

- Server returns **file handle (fh)**

Client: create in-memory VFS **inode** (vnode) at
mount point internally points to remote files
(client keeps state, not the server)

NFS Protocols: `Mount`

Static mounting

- mount request contacts server

Server: add list of shared directories to /etc/exports

Client:

```
mount 192.168.1.100:/users/paul /home/paul
```

NFS Protocols: `Lookup/READ/WRITRE...`

Directory and File **Access** Protocol:

- Access files and directories (`read, mkdir`, ...)

First, perform a lookup RPC

- Returns file handle and attributes

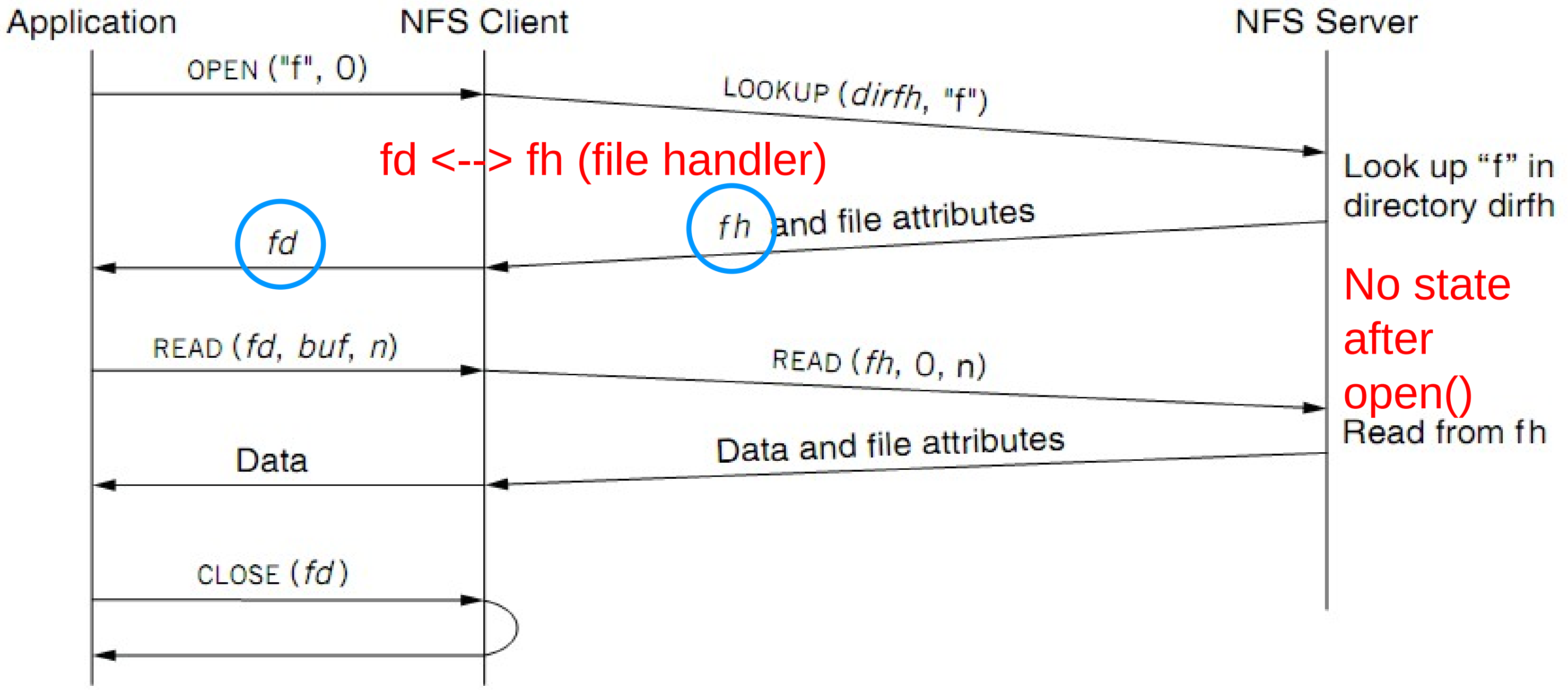
`lookup` is not like `open`

- Establish state on the client **only** (no information on server)
- Call the **NFS** lookup function

The **handle** passed as a parameter for other file access function

- e.g., `read(handle, offset, count)`

Read a file of NFS



fd <--> fh (file handler)

No state after open()

NFS Protocols: Lookup/READ/WRITRE...

NFS has 16 functions (version 2)

null
lookup

create
remove
rename

read
write

link
symlink
readlink

mkdir
rmdir
readdir

getattr
setattr

statfs

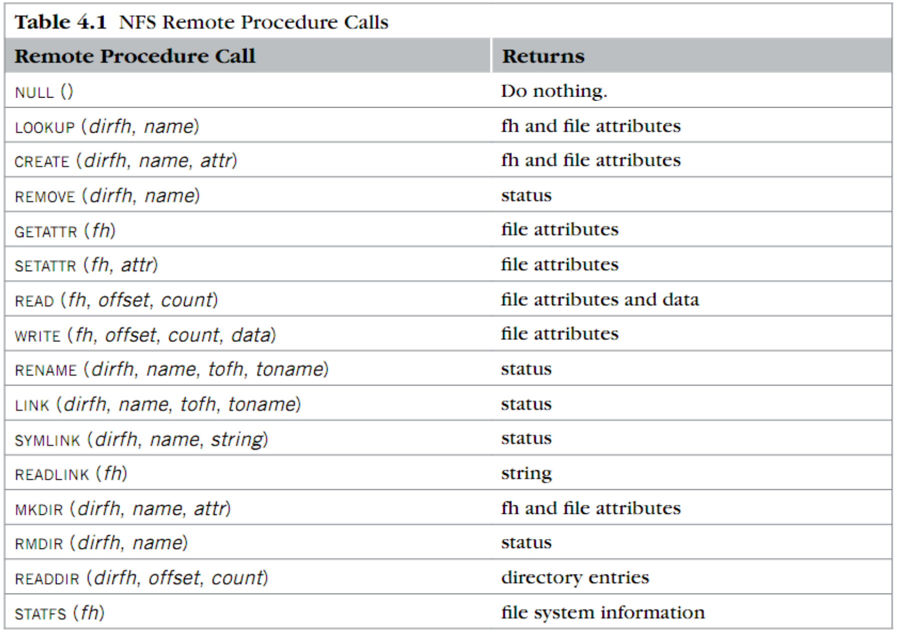


Table 4.1 NFS Remote Procedure Calls

Remote Procedure Call	Returns
NULL ()	Do nothing.
LOOKUP (*dirfh, name*)	fh and file attributes
CREATE (*dirfh, name, attr*)	fh and file attributes
REMOVE (*dirfh, name*)	status
GETATTR (*fh*)	file attributes
SETATTR (*fh, attr*)	file attributes
READ (*fh, offset, count*)	file attributes and data
WRITE (*fh, offset, count, data*)	file attributes
RENAME (*dirfh, name, tofh, toname*)	status
LINK (*dirfh, name, tofh, toname*)	status
SYMLINK (*dirfh, name, string*)	status
READLINK (*fh*)	string
MKDIR (*dirfh, name, attr*)	fh and file attributes
RMDIR (*dirfh, name*)	status
READDIR (*dirfh, offset, count*)	directory entries
STATFS (*fh*)	file system information

File Handler for a Client

File handler contains three parts

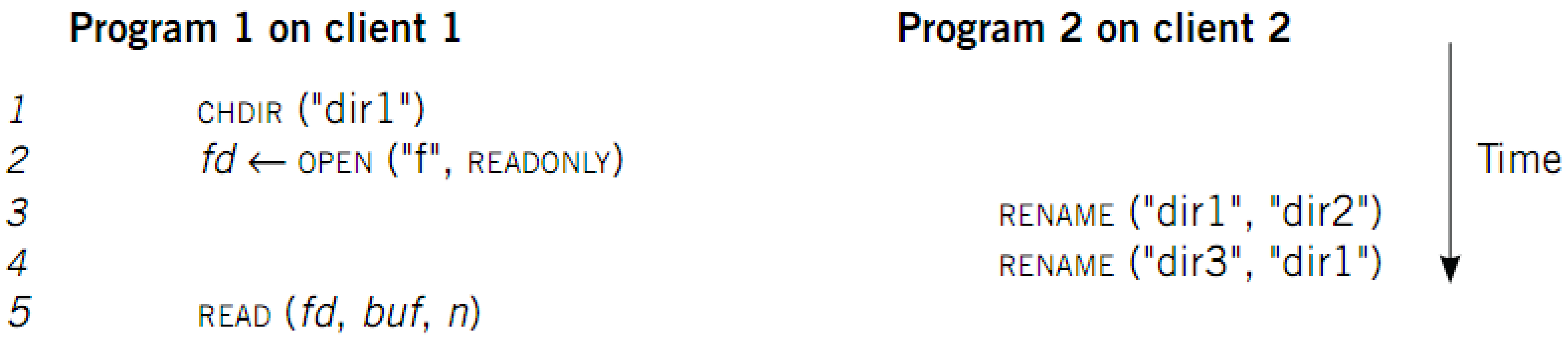
- **File system identifier**: for server to identify the file system
- **inode number**: for server to locate the file
- **Generation number**: for server to maintain consistency of a file

Can still work across server failures

- E.g., server reboot

Q: Why not put path name in the handle?

Case 1: Rename After Open



	Program 1 on client 1	Program 2 on client 2
1	CHDIR ("dir1")	
2	*fd* ← OPEN ("f", READONLY)	
3		RENAME ("dir1", "dir2")
4		RENAME ("dir3", "dir1")
5	READ (*fd*, *buf*, *n*)	

UNIX Spec:

- Program 1 should read "dir2/f"
- NFS should keep the spec

Stateless on NFS server

Stateless on NFS server

- Each RPC contains all the information

Q: What about states like file cursor?

- Client maintains the states, including the file cursor

Client can repeat a request until it receives a reply (at least once)

- Server may execute the same request twice
- Solution: each RPC is tagged with a transaction number, and server maintains some "soft" state: reply cache
- **Q**: What if the server fails between two same requests?

Case 2: Delete After Open

	Program 1 on client 1	Program 2 on client 2
1		*fd* ← OPEN ("f", READONLY)
2	UNLINK ("f")	
3	*fd* ← OPEN ("f", CREATE)	
4		READ (*fd*, *buf*, *n*)

UNIX spec:

- On local FS, program 2 will read the old file

How to avoid program 2 reading new file?

- Generation number
- "stale file handler"

Not the same as UNIX spec! It's a tradeoff...

NFS performance

Usually **slower** than local

- Question: depends on what?

File server performance & network speed

Optimization: **caching** at client

- **Goal**: reduce number of remote ops
- Caching: `read, readlink, getattr, lookup, readdir`
  1. Cache file data at client (buffer cache)
  2. Cache file attribute information at client
  3. Cache pathname bindings for faster lookup

Server side

- Caching is “automatic” via buffer cache
- All NFS writes are write-through to disk

Problem: cache coherence

Type-1: <u>Read/write coherence</u>

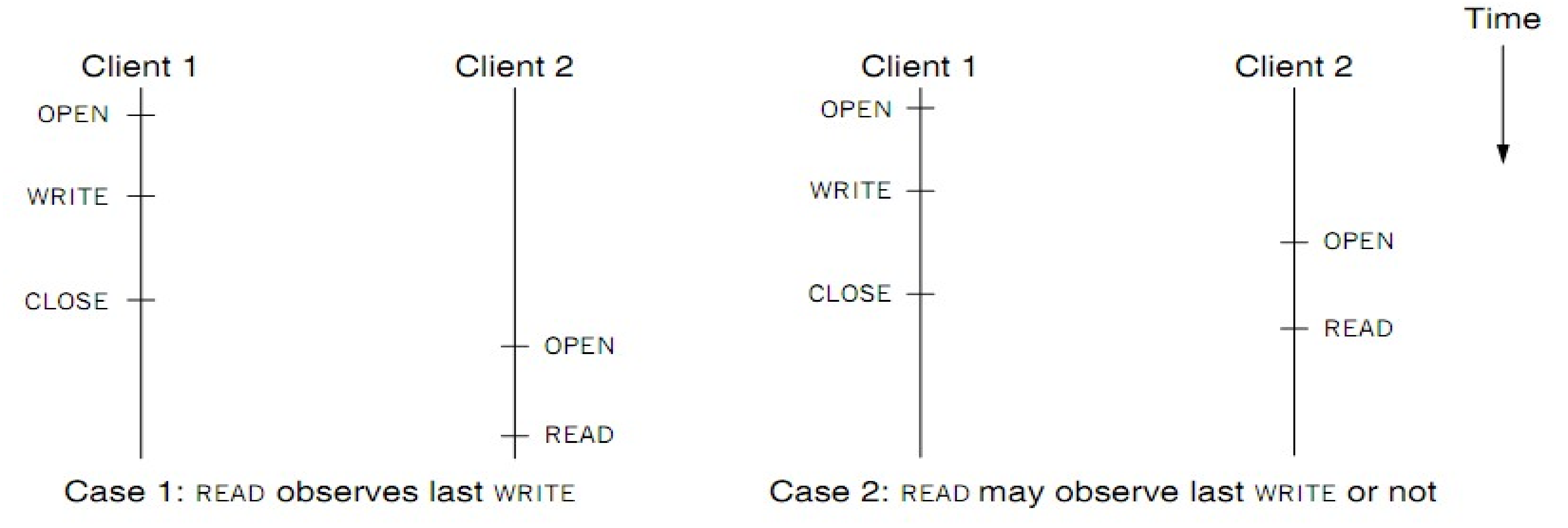
- On local file system, **READ** gets newest data
- On NFS, client has cache
- NFS could guarantee read/write coherence for every operation, or just for certain operation

Type-2: <u>Close-to-open consistency</u>

- Higher data rate
- **GETATTR** when **OPEN**, to get last modification time
- Compare the time with its cache
- When **CLOSE**, send cached writes to the server

Coherence

Two cases of close-to-open consistency



More contents of consistency in later lectures

VFS: Extend the inode-based FS to support NFS

Vnode

- Abstract whether a file or directory is local or remote
- In volatile memory (why?)
- Support several different local file system
- Where should vnode layer be inserted?

Vnode API

- Same as we learnt: OPEN, READ, WRITE, CLOSE…
- Code of fd_table, current dir, file name lookup, can be moved up to the file system call layer

Validation

Inconsistencies may arise in NFS

Resolve **inconsistencies** with **validation**

Both server and client save **timestamp** of files

When file opened or server contacted for new

1. Compare last modification time
2. If remote is more recent, invalidate cached data

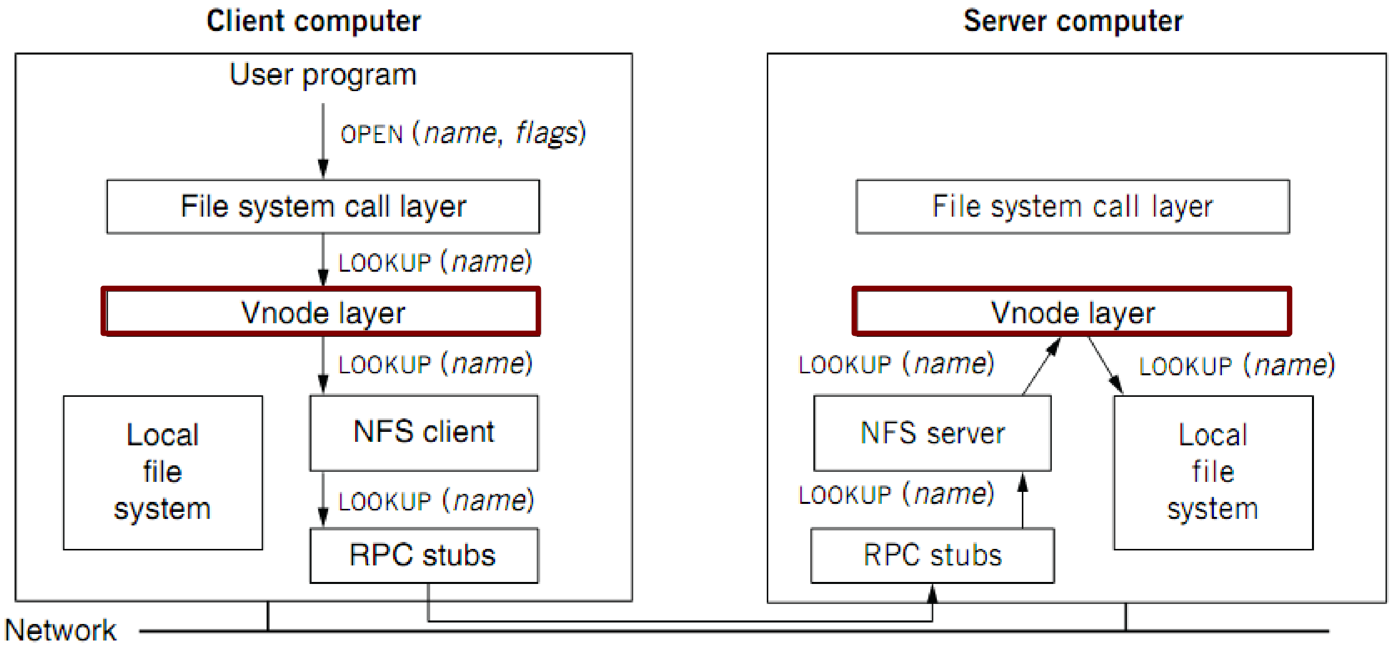
Always **invalidate** data after some time

– `open` files (3 sec), `directories` (30 sec)

If data block is modified, it is:

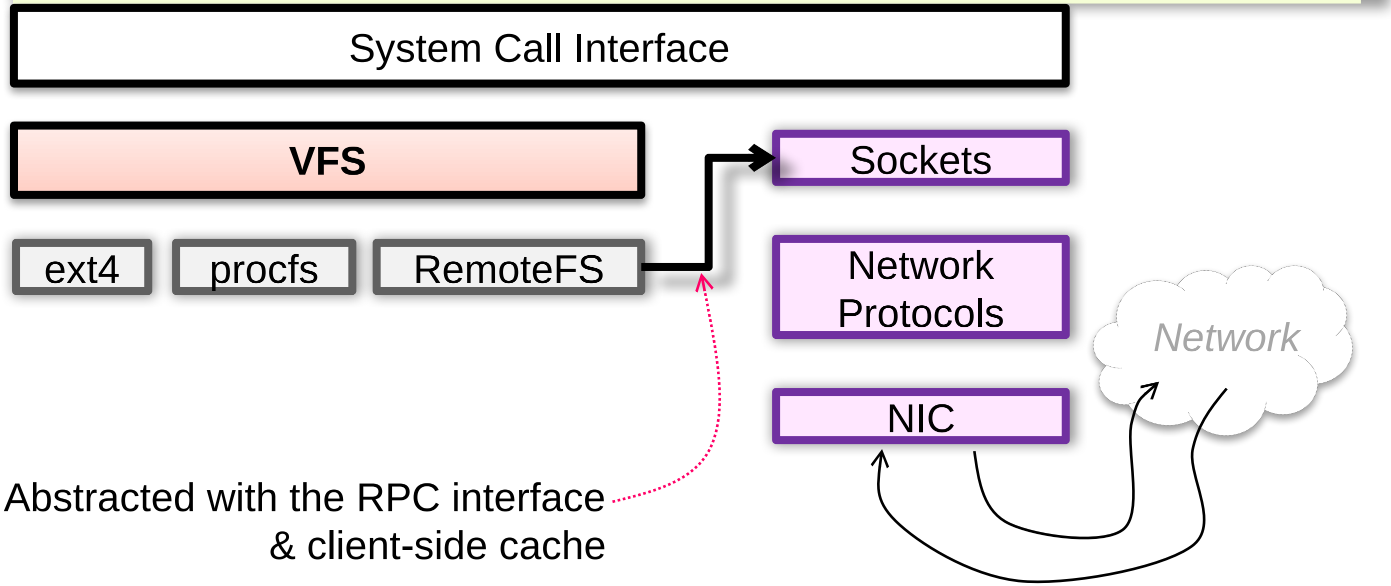
– Marked **dirty**, then flushed on file `close`

VFS: Extend the inode-based FS to support NFS



Accessing Remote Files

Implement the client module as a FS under VFS



Abstracted with the RPC interface
& client-side cache

Improving Read Performance

Transfer data in large chunks

- 8KB default

Read-ahead

- Optimize for sequential file access
- Send requests to read disk blocks before they are requested by the applications

NFS is continuously improving

Version 3

User-level lock manager

NVRAM support

Adjust RPC retries dynamically

Client-side disk caching

Support 64-bit file sizes

TCP support and large-block transfers

Commit operation

…

Version 4

More state: control of caching, notify of file changes

Server export a single name space (pseudo file system)

Compound RPC support

Extended attribute and ACL

Negotiate security mechanism on mount

…

Reference Materials

RFC 1831: RPC Specification

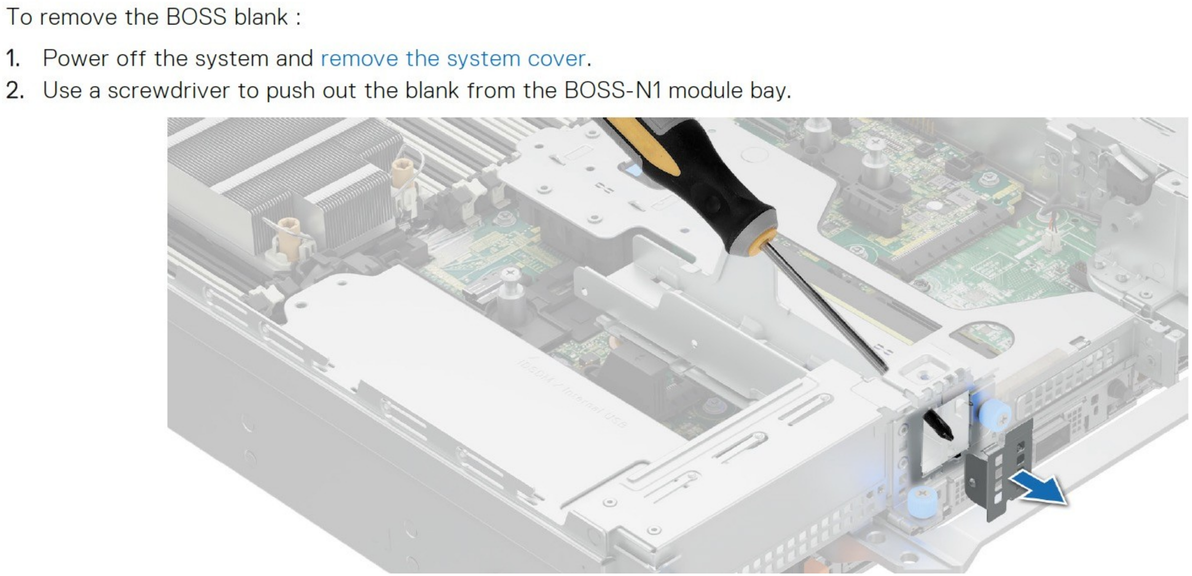
- http://www.ietf.org/rfc/rfc1831.txt?number=1831

RFC 1832: XDR Specification

- http://www.ietf.org/rfc/rfc1832.txt?number=1832

Drawback of NFS

1. **Capacity**
   - Can only disks on a single server, which has a limited capacity to insert disks



To remove the BOSS blank :

1. Power off the system and remove the system cover.
2. Use a screwdriver to push out the blank from the BOSS-N1 module bay.

Drawback of NFS

1. **Capacity**

   – Can only disks on a single server, which has a limited capacity

2. **Reliability**

   – If the server crashes, the remote files are unavailable

3. **Performance**

   – The file performance is limited to a single file (and a single network bandwidth)

Observation: plenty of available servers in a datacenter

Plenty of machines in a datacenter

- E.g., much cheaper than a high-end server

Idea: utilis multiple machines to form a single, large distributed file system

- We can leverage the aggregated capacity of all the machines!

Naïve solution: do manual partitioning

- User decides which files / directories are stored on which server
- Not so good: dates back to the FTP case ⏪

Design goal: transparent remote file accesses

We don’t want the developers to manage which file store on which server

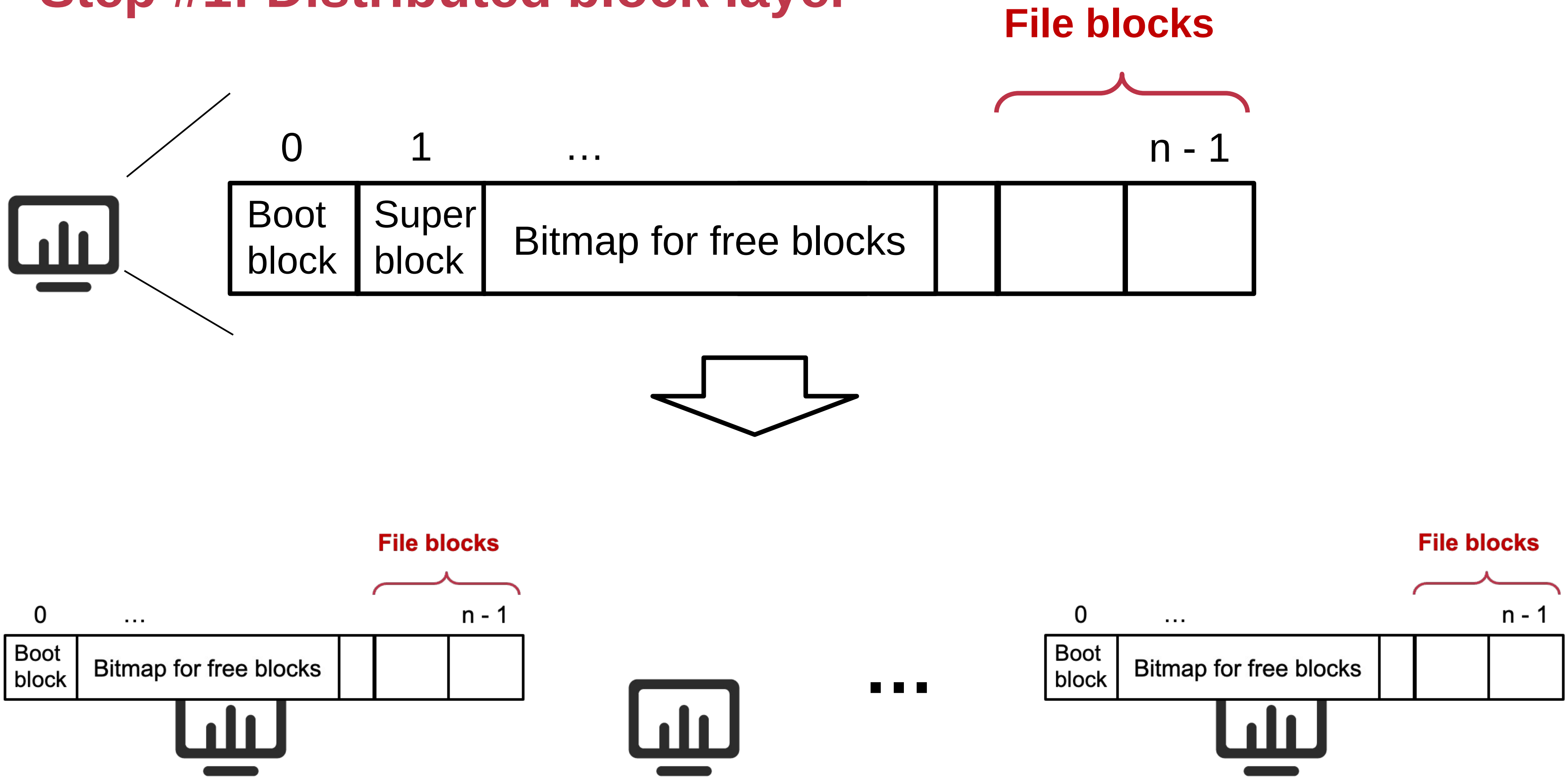
- It is difficult to the developers
- Also, does not support existing applications

But, it means that we cannot reuse the inode-based filesystem implementations

- Nevertheless, the overall principles are the same, e.g., we still needs things like block layer, inode layer, etc.

We will see many challenges faced by distributed systems during our journal ⏪

Step #1: Distributed block layer



Step #1: Distributed block layer

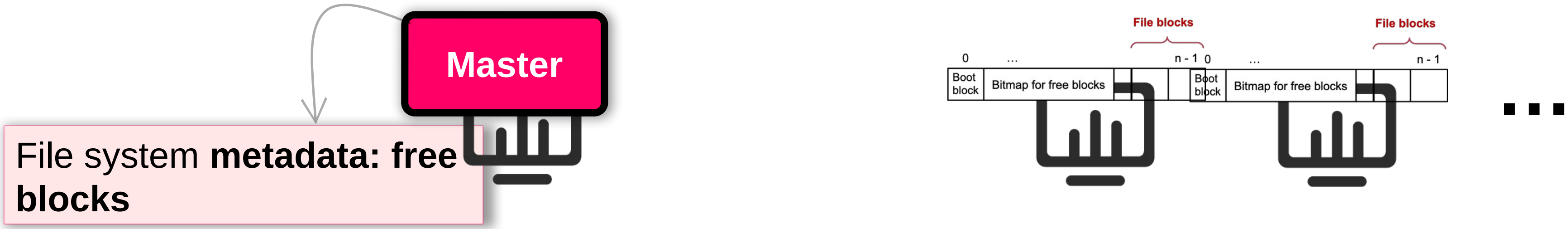
File (inode)	Block num	Disk Block

Q1. How to access the block in a distributed setting?

- Simple, we can extend the block_id to <mac_id, block_id>

Q2. Second, how can a client know which machine has a free block?

- Sol#1. select a random server; if it has free block, then we are done; if not, retry until we can find one.
- Sol#2. use a master server to record the free blocks at each machine, and all the block allocation & deallocation go to the master



File system **metadata: free blocks**

L2: File Layer: do we need a redesign?

File (inode)	Block num	Disk Block

Recall: Given an inode, can map a block index number (of a file) to a block number (of a disk)

- Index number: e.g., the 3rd block of a file is number 78

No need for now for now for functionality, since we can access the blocks

- With our previous designed distributed layer

There are some other issues, e.g., performance, reliability, consistency, etc.

- We will back to these example later

L3: Distributed inode Number Layer

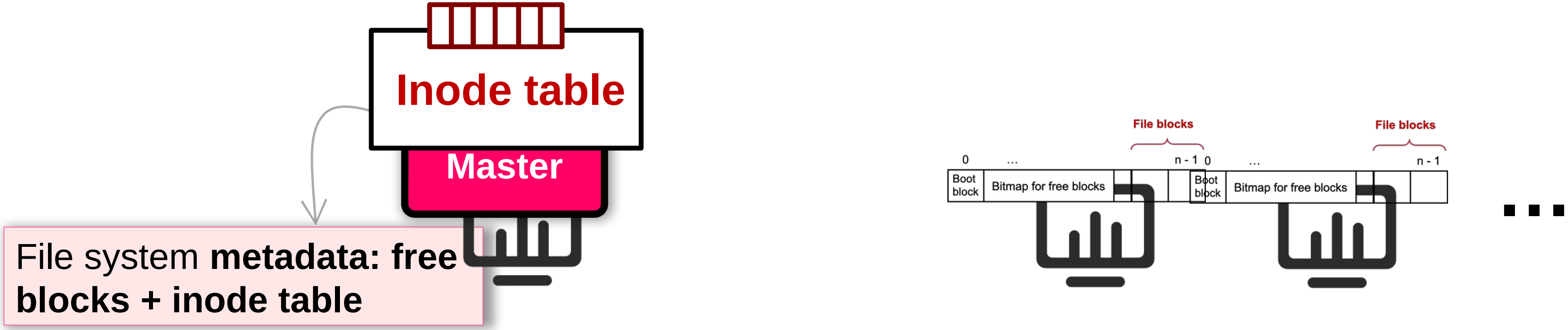
Inode num	File (inode)	Block num	Disk Block

Mapping: **inode number -> inode**

inode table: at a fixed location on storage

- inode number is the index of inode table
- Track which inode number are in use, e.g. free list, a field in inode

Distributed inode number layer: store the inode table at the master!



File system **metadata: free blocks + inode table**

L4: File Name Layer

File name	Inode num	File (inode)	Block num	Disk Block

File name

- Hide metadata of file management
- Files and I/O devices

```
struct inode
  integer block_nums[N]
  integer size
  integer type
```

Mapping

- Mapping table is saved in directory
- Default context: **current working directory**
  - Context reference is an inode number
  - The current working directory **is also a file**

Overview of inode content

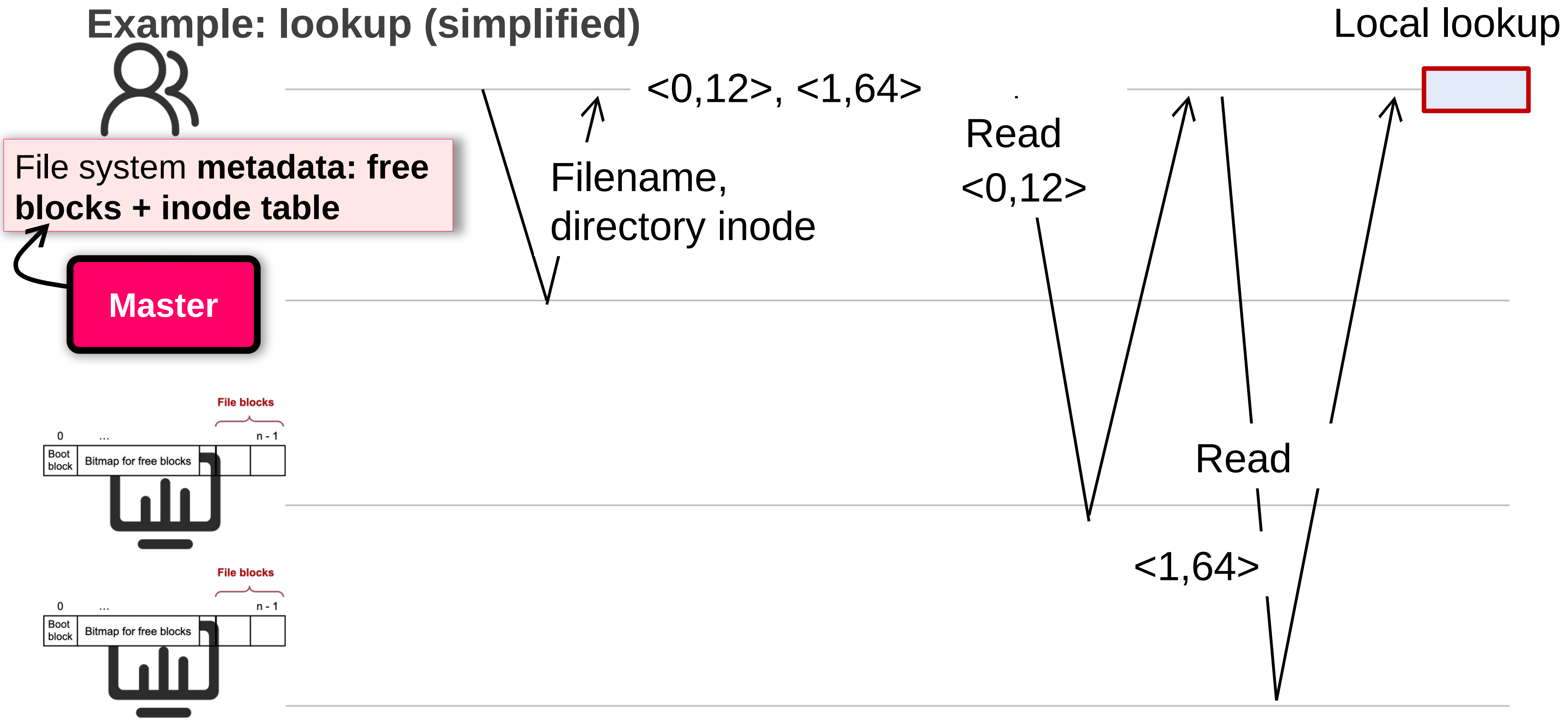
File name	inode num
helloworld.txt	12
cse2021.md	73

Question

- Do we need to extend it to a distributed setup? So far, so good!

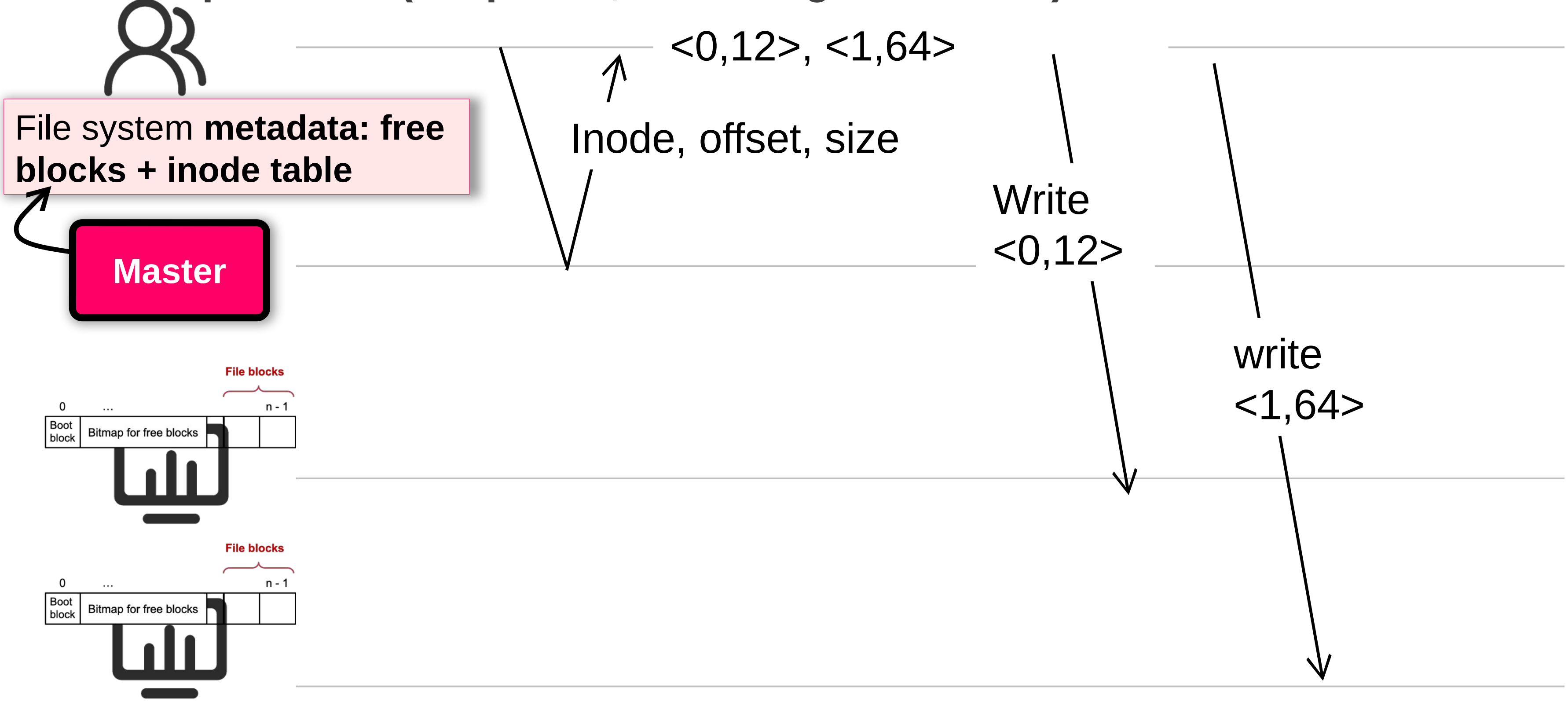
Put the distributed file layers so far together

Example: lookup (simplified)



Put the distributed file layers so far together

Example: write (simplified, assuming no blocks)



File system **metadata: free blocks + inode table**

Master

Inode, offset, size

<0,12>, <1,64>

Write <0,12>

write <1,64>

L5: Path Name Layer

Path name	File name	Inode num	File (inode)	Block num	Disk Block

Hierarchy of directories and files

- Structured naming: E.g. "projects/paper"

```
procedure PATH_TO_INODE_NUMBER(string path, integer dir)-> integer
    if PLAIN_NAME(path) return NAME_TO_INODE_NUMBER(path,dir)
    else
        dir <- LOOKUP(FIRST(path), dir)
        path <- REST(path)
        return PATH_TO_INODE_NUMBER(path,dir)
```

Context: the working directory **`dir`**

If we have the lookup, then we can also do the path lookup ⏪

Issues of our naïve design so far

Performance

- E.g., path lookup is slow due to multiple RTTs
- Though we can use cache, like we have done in NFS ⏪
- But, what about data in a file? If the read is large, it uses many blocks, so many servers needed to be communicated

Reliability

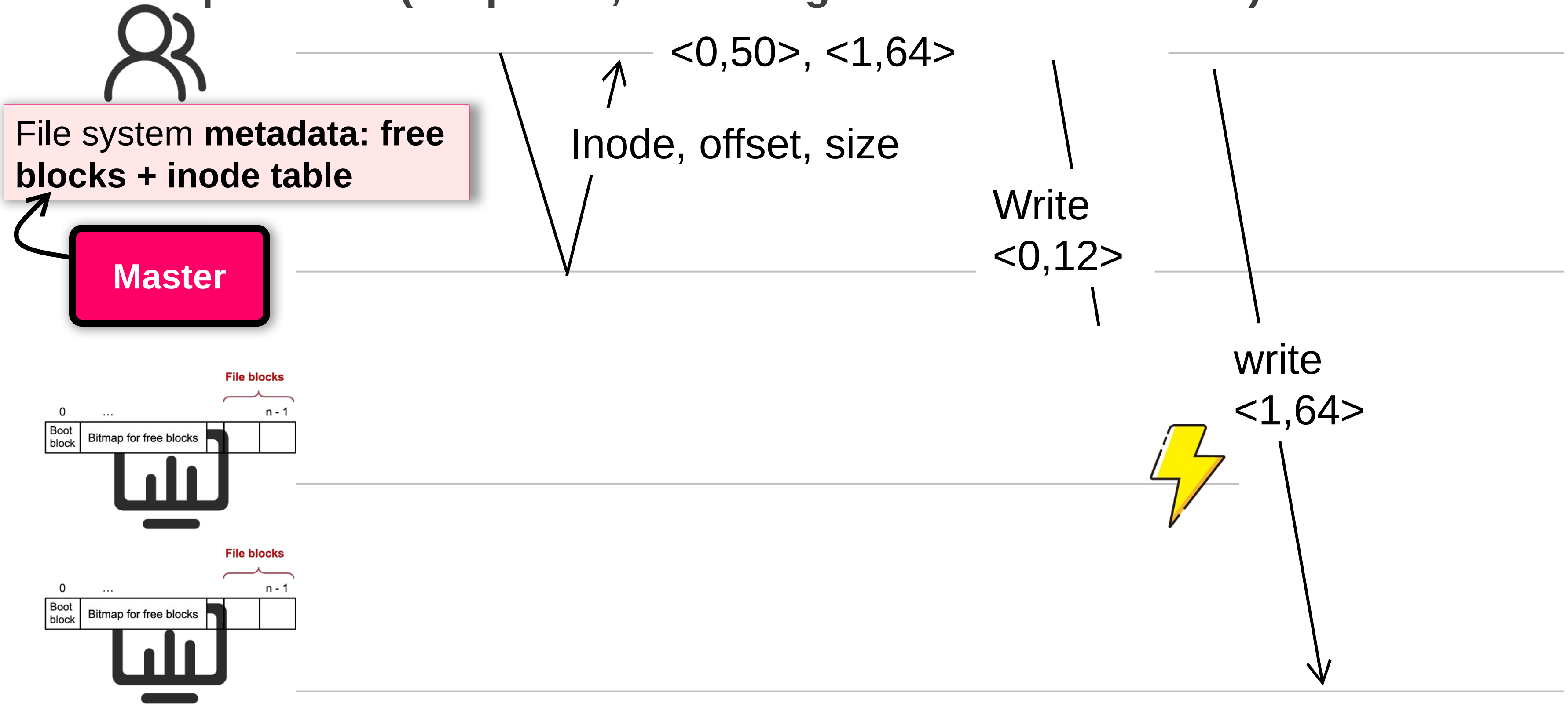
- E.g., what if the master fails?
- E.g., what if some servers that store the blocks fail?

Correctness

- If a failure happens, the overall system states will be corrupted!

Correctness broken under failure: example

Example: write (simplified, assuming no blocks allocation)



File system **metadata: free blocks + inode table**

Master

Inode, offset, size

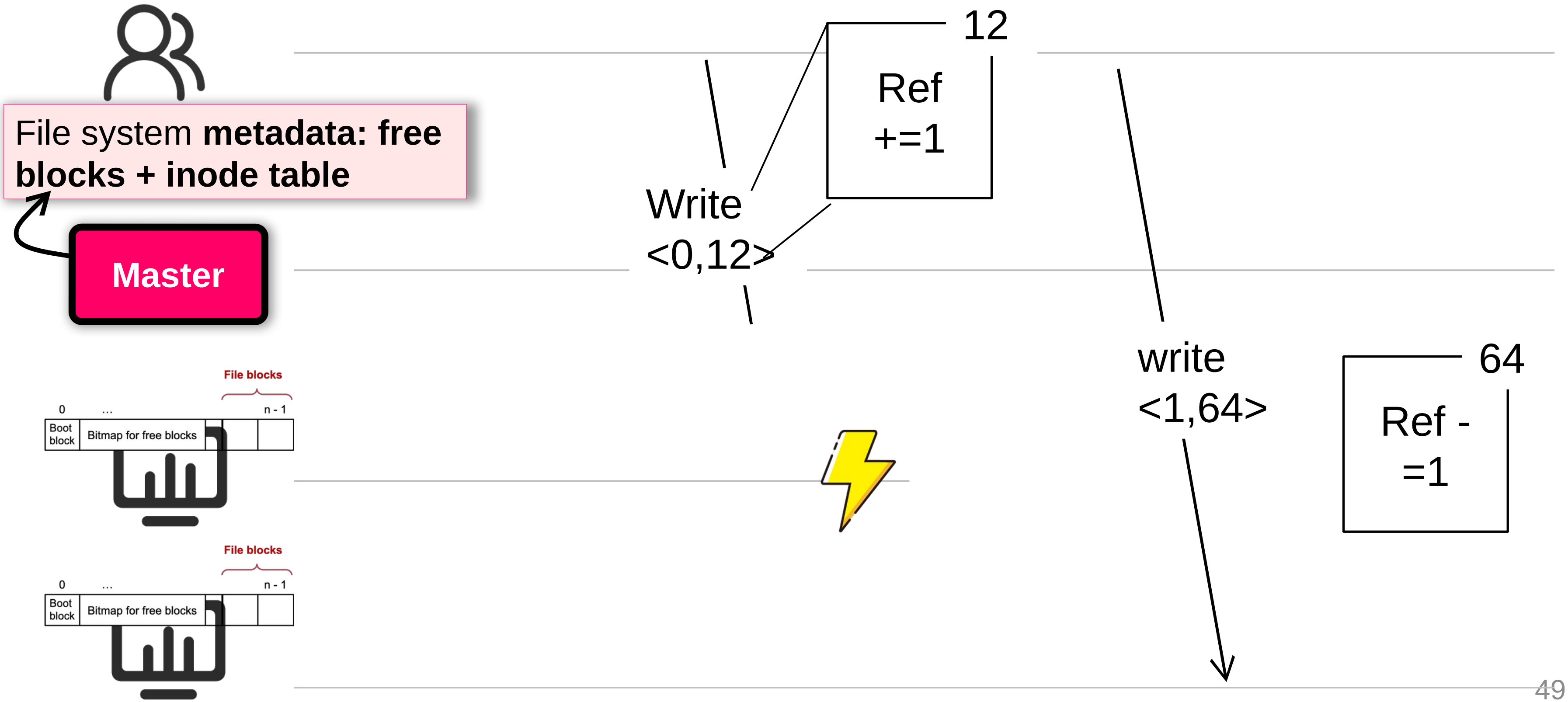
<0,50>, <1,64>

Write <0,12>

write <1,64>

Correctness broken under failure: example

Example: rename (simplified with only reference counters operations)



File system **metadata: free blocks + inode table**

Master

Failure is a subtle issue & can even happens under in inode-base filesystem

We will talk about the issues in later lectures

We need more “weapons” to cope w/ the above issues

Performance

- E.g., path lookup is slow due to multiple RTTs
- Though we can use cache, like we have done in NFS ⏪

Reliability **=> we need data replication**

- E.g., what if the master fails?
- E.g., what if some servers that store the blocks fail?

Correctness **=> we need to define what is correct & how to achieve so** ⏪

- If a failure happens, the overall system states will be corrupted!

Case study: GFS

The Google File System

Sanjay Ghemawat, “The Google File System”. SOSP, **2003**

GFS design goals

Scalable distributed file system

- E.g., Can NFS stores a very large file?

Designed for large **data-intensive** applications

- Essential for distributed computing frameworks, e.g., MapReduce

Fault-tolerant; runs on commodity hardware

- E.g., what if a server crashes, in the case of NFS?

Delivers **high performance** to a large number of clients

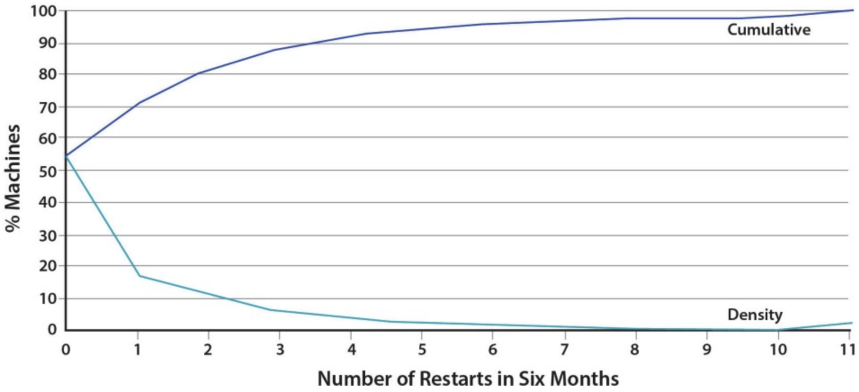
Design Assumptions

Assumptions for **conventional** file systems **don’t work**

- E.g., “most files are small”, “short lifetimes”

Component **failures** are the **norm**, not an exception

- File system = thousands of storage machines
- Some % not working at any given time (recall from lecture2)



Machine failures at Google. Source: An Introduction to the Design of Warehouse-Scale Machines

Design Assumptions: environments

Assumptions for **conventional** file systems **don’t work**

- E.g., “most files are small”, “short lifetimes”

Component **failures** are the **norm**, not an exception

- File system = thousands of storage machines
- Some % not working at any given time (recall from lecture2)

Files are huge: n-GB/TB files are the norm, e.g., large web index

- I/O ops and block size choices are affected

More Design Assumptions: File Access

Most files are **appended**, not overwritten (why? E.g., add new web pages to the Google’s global store)

- **Random** writes within a file are **rare**
  - Means we need to provide efficient **appends**

Workload is mostly:

- mostly reads: large streaming reads;
  - Once created, files are **mostly read**; often **sequential** (e.g., read a previous archived page)

We should designing the **FS API** with the design of **apps** benefits the system

GFS interface

GFS does **not** have a **standard OS-level API**

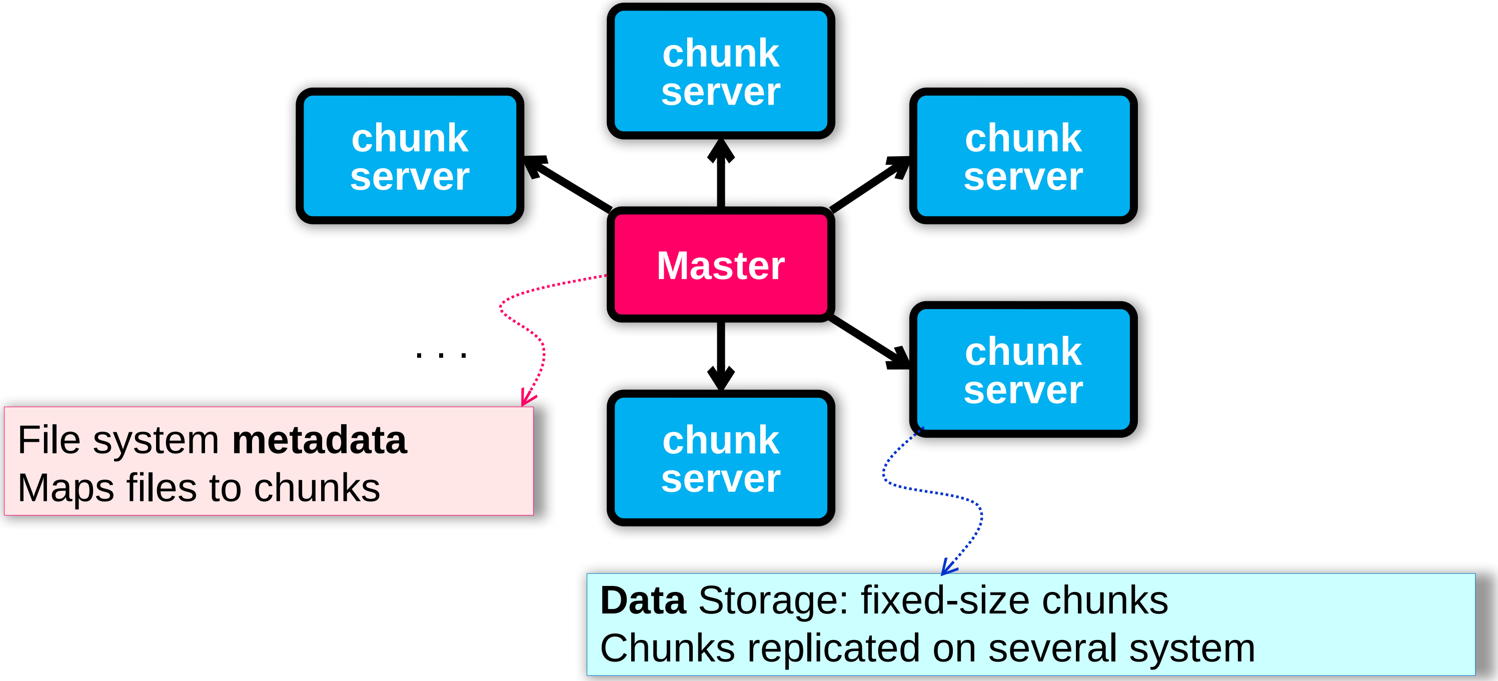
- No POSIX API
- No kernel/VFS implementation
- It provides user-level API

Operations

- Basic ops: `create/delete/open/close/read/write`
- Additional ops: `snapshot/append`
- Not supported ops: `link, symlink, rename`
- Why not rename? As I illustrated before, hard to ensure consistency under failures (no mature distributed TX---see later lectures---at that time)

GFS architecture

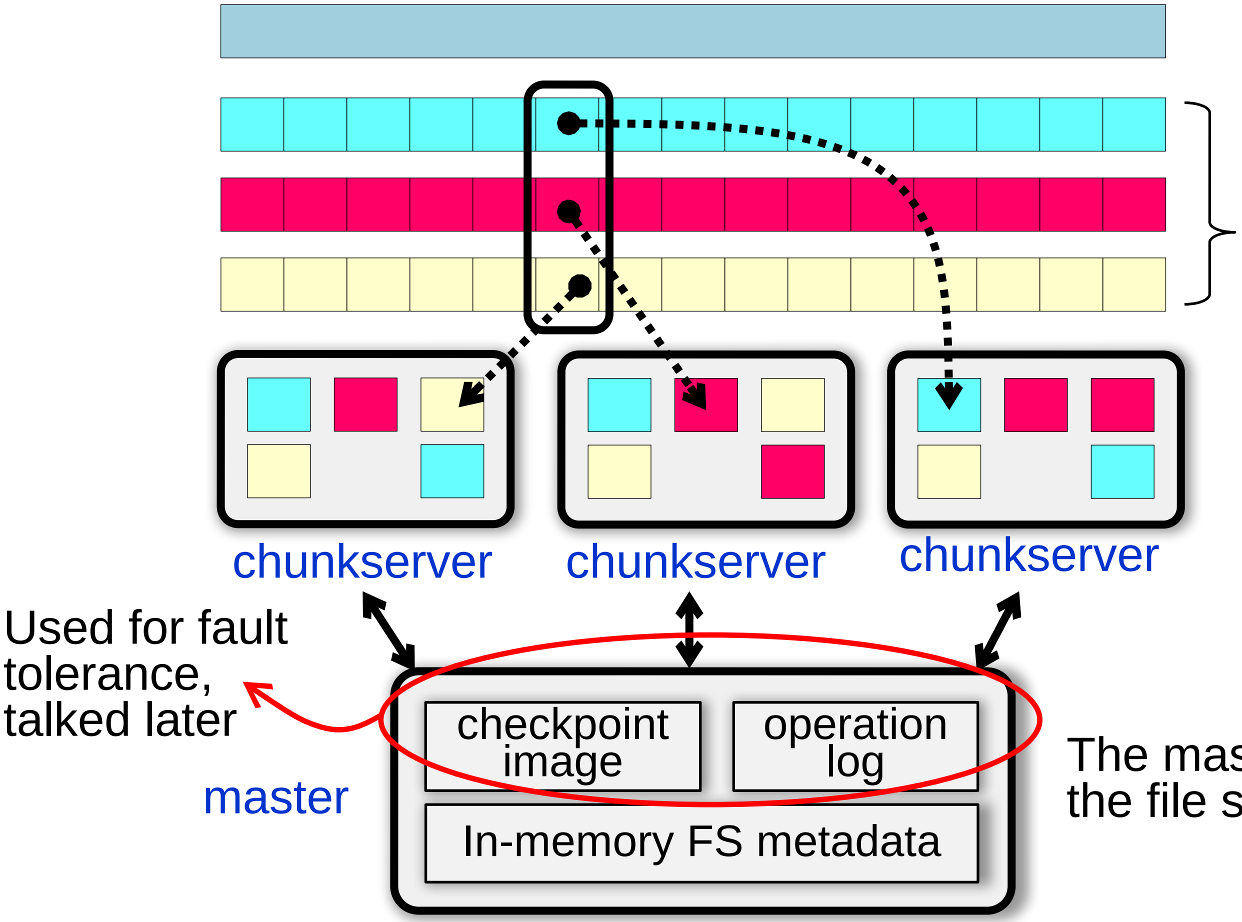
A GFS cluster = **1 Master** (Distributed inode layer) + **N Chunkservers** (Distributed blocks layer + replication)



File system **metadata**
Maps files to chunks

Data Storage: fixed-size chunks
Chunks replicated on several system

GFS files



A File is made of 64MB chunks, possibly spawn multiple machines

That are replicated for fault-tolerance

Chunks live on chunkservers

chunkserver

chunkserver

chunkserver

Used for fault tolerance, talked later

master

checkpoint image

operation log

In-memory FS metadata

The master manages the file system namespace

Chunks (Blocks) and Chunkservers

Chunk size **= 64 MB** (default)

- 32-bit checksum with each chunk

Chunk **handle**

- Globally unique 64-bit number
- Assigned by the master when creation

Chunks are stored on local disk as Linux files (aka. file system overlay)

Each chunk is **replicated** on multiple nodes

- Three replicas (default)
- More replicas for popular files to avoid **hotspots**

Why Large Chunks?

Default chunks size = 64MB

- Compare to Linux ext4 block size: 4KB~1MB

Benefits:

- **Reduce** the need for **frequent communication** with master for chunk location info
- Makes it feasible to keep a **TCP connection** open for an extended time
  - Establishing a TCP connect can be costly (two-way handshake, see later lectures)
- Master stores all **metadata** in memory

GFS Master

Maintains all file system metadata

- Access control info, filename to chunks mappings, current locations of chunks

Manages

- ~~Chunk **leases** (locks), **garbage** collection, chunk **migration**~~ (not the focus of this lecture)

Master replicates its data for fault-tolerance

- A large topic also not the focus of today’s lecture

Questions: differences from our naïve filesystem?

1. **Master stores current locations of chunks (blocks)**
    1. Unlike naïve distributed file, which store the locations in the inode block
    2. Why? For performance
2. **Master replicates its data for fault-tolerance**
    1. Our previous design does not replicate the data

GFS uses one master, why? Make the design simple

All metadata stored in master's memory

- Super-fast access

Name-to-chunk maps (e.g., using an in-memory tree)

- Stored in **memory**
- Also persist in **an operation log** on the disk (talk about in later chapters)

Don't store chunk location persistently

- This is queried from all the chunkservers at startup
- Can keep up-to-date: master controls all the management
- Benefits: simpler for consistency management

Client-GFS interaction model

GFS client code linked into each app

- No **OS-level** API
- Interacts with **master** for metadata-related ops
- Interacts **directly with chunkservers** for data
  - Master is not a point of congestion

No caching: neither clients nor chunkservers cache data

- Except for the system buffer cache

Clients cache metadata

- e.g., location of a file’s chunks

Reading a file in GFS (very similar to the naïve DFS)

Reading a file is simple in GFS

1. Contact the **master**

2. Get file’s **metadata**: chunk handles

3. Get the **location** of each of the chunk handles
   – Multiple replicated chunkservers per chunk

4. Contact any **available** chunkserver for chunk

Writing a File in GFS (More complicated due to replication)

Less frequent than reading

- But is more complex, because we need to deal with the consistency issues
- GFS adopts a **relaxed consistency model (see later lectures)**
- E.g, may have inconsistency state, but work well for their apps
- Benefits: simple & efficient to implement

Master grants a chunk lease to one of the replicas

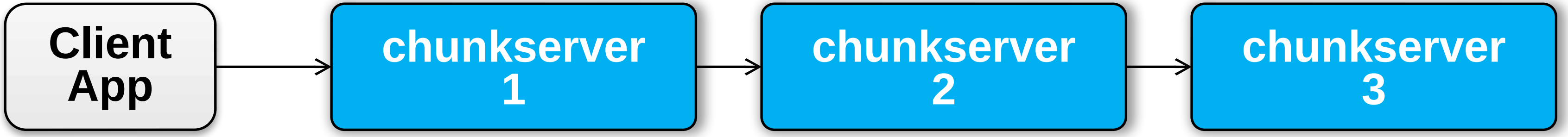
- This replica will be the **primary** chunkserver
  - The only one that can modify the chunk
- Primary can request extensions (of lease), if needed
  - Master increases the chunk version number and informs replicas

Writing a File in GFS: Two-phases

Phase 1: send data

Deliver data but **don't write** to the file

- A client is given a list of replicas
    - Identifying the primary and secondaries
- Client writes to the closest replica
    - Pipeline forwarding
- Chunkservers store this data in a cache (in memory)

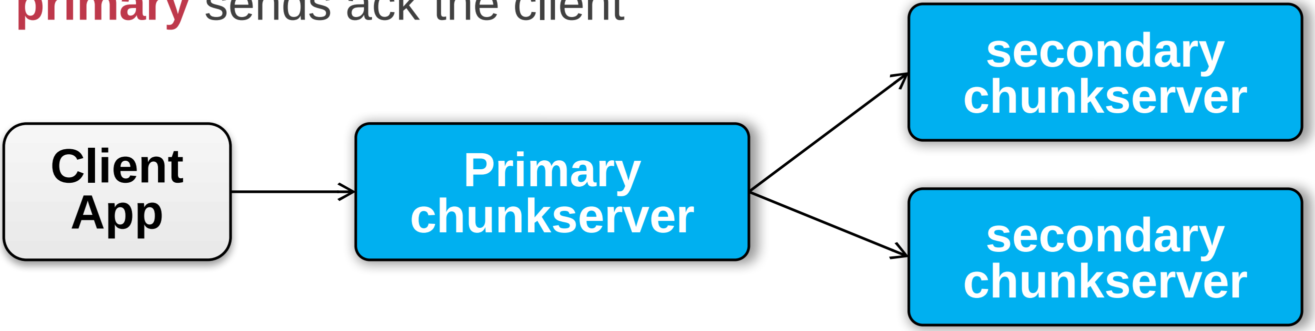


Writing a File in GFS: Two-phases

Phase 2: write data

Add the data to the file (commit)

- Client waits for replicas' ack of receiving data
- Send a **write** request to the **primary**
- The **primary** is responsible for serialization of writes (applying then forwarding)
- Once all acks have been received
  - ✉ The **primary** sends ack the client



Writing a File in GFS: Two-phases

Data flow (phase 1) is different from **control flow** (phase 2)

Data flow

- Client ✉ chunkserver ✉ chunkserver ✉ ...
- Order does not matter

Control flow

- Client ✉ primary ✉ all secondaries
- Order maintained (also for concurrent writes from multiple clients)

Chunk version numbers are used to detect if any replica has stale data

- Is maintained by the primary chunkserver
- If a replica has stale data, it shall be replaced

Naming in GFS: simple flat naming

No per-directory data structure like most file systems

- E.g., directory file contains names of all files in the directory

No aliases (i.e., no hard or symbolic links)

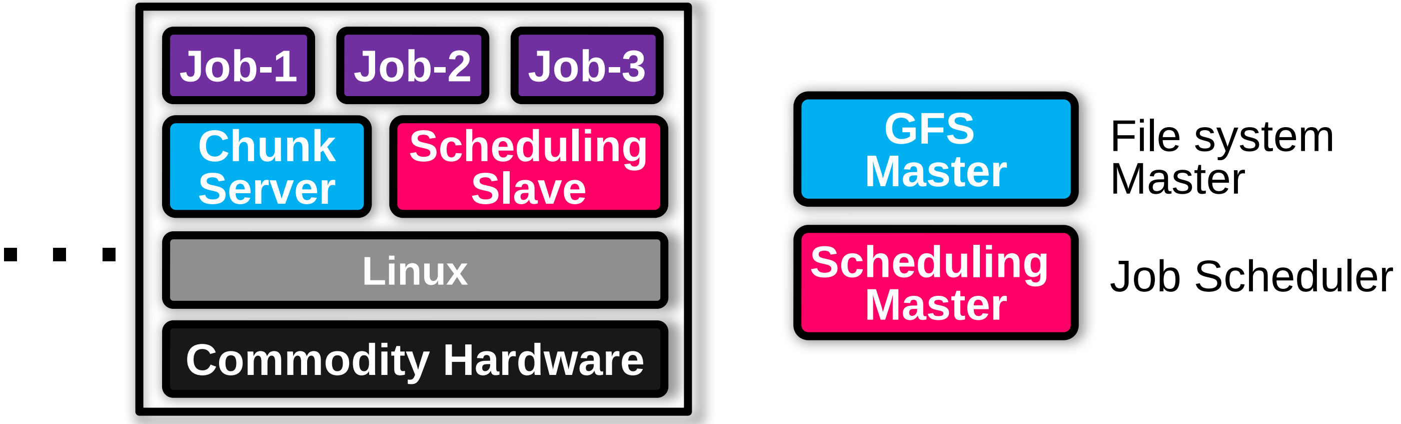
Namespace is a single lookup table

- Maps pathnames to metadata

Summary: GFS in Google cluster

Google cluster environment

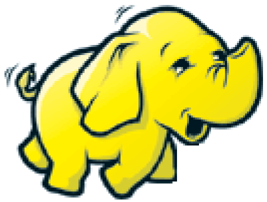
- Core services: GFS + cluster scheduling system
- Typically, 100s to 1000s of active jobs
- 200+ clusters, many with 1000s of machines
- Pools of 1000s of clients
- 4+ PB filesystems, 40GB/s read/write loads



HDFS: another popular (open-source) DFS

Hadoop **D**istributed **FS**

Primary storage system for **Hadoop** apps



A framework that allows for the distributed processing of large data sets across clusters of computers

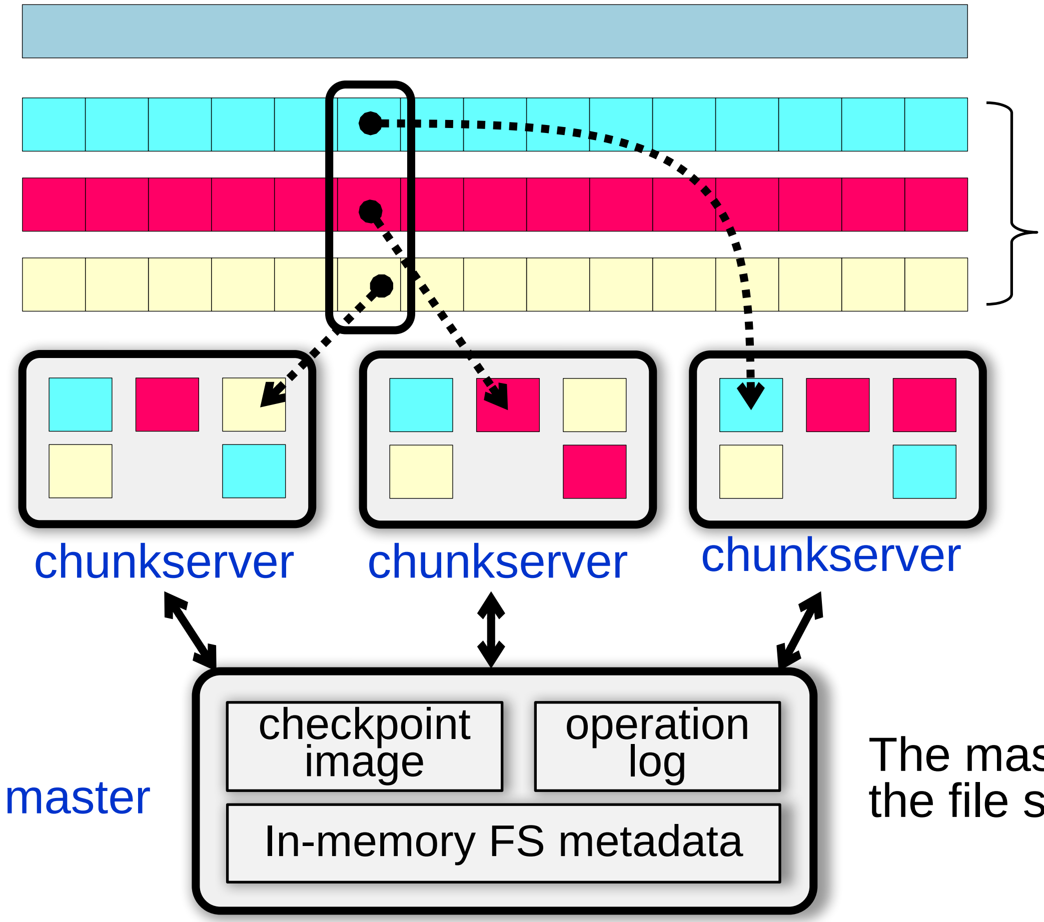
Design Goals & Assumptions of HDFS

HDFS is an open source (Apache) implementation inspired by GFS design

Similar goals as GFS

- Run on commodity hardware
- Highly fault tolerant
- High throughput & large-scale deployments
- …

Recall GFS files



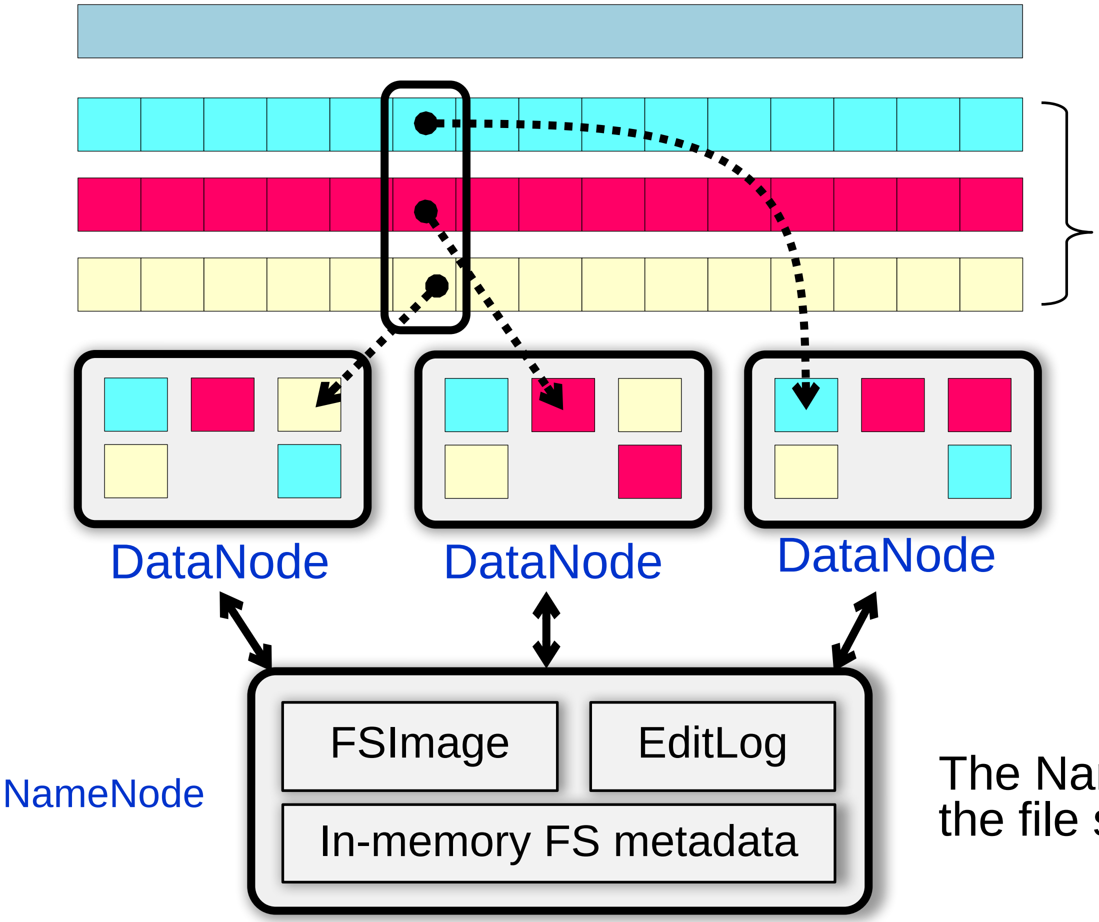
A File is made of 64MB chunks, possibly spawn multiple machines

That are replicated for fault-tolerance

Chunks live on chunkservers

The master manages the file system metadata

HDFS files



DataNode

DataNode

DataNode

NameNode

FSImage

EditLog

In-memory FS metadata

A File is made of 64MB blocks, possibly spawn multiple machines

That are replicated for fault-tolerance

Blocks live on DataNodes

The NameNode manages the file system metadata

Summary

Designing distributed file system (DFS) is not simply

- Single-node file system + RPC

Many design choices for performance, consistency model & failure handling

- Interface
- Caching
- Access model

Two case studies

- **NFS**: transparent access files on a remote server
- **GFS**: highly-scalable & fault-tolerant DFS optimized for Google’s workload

GFS or NFS are not Perfect

NFS

- Can **not scale**
- Is **not fault-tolerant**
- But is well-enough for many workloads, e.g., sharing the data for experiments in our lab ⏪

GFS

- **Relaxed consistency model**: the results of concurrent mutations are undefined
- Single-node master: single point of failures (though next-generation of GFS refines this, with more advanced techniques developed later)
- Work well in Google’s datacenter workloads

GFS or NFS are not Perfect

NFS

- Can **not scale**
- Is **not fault-tolerant**
- But is well-enough for many workloads, e.g., sharing the data for experiments in the lab

GFS

- **Relaxed consistency model**: the results of concurrent mutations are undefined
- Single-node master: single point of failures (though next-generation of GFS refines this)
- Work well in Google's datacenter workloads

We will see system principles to cope with them in future lectures.

For other workloads (e.g., Database) may not sufficient.