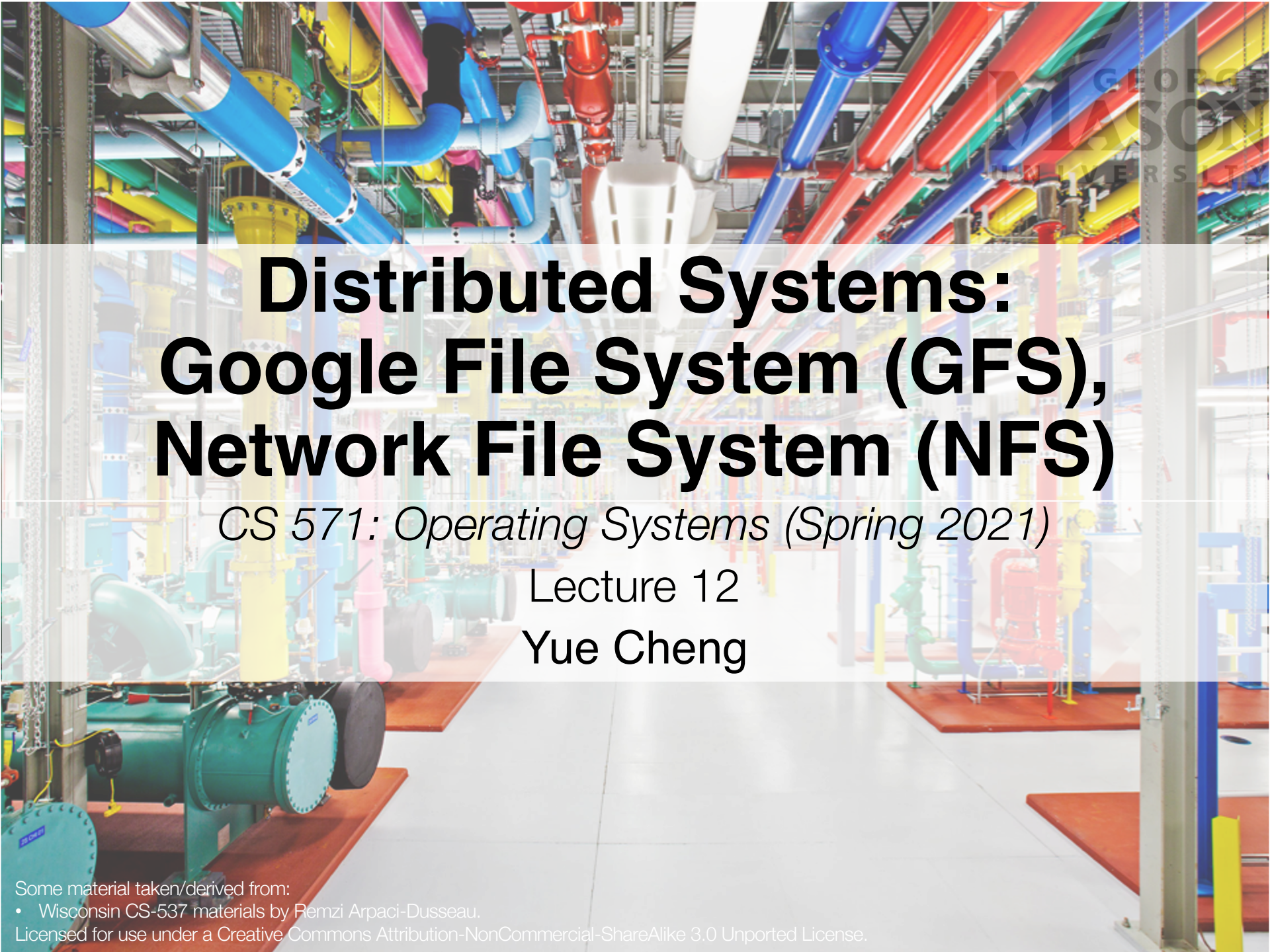


Distributed Systems: Google File System (GFS), Network File System (NFS)

CS 571: Operating Systems (Spring 2021)

Lecture 12

Yue Cheng

Some material taken/derived from:

- Wisconsin CS-537 materials by Remzi Arpaci-Dusseau.

Google File System (GFS)

MapReduce assumptions

- Commodity hardware
  - Economies of scale!
  - Commodity networking with less bisection bandwidth
  - Commodity storage (hard disks) is cheap
- Failures are common
- Replicated, distributed file system for data storage

Fault tolerance

- If a task crashes:
  - Retry on another node
    - Why this is okay?
  - If the same task repeatedly fails, end the job

Fault tolerance

- If a task crashes:
  - Retry on another node
    - Why this is okay?
  - If the same task repeatedly fails, end the job

- If a node crashes:
  - Relaunch its current tasks on another node
    - What about task inputs?

Google file system (GFS)

- Goal: a global (distributed) file system that stores data across many machines
  - Need to handle 100's TBs

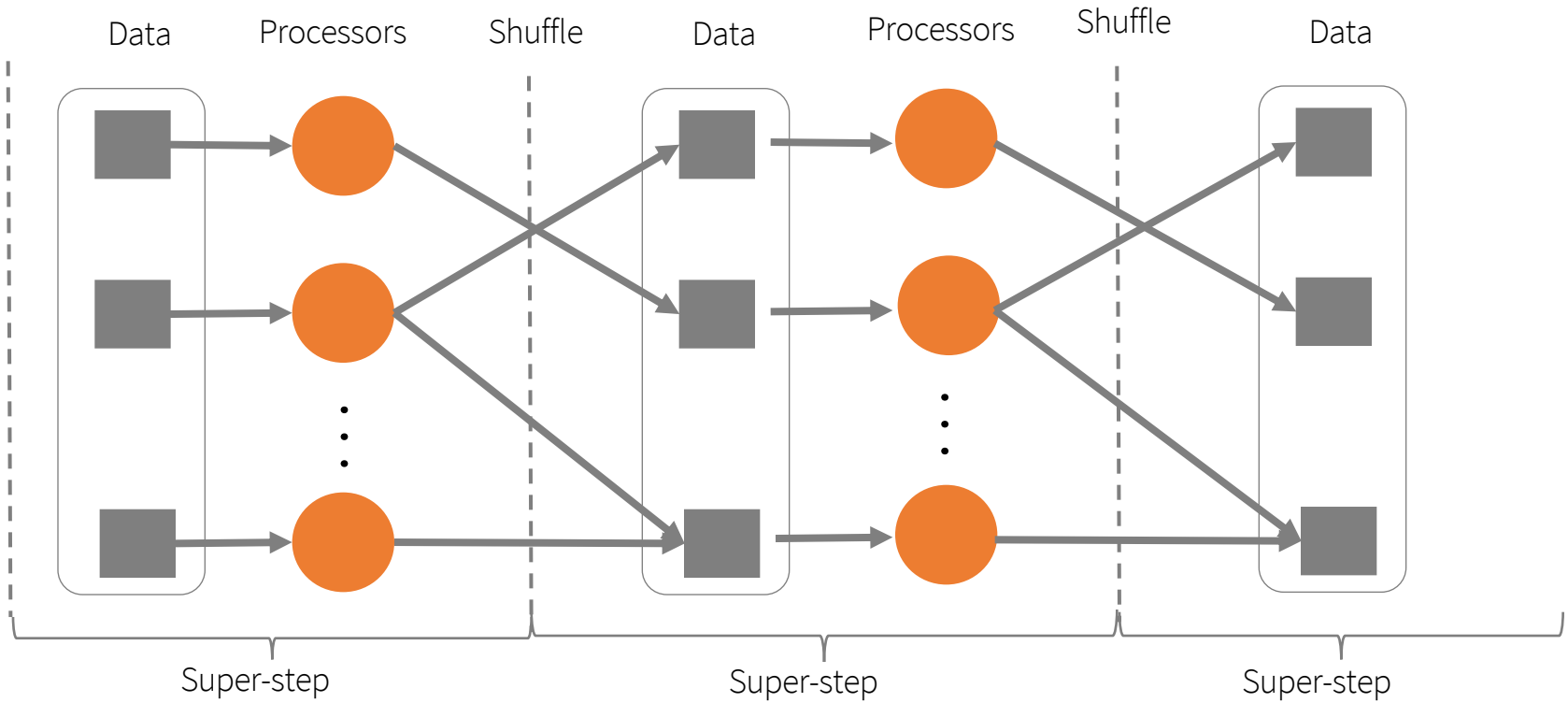
- Google published details in 2003

- Open source implementation:
  - Hadoop Distributed File System (HDFS)

Workload-driven design

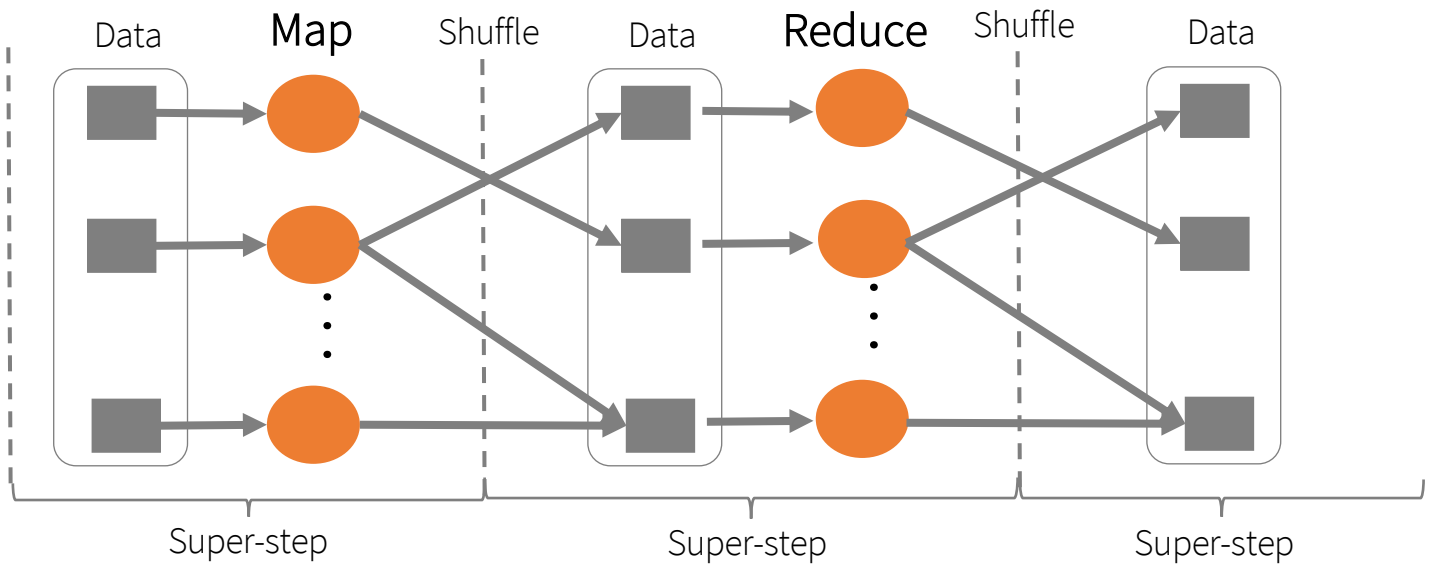
- MapReduce workload characteristics
  - Huge files (GBs)
  - Almost all writes are appends
  - Concurrent appends common
  - High throughput is valuable
  - Low latency is not

Example workloads: Bulk Synchronous Processing (BSP)



*Leslie G. Valiant, A bridging model for parallel computation, Communications of the ACM, Volume 33 Issue 8, Aug. 1990

MapReduce as a BSP system



- Read entire dataset, do computation over it
  - Batch processing

- Producer/consumer: many producers append work to file concurrently; one consumer reads and does work

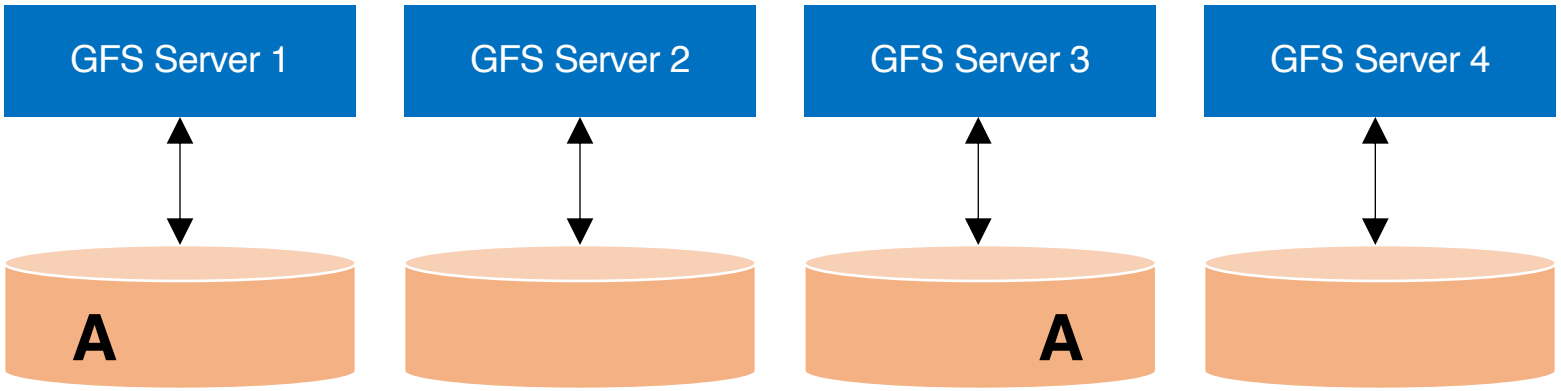
Workload-driven design

- Build a global (distributed) file system that incorporates all these application properties

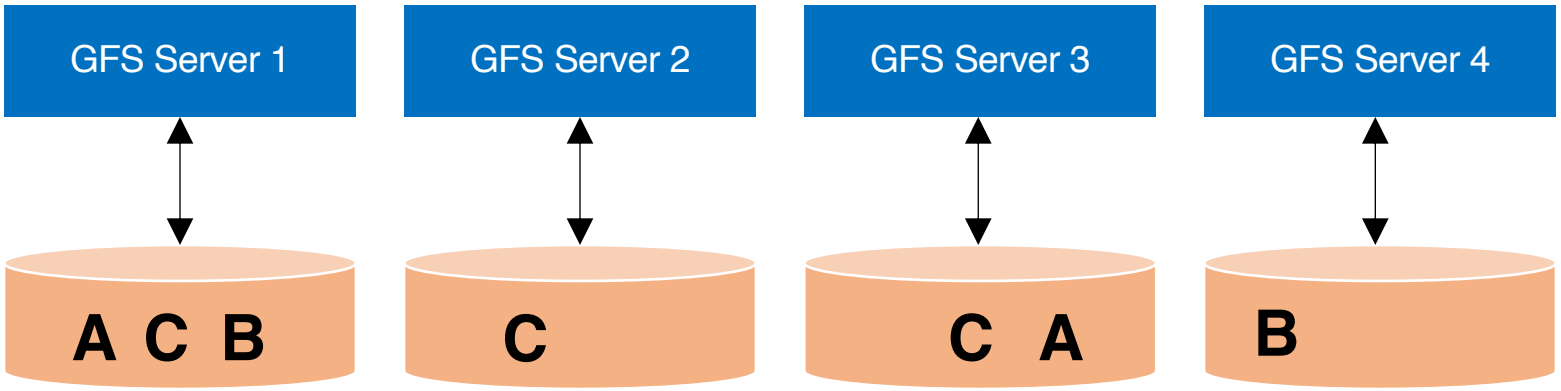
- Only supports features required by applications

- Avoid difficult local file system features, e.g.:
  - rename dir
  - links

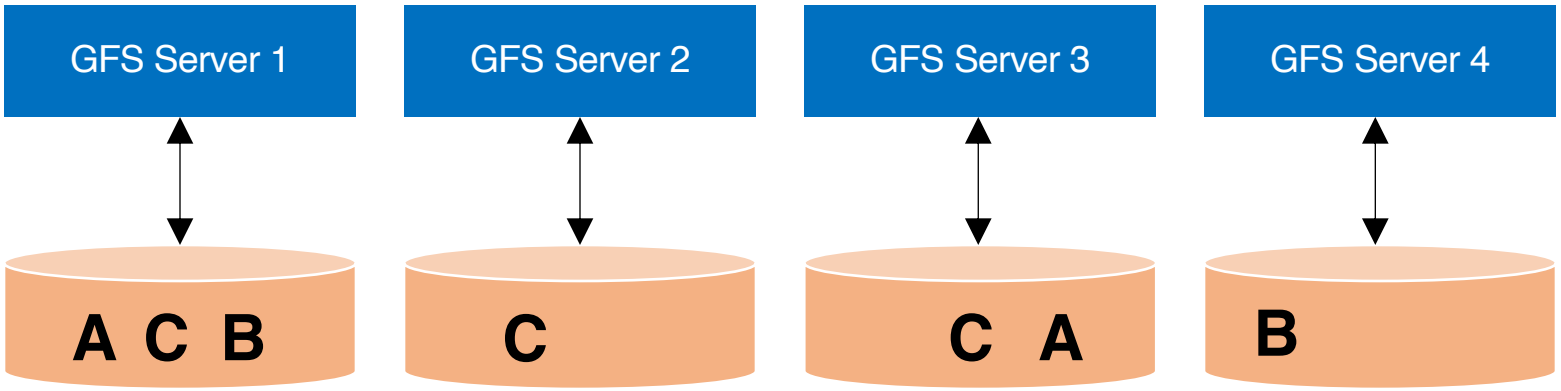
Replication



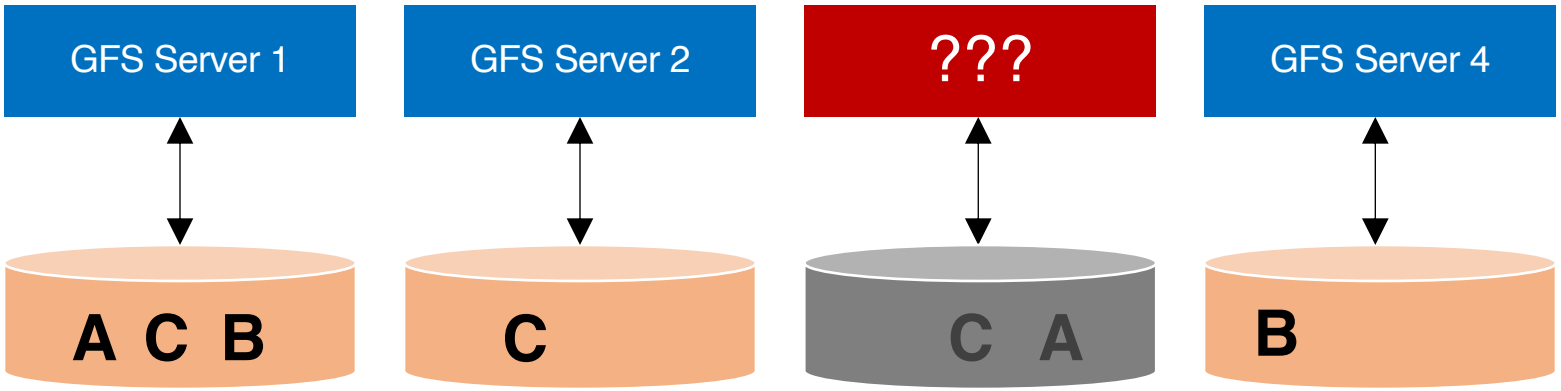
Replication



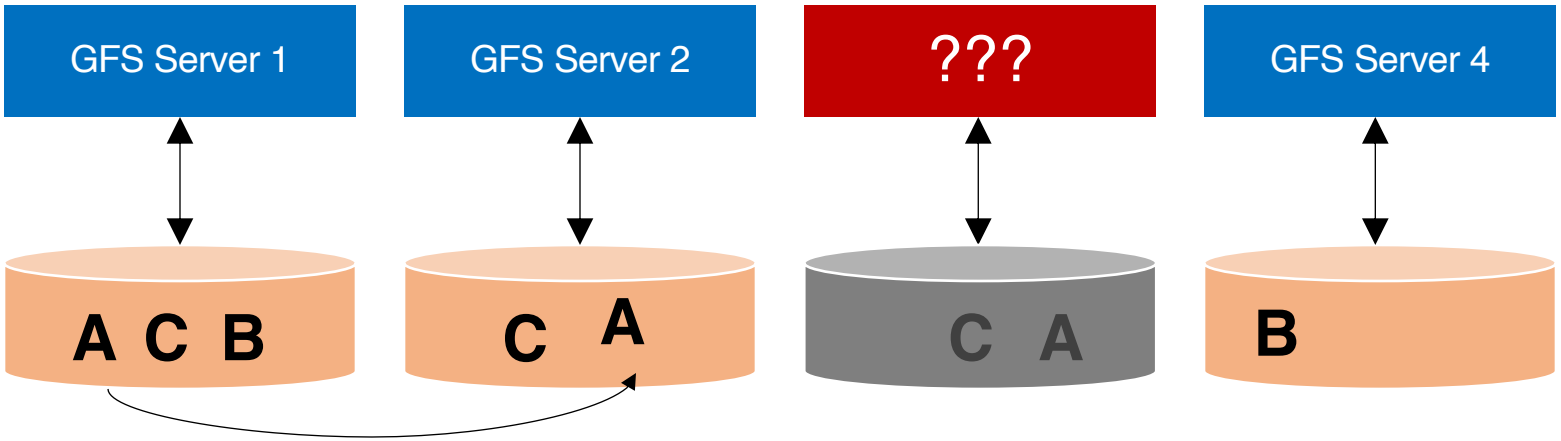
Resilience against failures



Resilience against failures

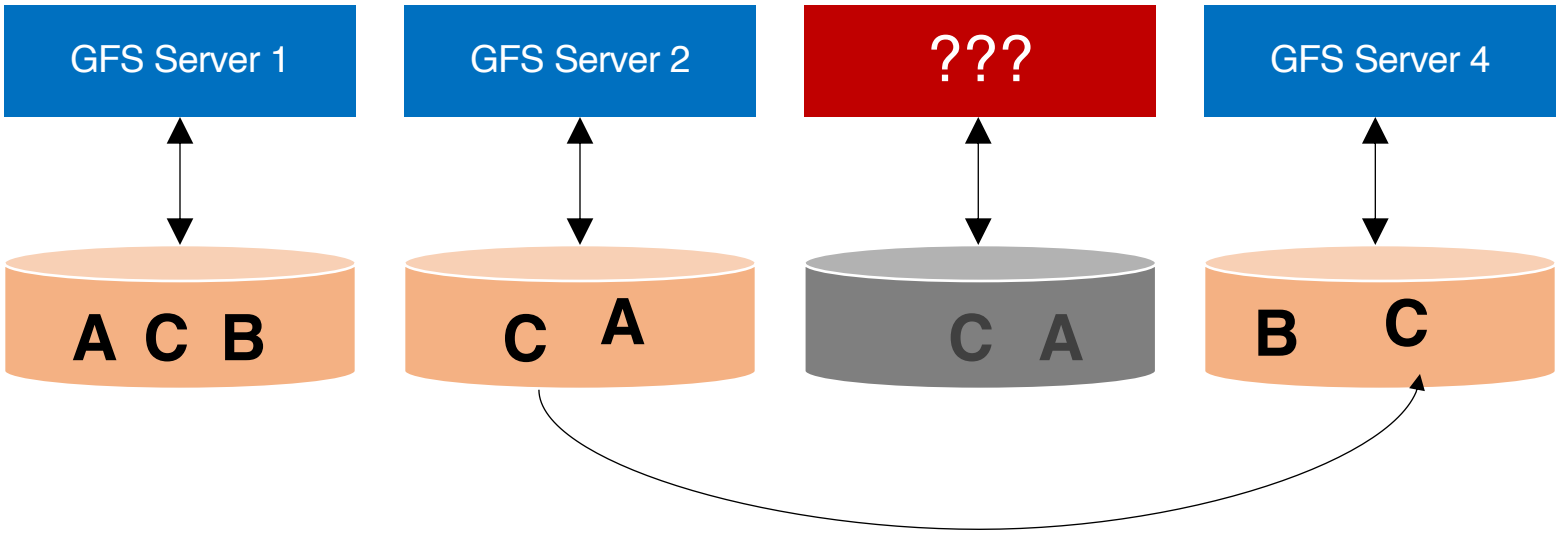


Data recovery



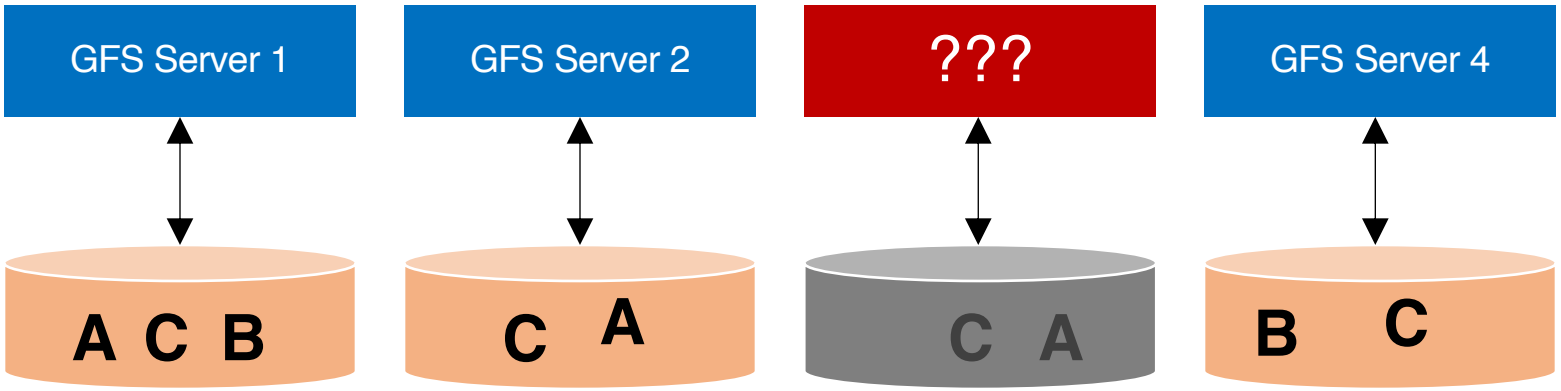
Replicating A to maintain a replication factor of 2

Data recovery



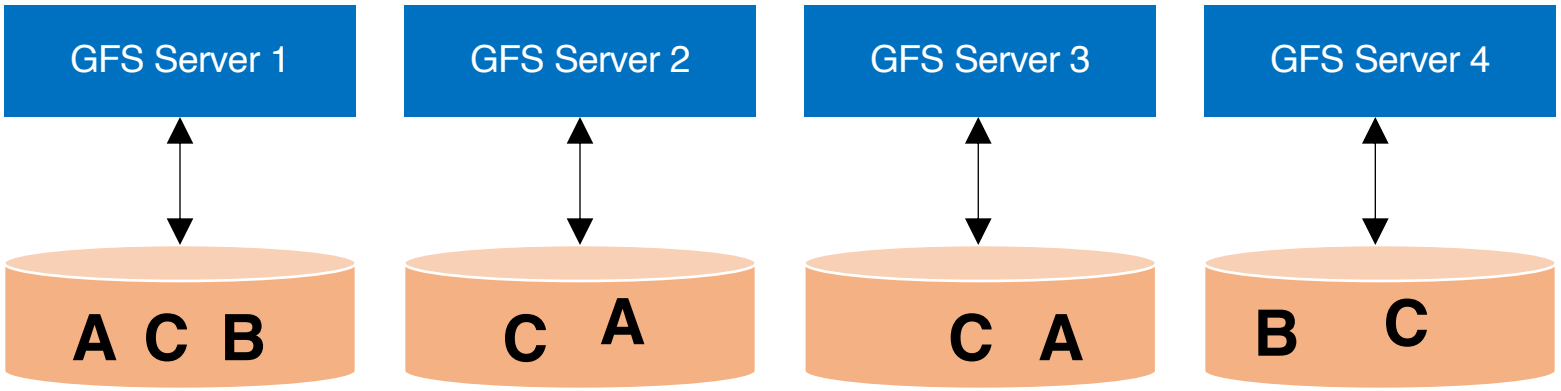
Replicating C to maintain a replication factor of 3

Data recovery



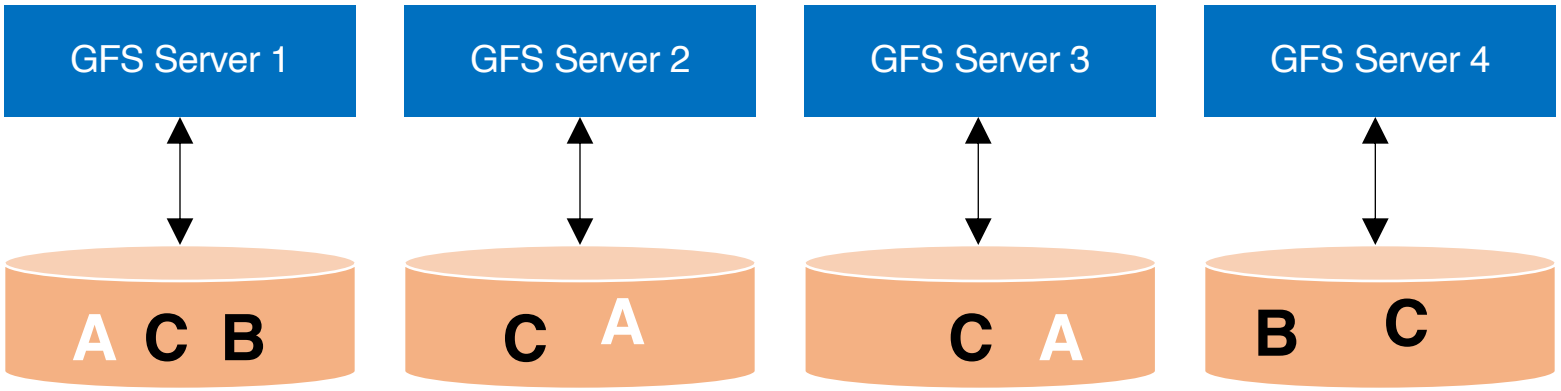
Machine may be dead forever, or it may come back

Data recovery

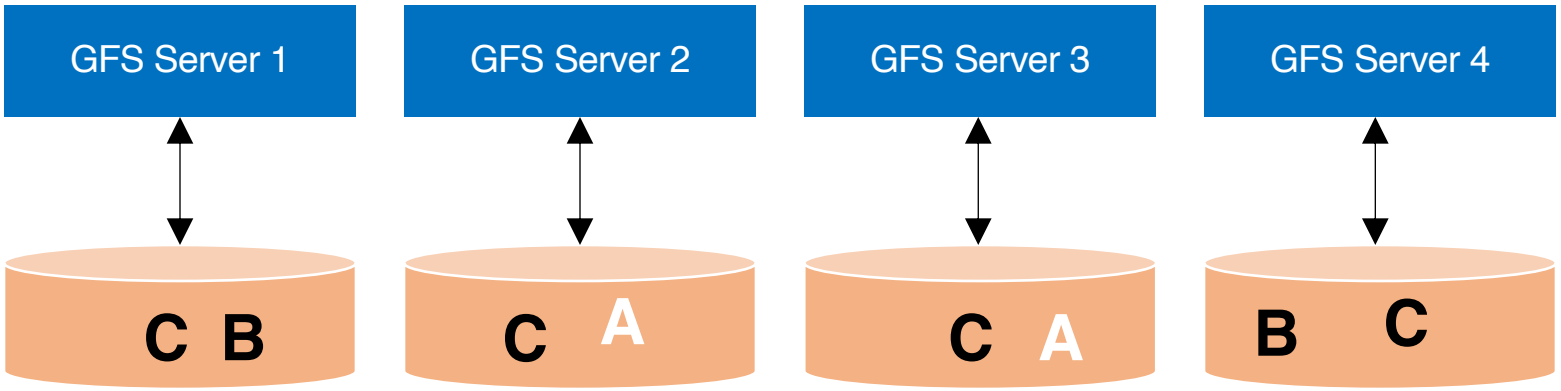


Machine may be dead forever, or it may come back

Data recovery



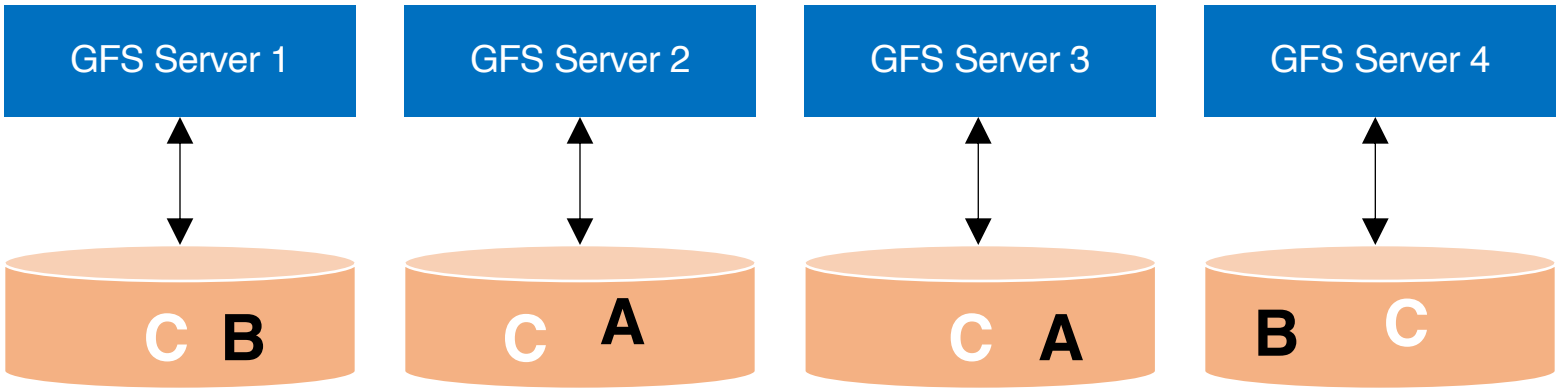
Data recovery



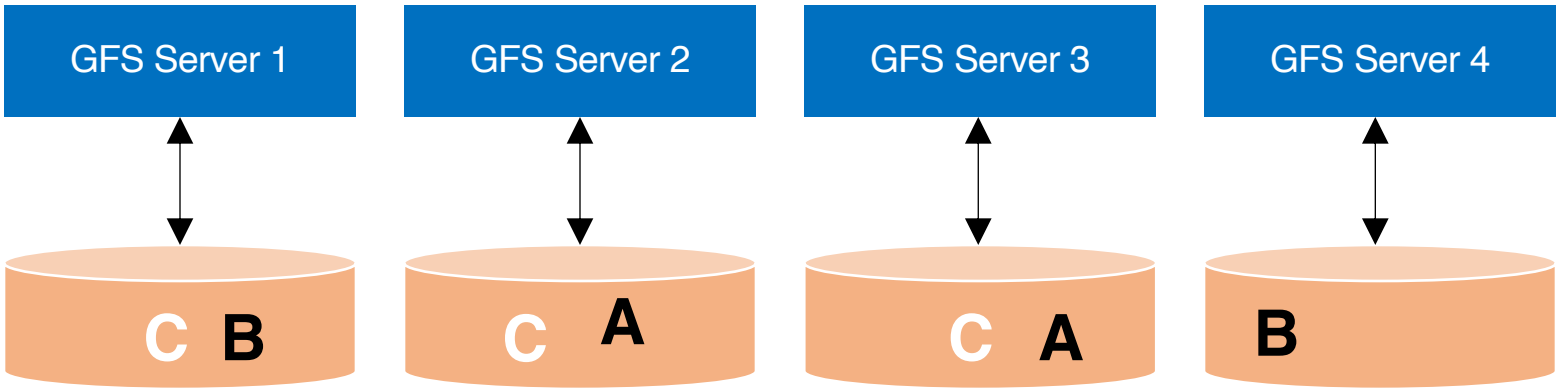
Data Rebalancing

Deleting one A to maintain a replication factor of 2

Data recovery



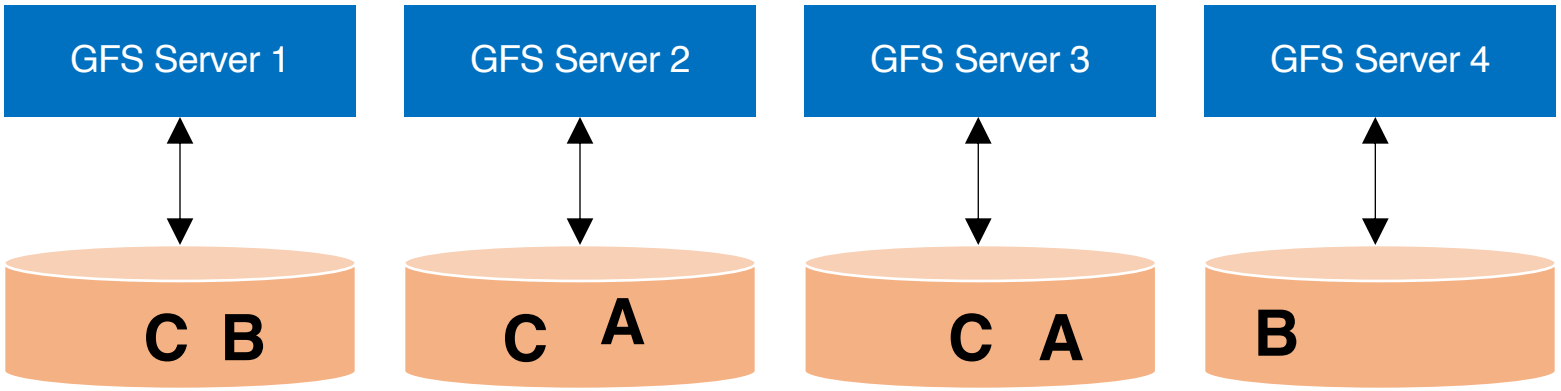
Data recovery



Data Rebalancing

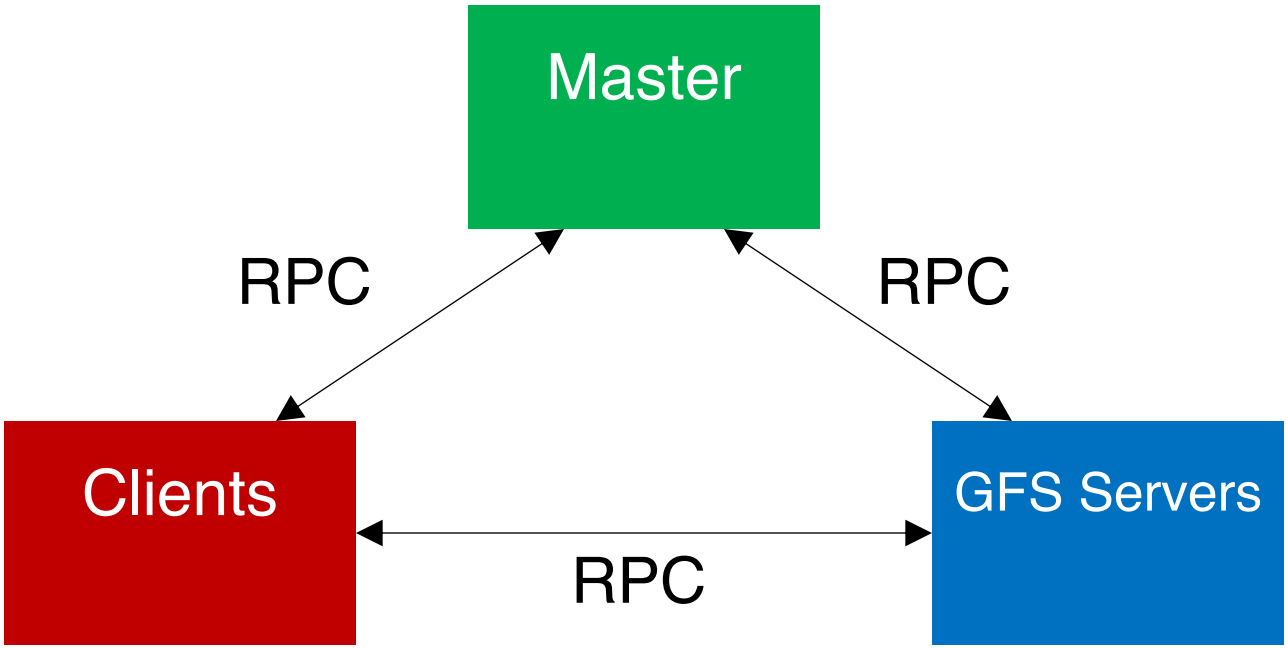
Deleting one C to maintain a replication factor of 3

Data recovery

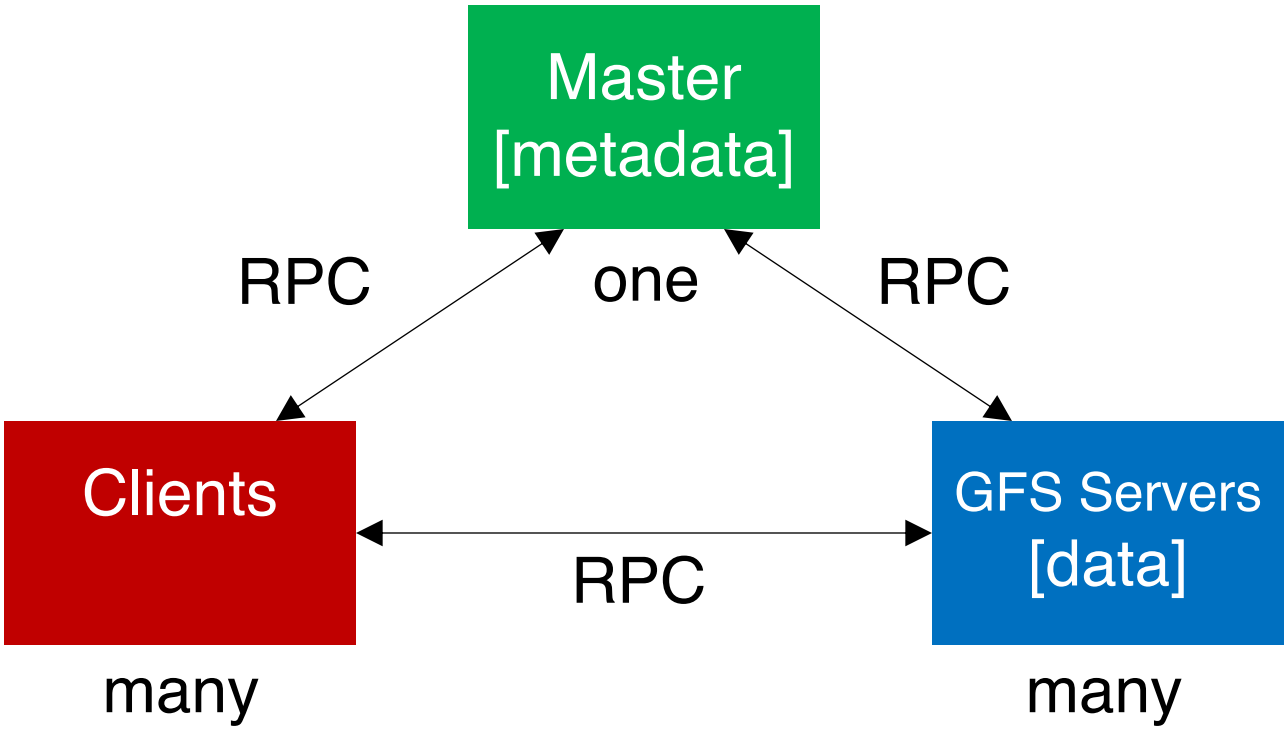


Question: how to maintain a global view of all data distributed across machines?

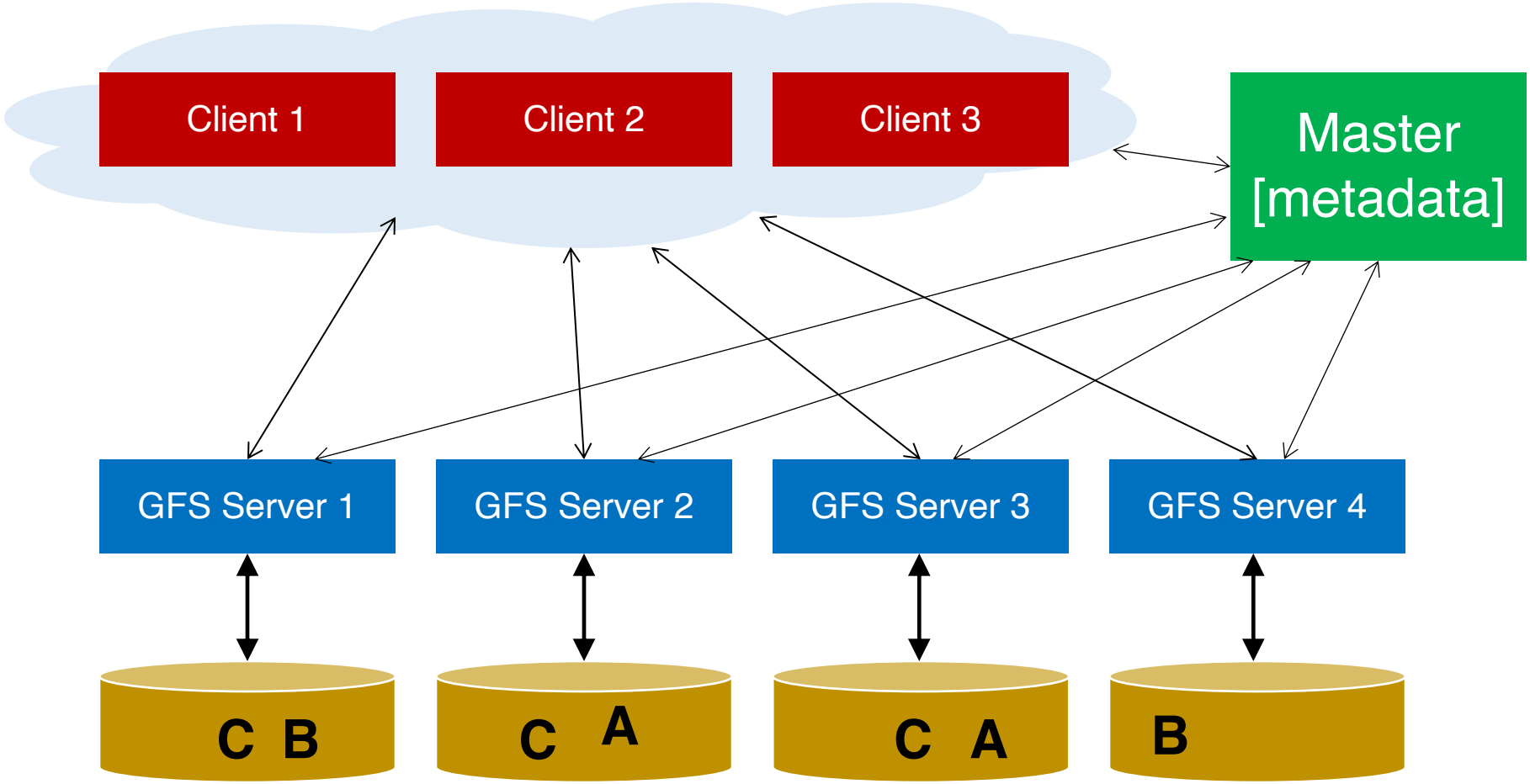
GFS architecture



GFS architecture



GFS architecture



Data chunks

- Break large GFS files into coarse-grained data chunks (e.g., 64MB)
- GFS servers store physical data chunks in local Linux file system
- Centralized master keeps track of mapping between logical and physical chunks

Chunk map

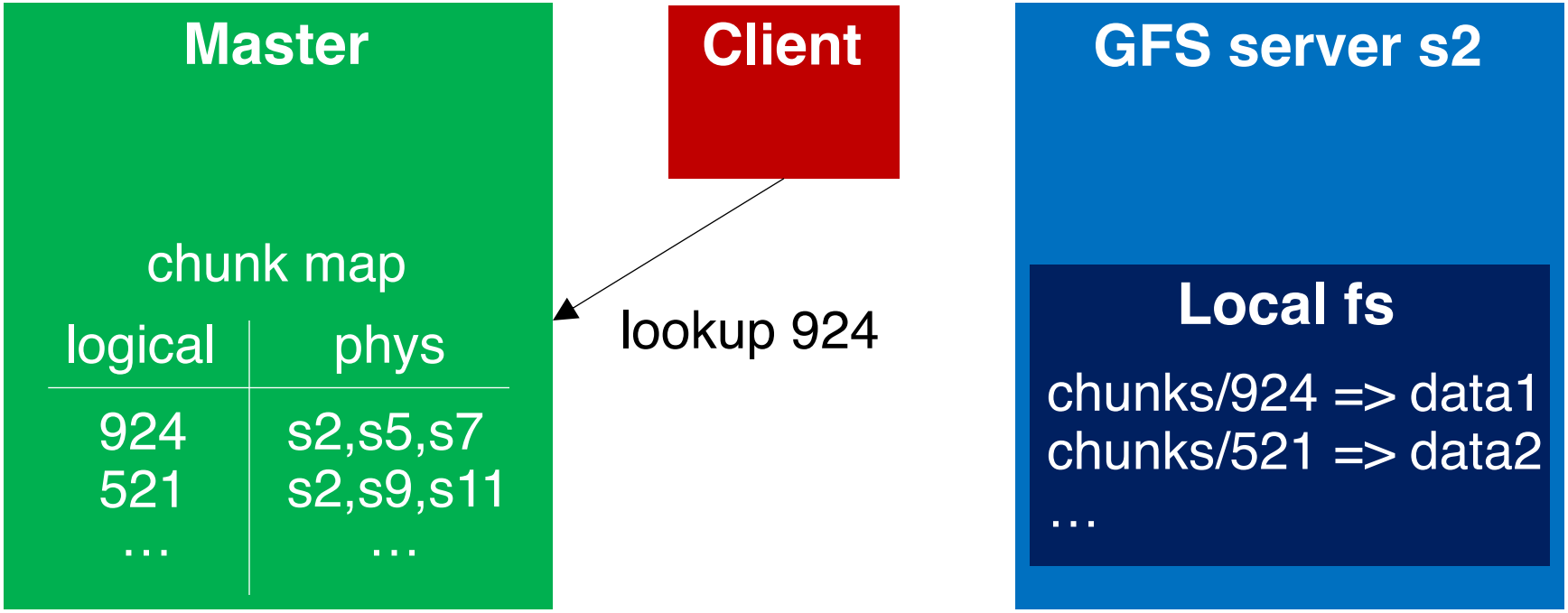
Master

chunk map

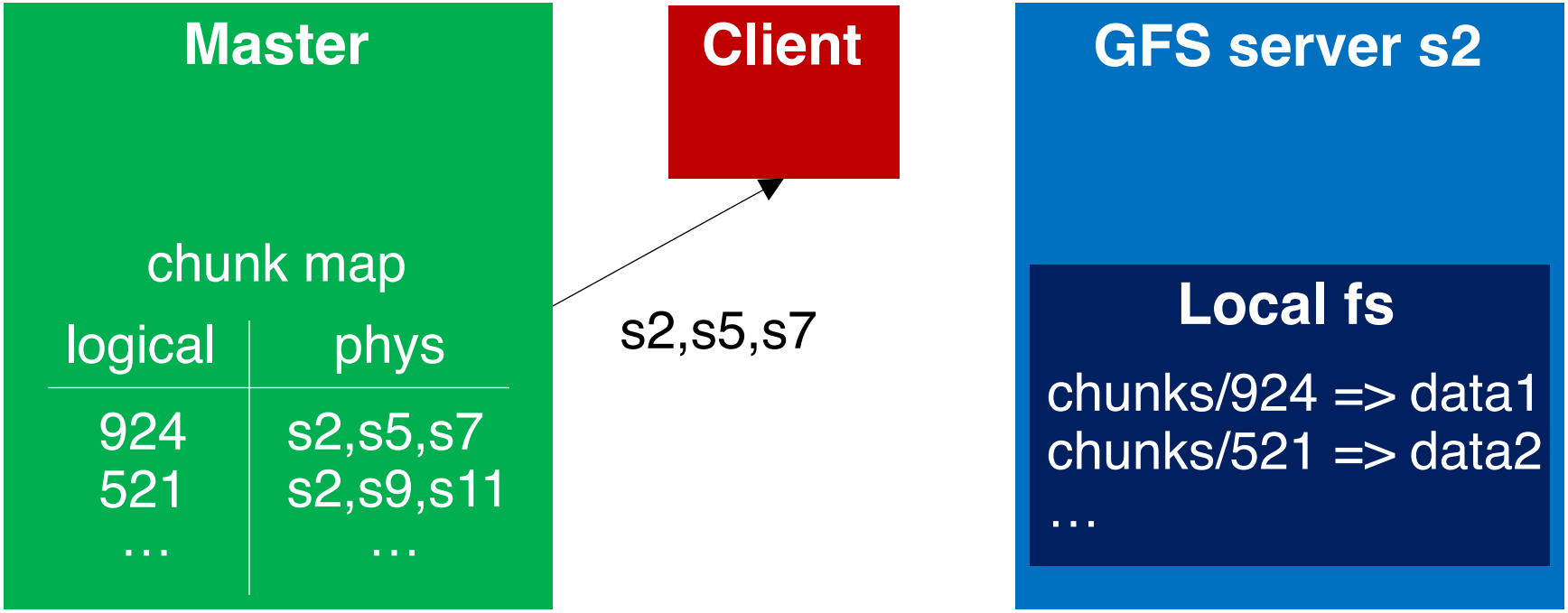
logical	phys
924	s2,s5,s7
521	s2,s9,s11
…	…

GFS server s2

Client reads a chunk



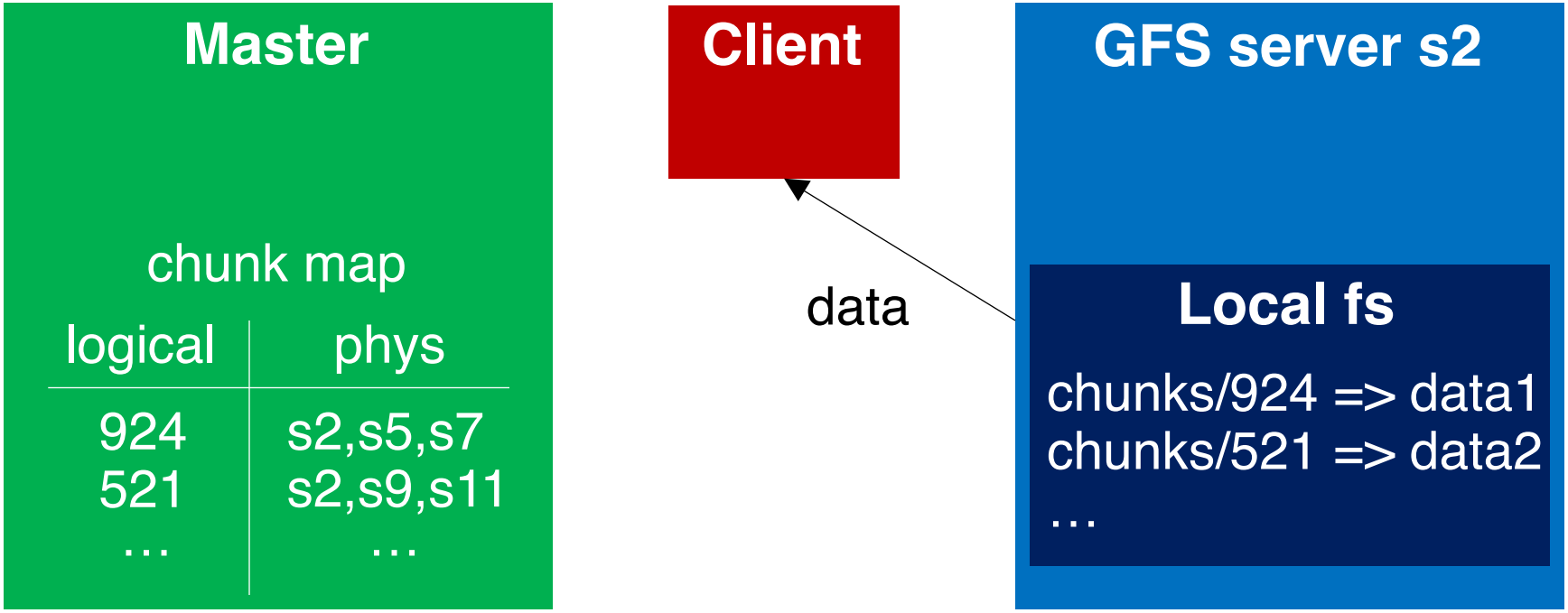
Client reads a chunk



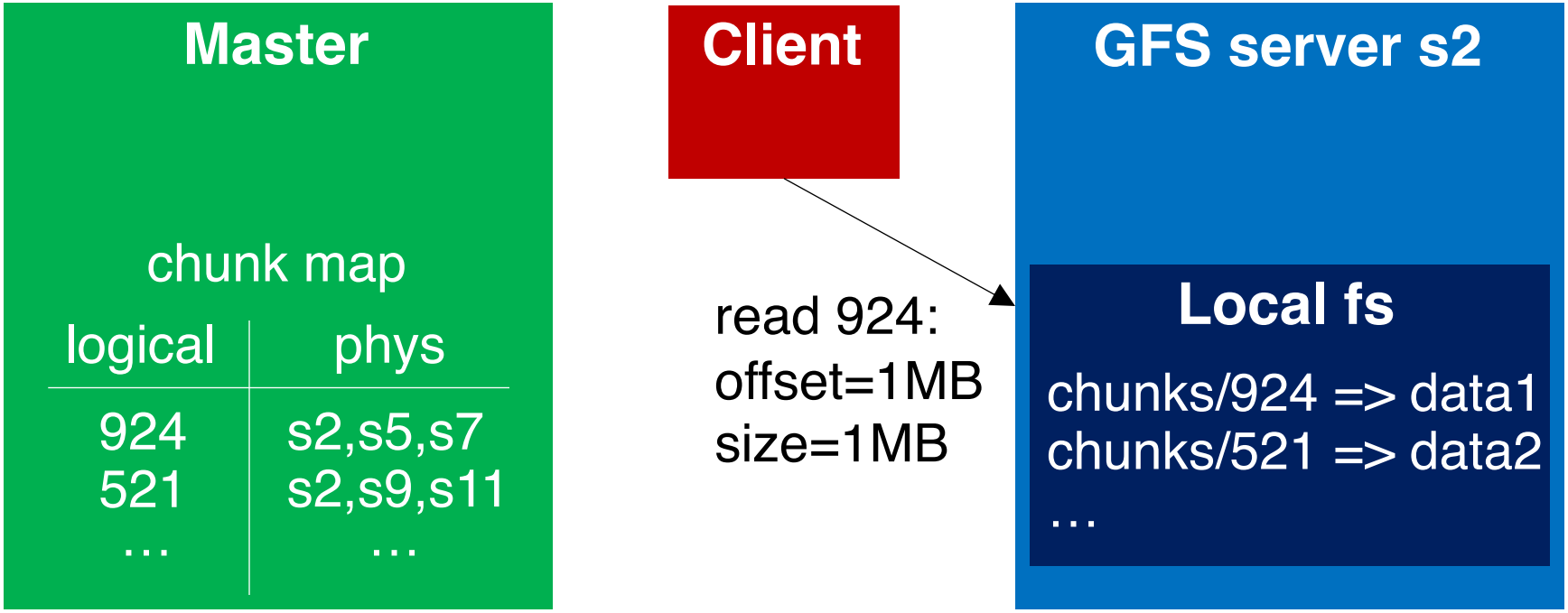
Client reads a chunk

Client reads a chunk

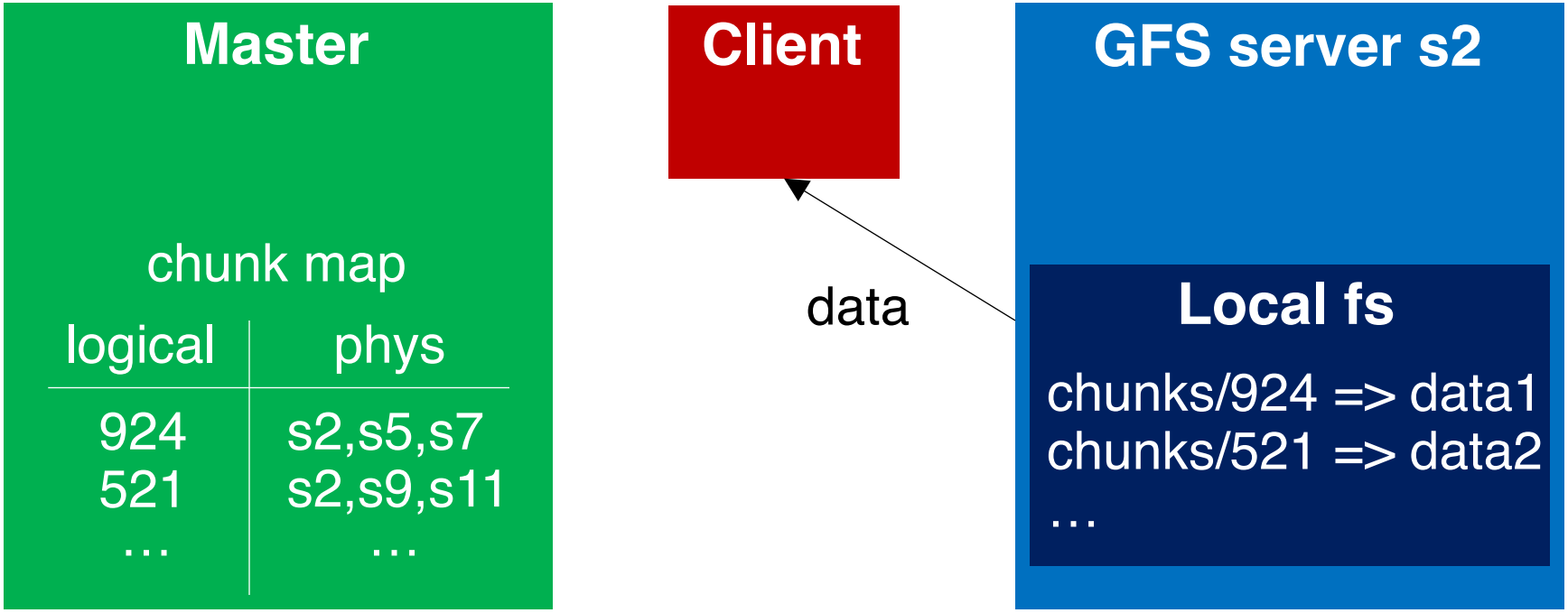
Client reads a chunk



Client reads a chunk



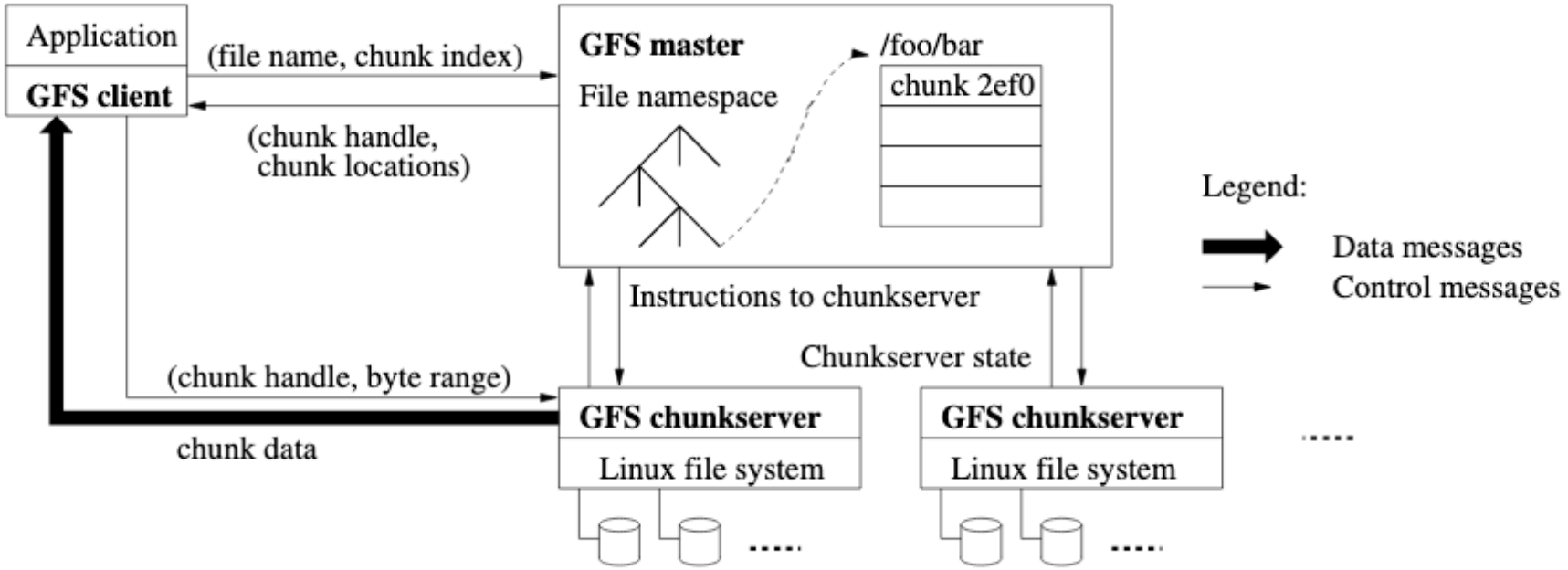
Client reads a chunk



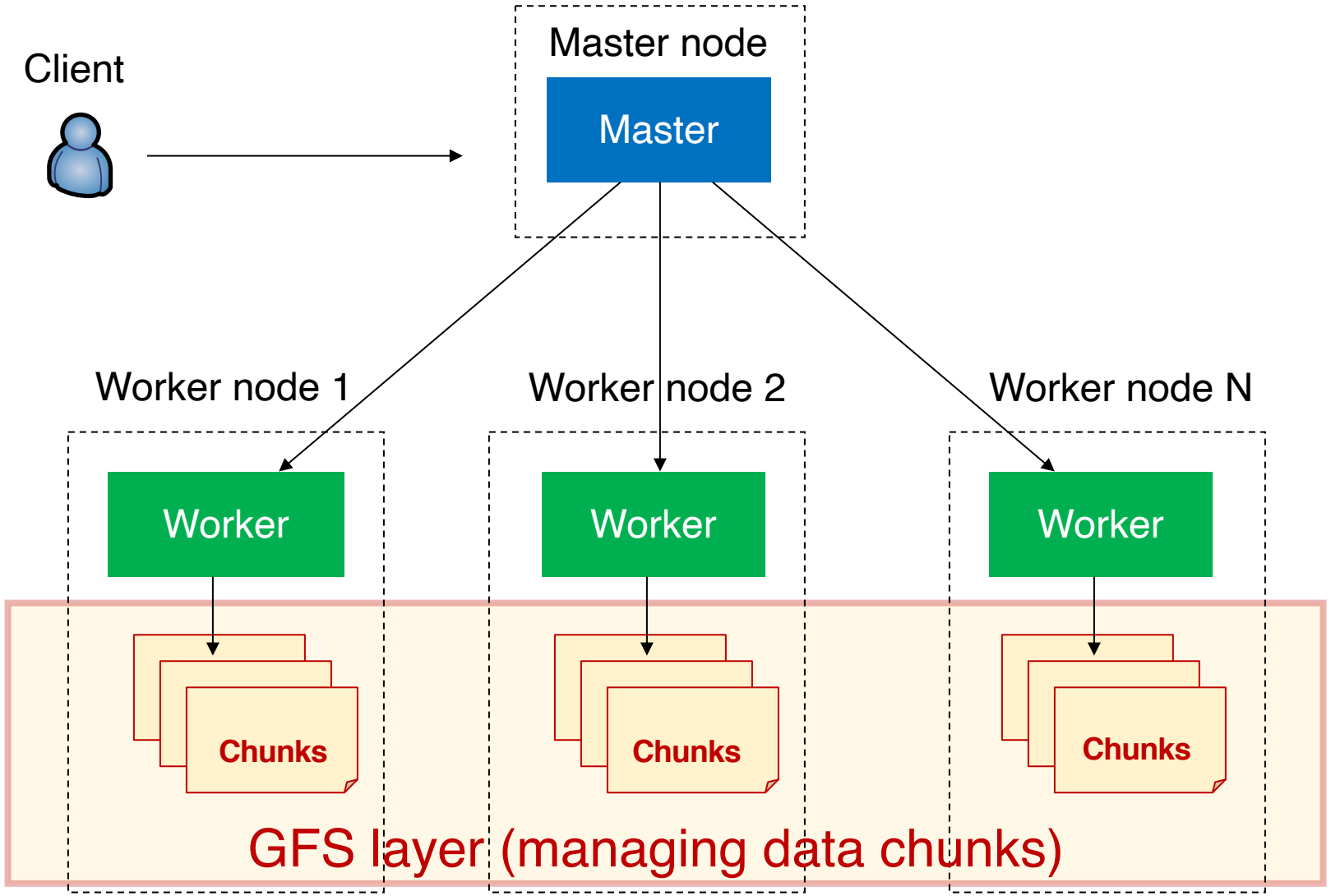
File namespace

path names mapped to logical names

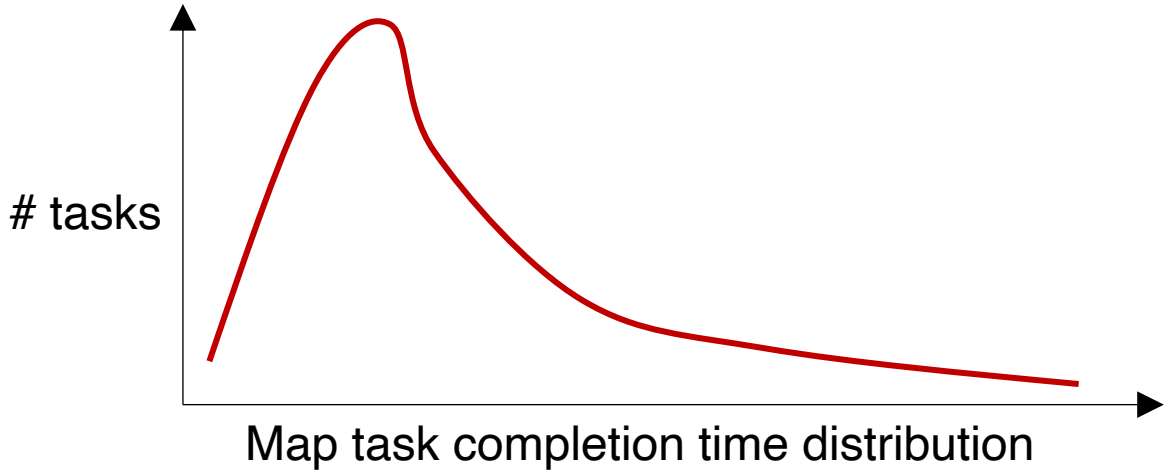
GFS architecture (original paper)



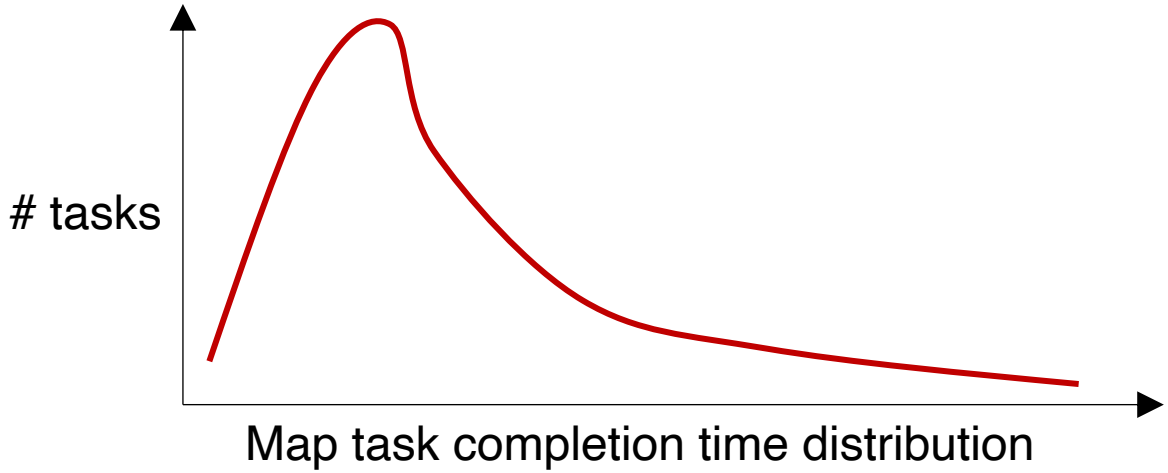
MapReduce+GFS: Put everything together



Stragglers



Stragglers



- Tail latency means some workers (always) finish late

- Q: How can MapReduce work around this?
  - **Hint**: its approach to **fault-tolerance** provides the right tool

Resilience against stragglers

- If a task is going slowly (i.e., straggler):
  - Launch second copy of task on another node
  - Take the output of whichever finishes first

More design

- Master failure
- Locality
- Task granularity

GFS usage at Google

- 200+ clusters
- Many clusters of 1000s of machines
- Pools of 1000s of clients
- 4+ PB filesystems
- 40 GB/s read/write load
  - In the presence of frequent hardware failures

* Jeff Dean, LADIS 2009

MapReduce usage statistics over time

	Aug, '04	Mar, '06	Sep, '07	Sep, '09
Number of jobs	29K	171K	2,217K	3,467K
Average completion time (secs)	634	874	395	475
Machine years used	217	2,002	11,081	25,562
Input data read (TB)	3,288	52,254	403,152	544,130
Intermediate data (TB)	758	6,743	34,774	90,120
Output data written (TB)	193	2,970	14,018	57,520
Average worker machines	157	268	394	488

* Jeff Dean, LADIS 2009

MapReduce discussion

- What will likely serve as a performance bottleneck for Google's MapReduce back in 2004 (or even earlier)? CPU? Memory? Disk? Network? Anything else?

MapReduce discussion

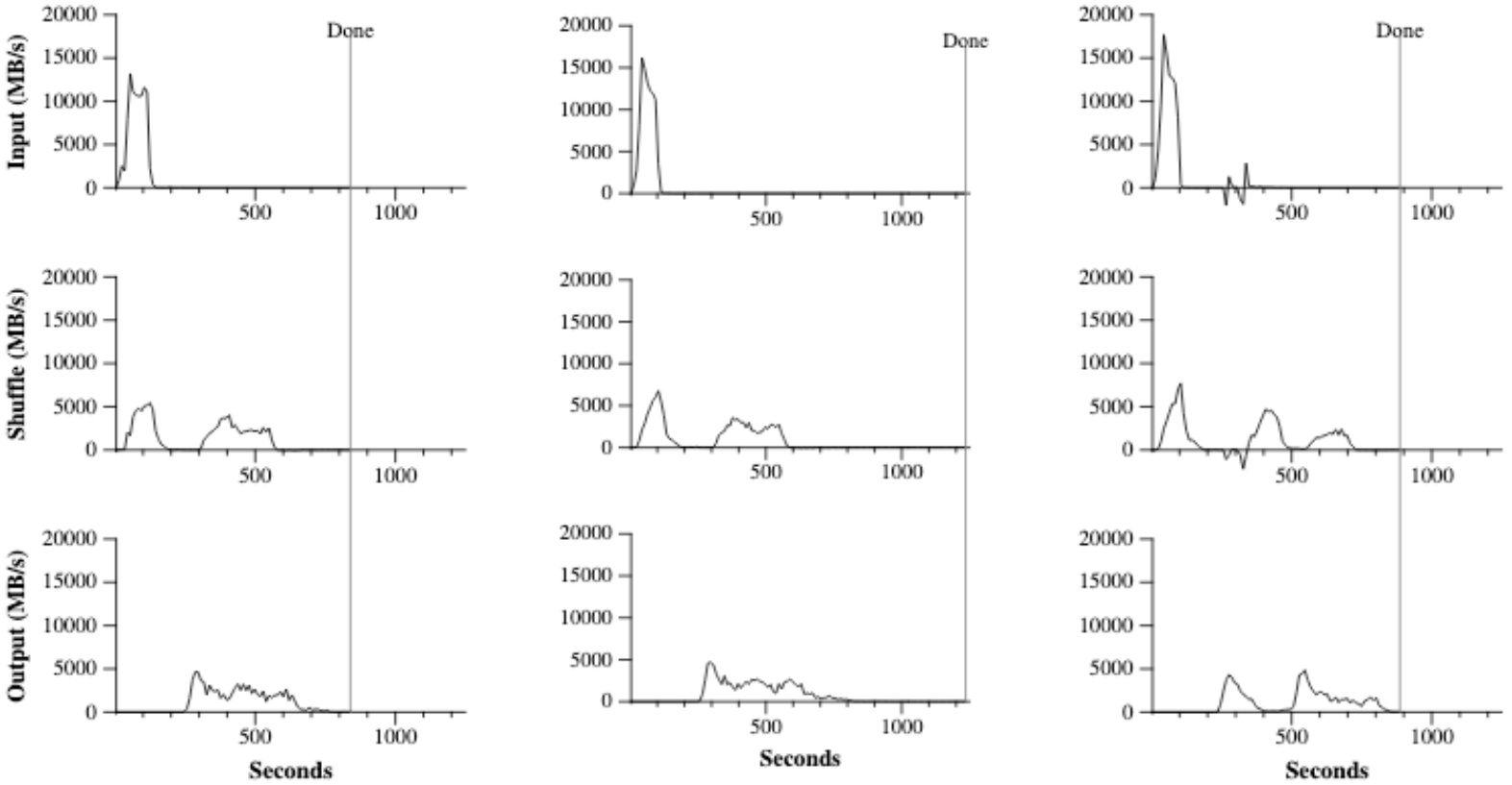
- What will likely serve as a performance bottleneck for Google's MapReduce back in 2004 (or even earlier)? CPU? Memory? Disk? Network? Anything else?

- How does MapReduce reduce the effect of slow network?

MapReduce discussion

- How does MapReduce jobs get good load balance across worker machines?

MapReduce discussion



(a) Normal execution

(b) No backup tasks

(c) 200 tasks killed

Network File System (NFS)

Primary goal

- Local FS: processes on same machine access shared files

- Network FS: processes on different machines access shared files in same way

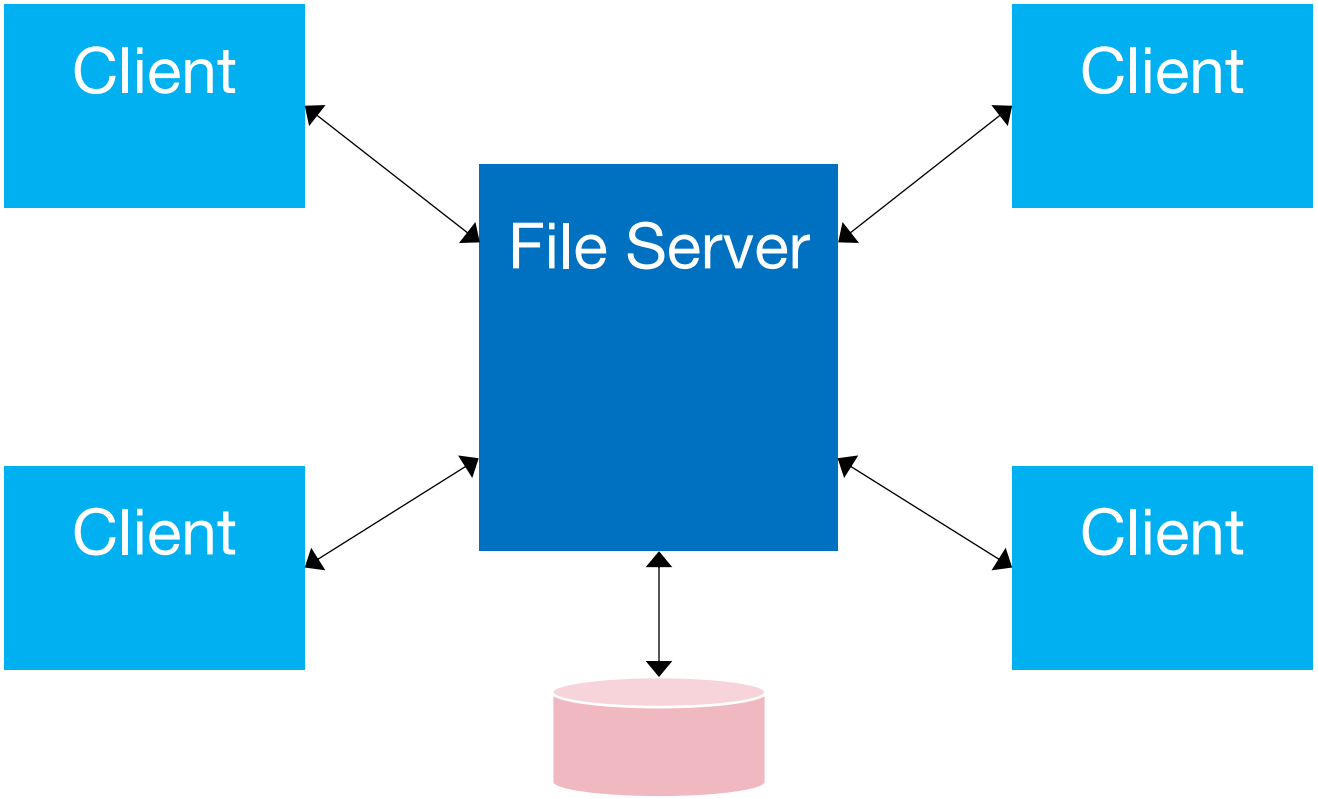
Sub-objectives

- Fast + simple crash recovery
  - Both clients and file server may crash
- Transparent access
  - Can't tell it's over the network
  - Normal UNIX semantics
- Reasonable performance

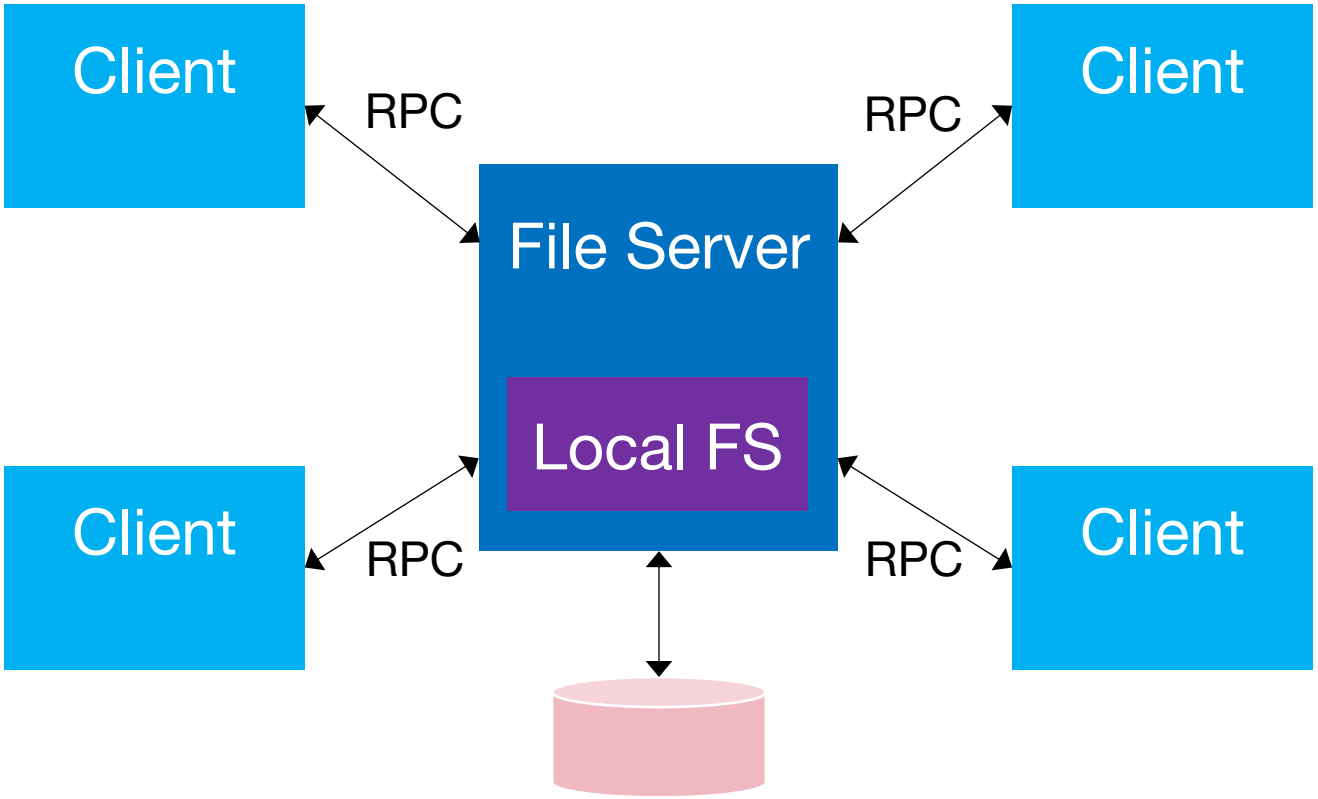
NFS agenda

- Architecture
- Network API
- Cache

NFS architecture



NFS architecture



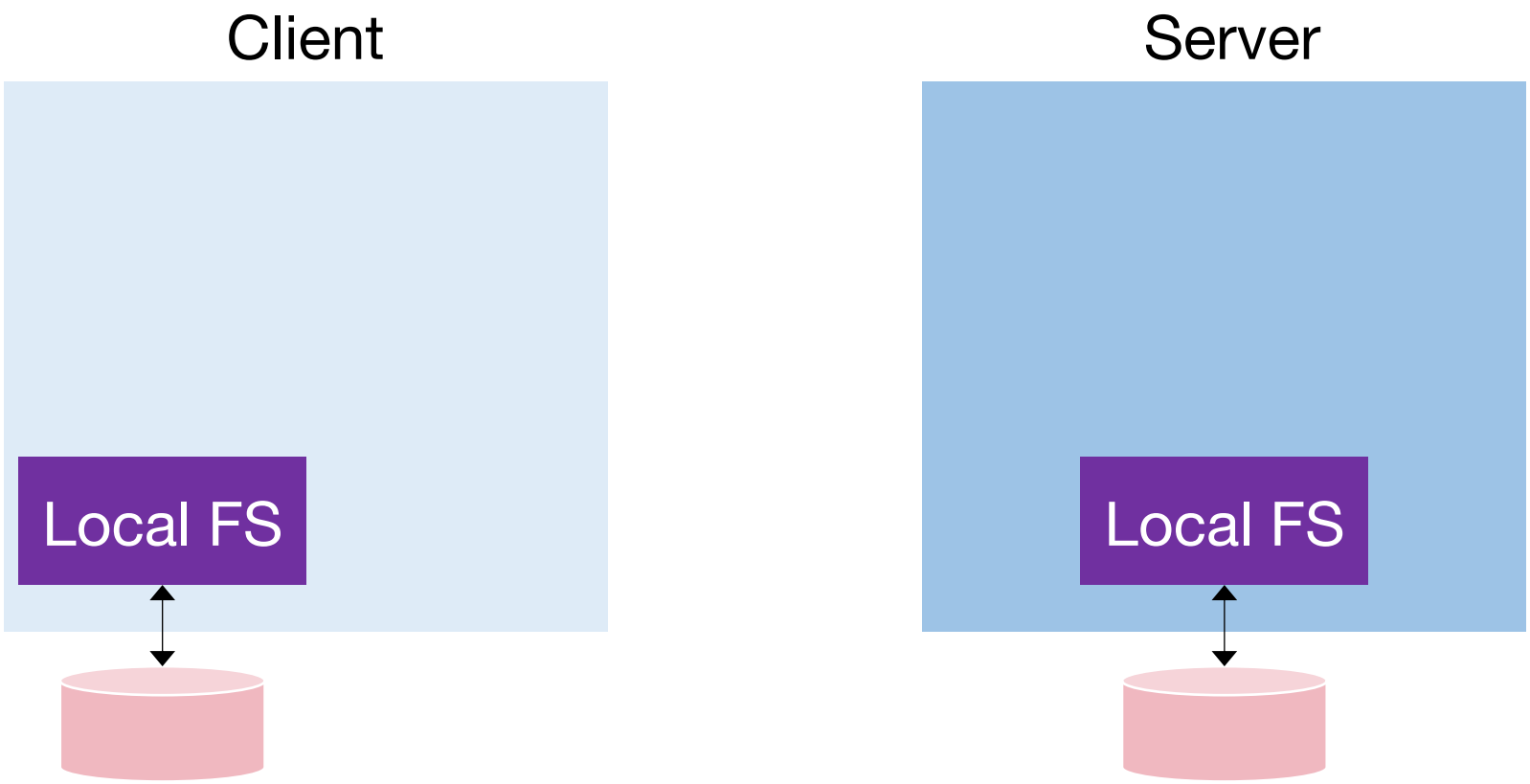
Main design decisions

- What functions to expose via RPC?
- Think of NFS as more of a protocol than a particular file system implementation
- Many companies have implemented NFS:
  - Oracle/Sun, NetApp, (Dell) EMC, IBM, etc.

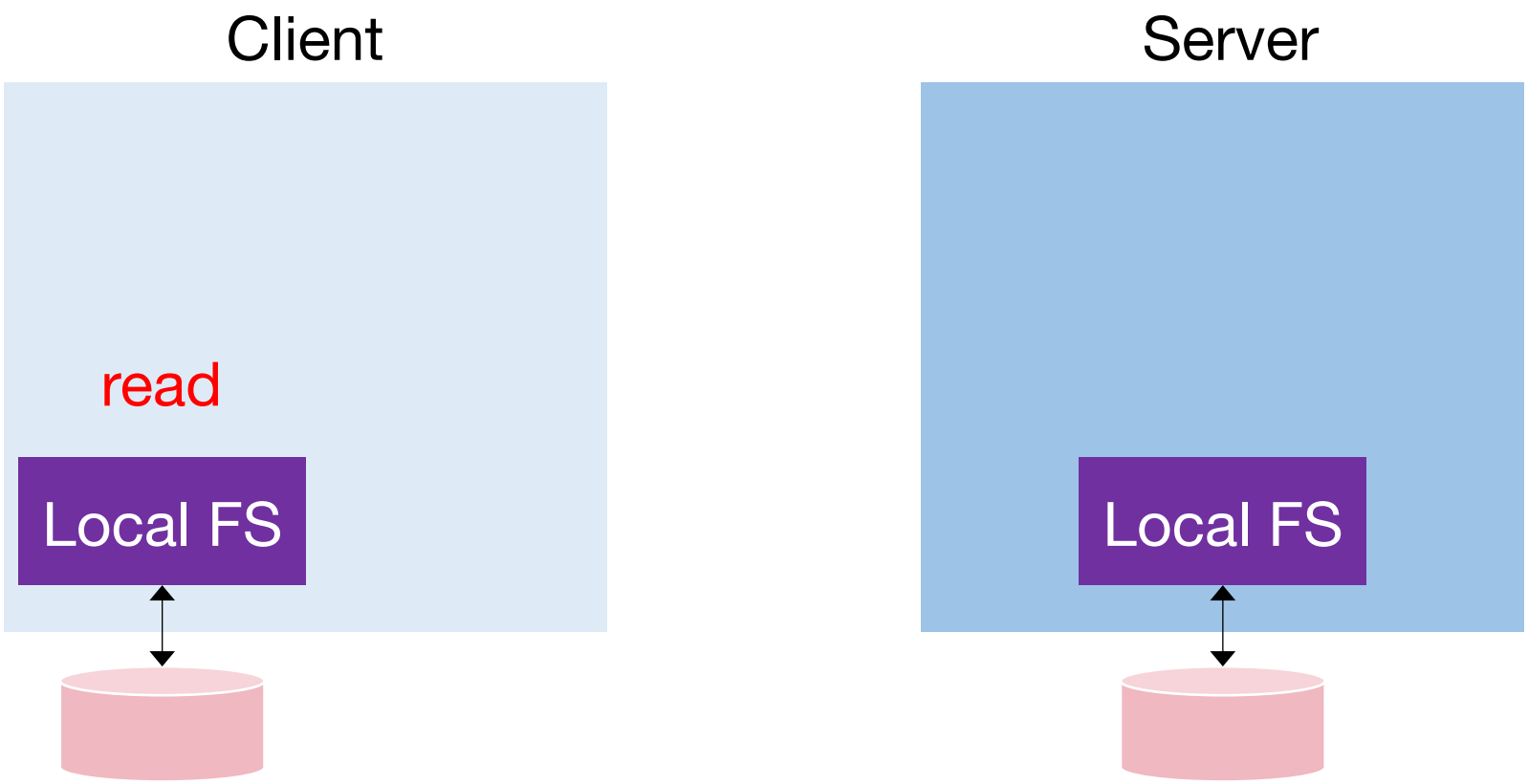
Today's lecture

- We're looking at NFSv2
- There is now an NFSv3 and NFSv4 with many changes

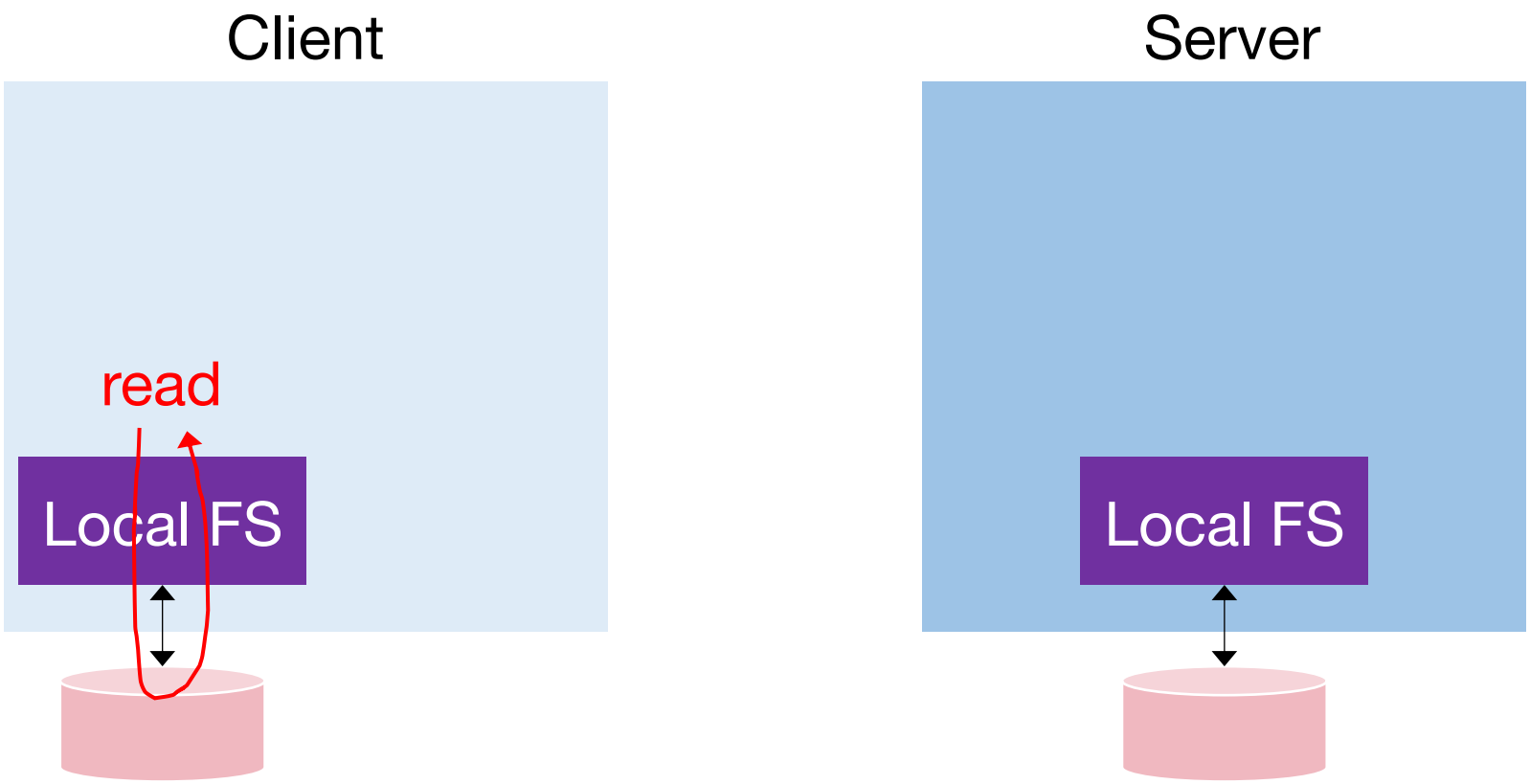
General strategy: Export FS



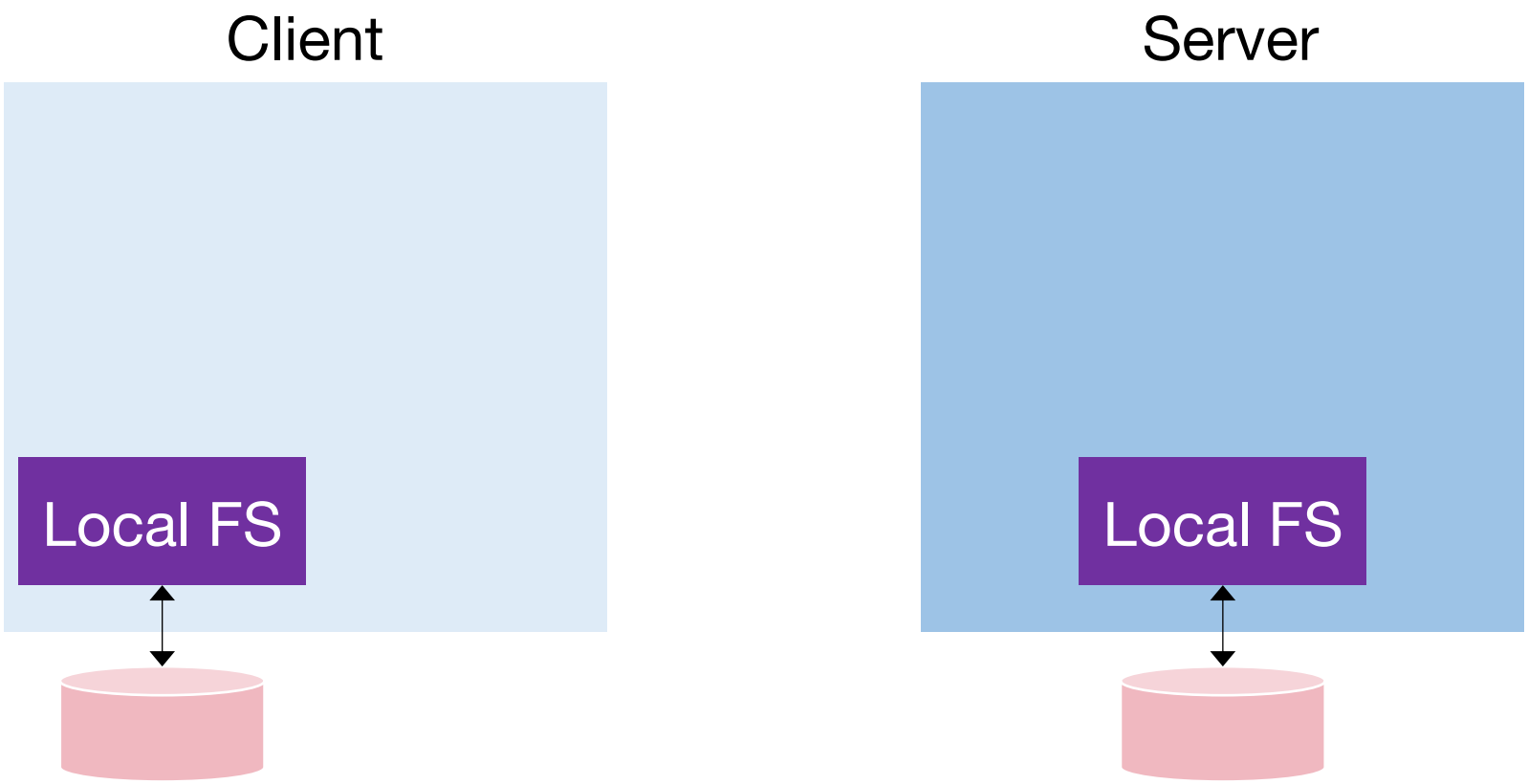
General strategy: Export FS



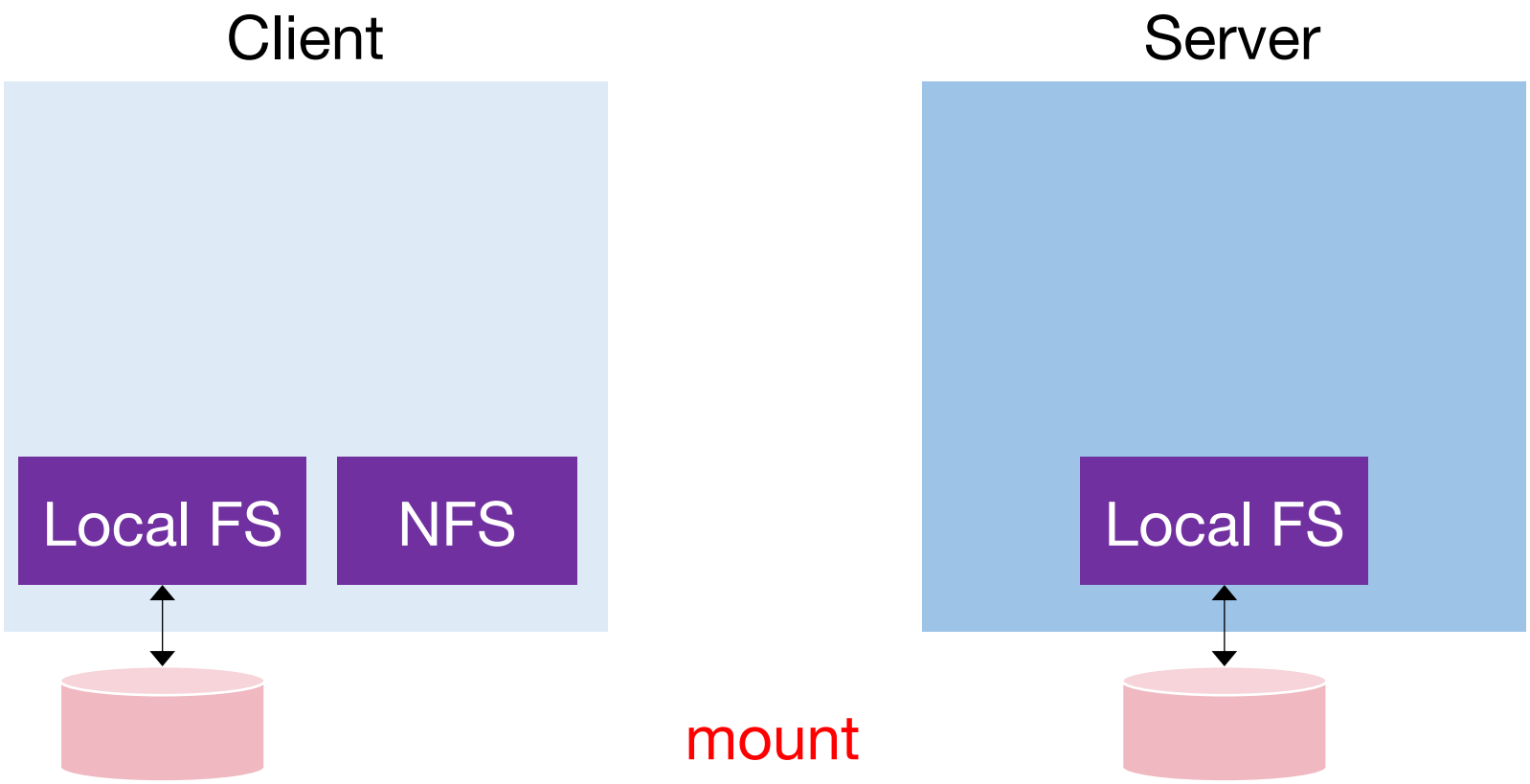
General strategy: Export FS

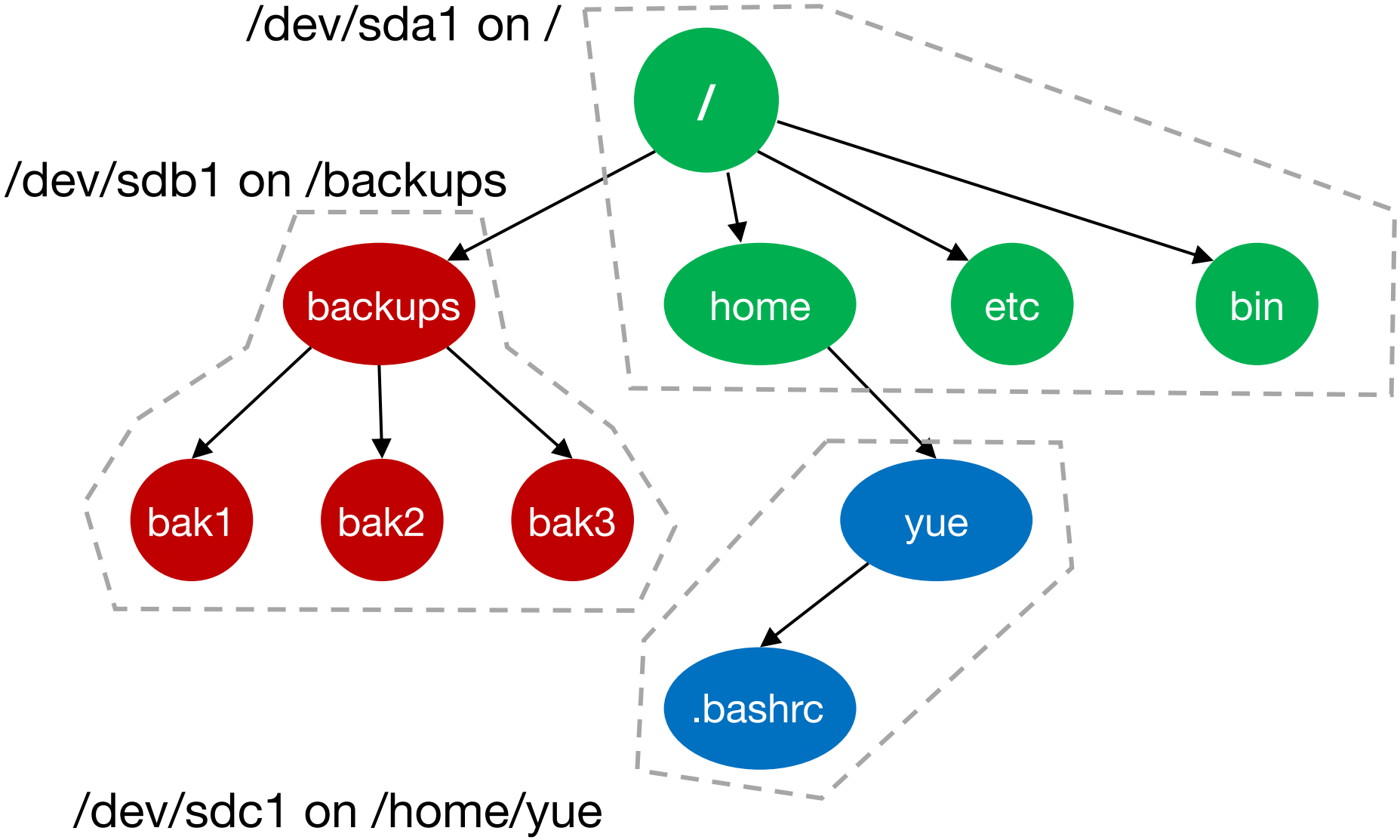


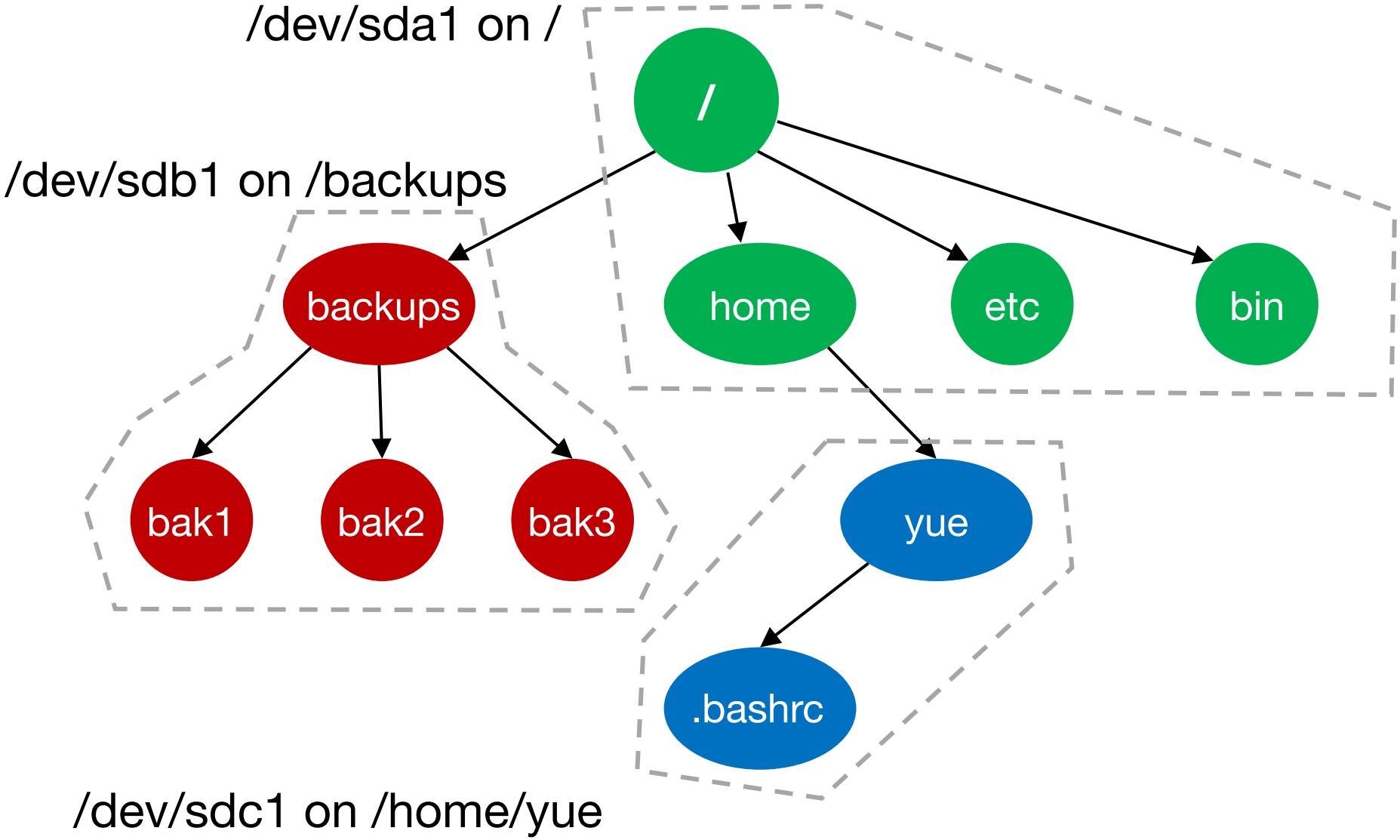
General strategy: Export FS



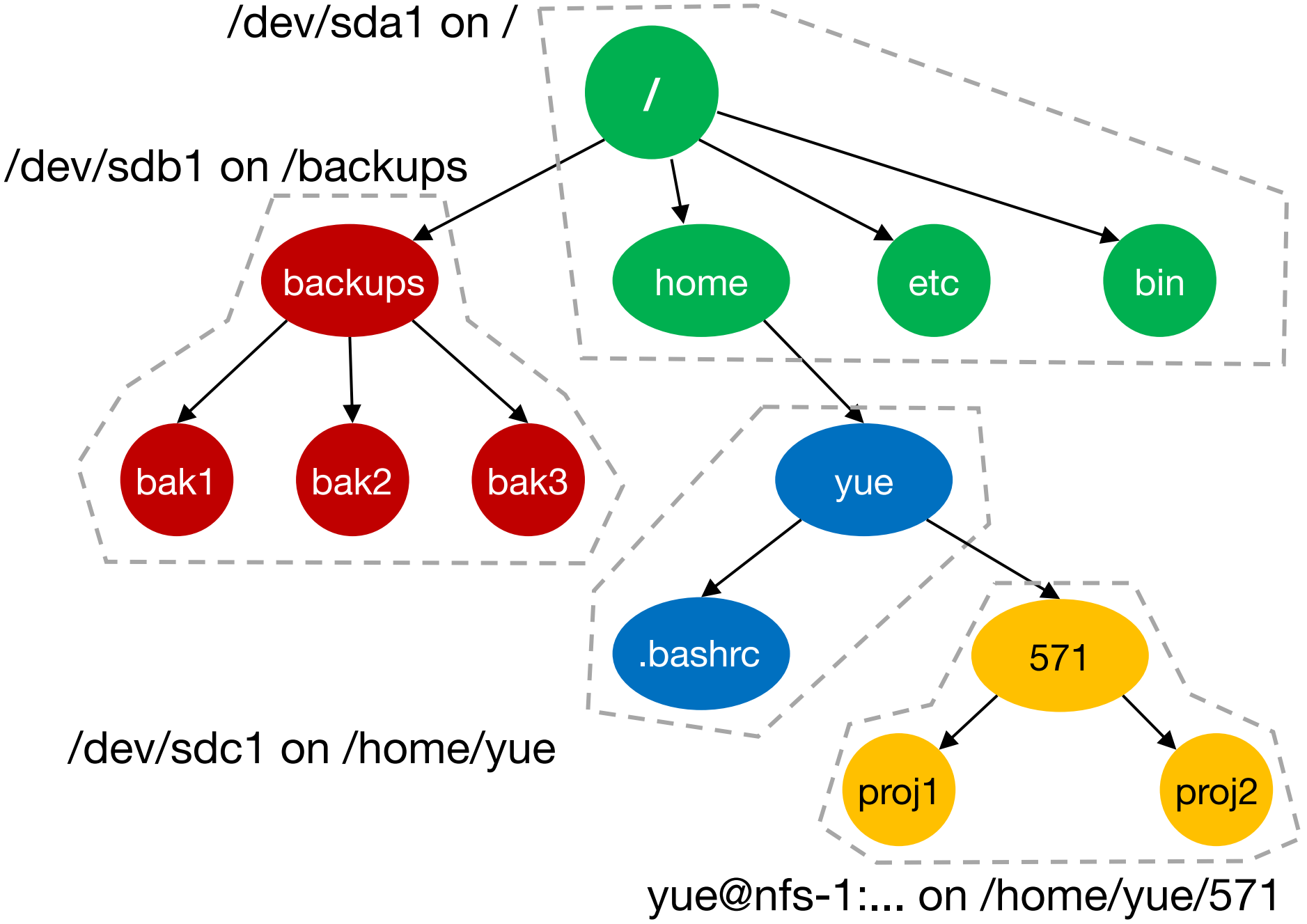
General strategy: Export FS



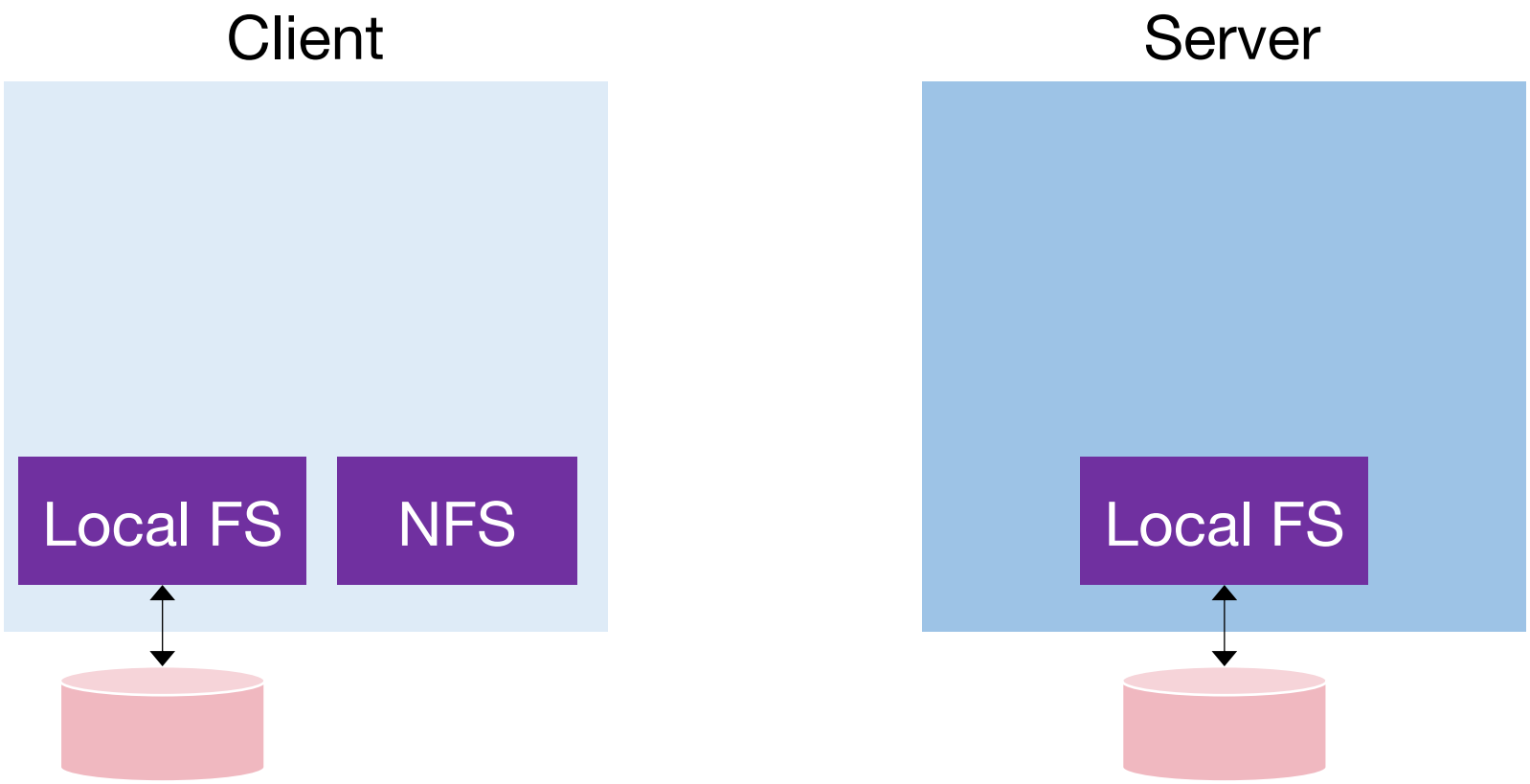

/dev/sda1 on /
/dev/sdb1 on /backups
/dev/sdc1 on /home/yue
/
backups
home
etc
bin
bak1
bak2
bak3
yue
.bashrc



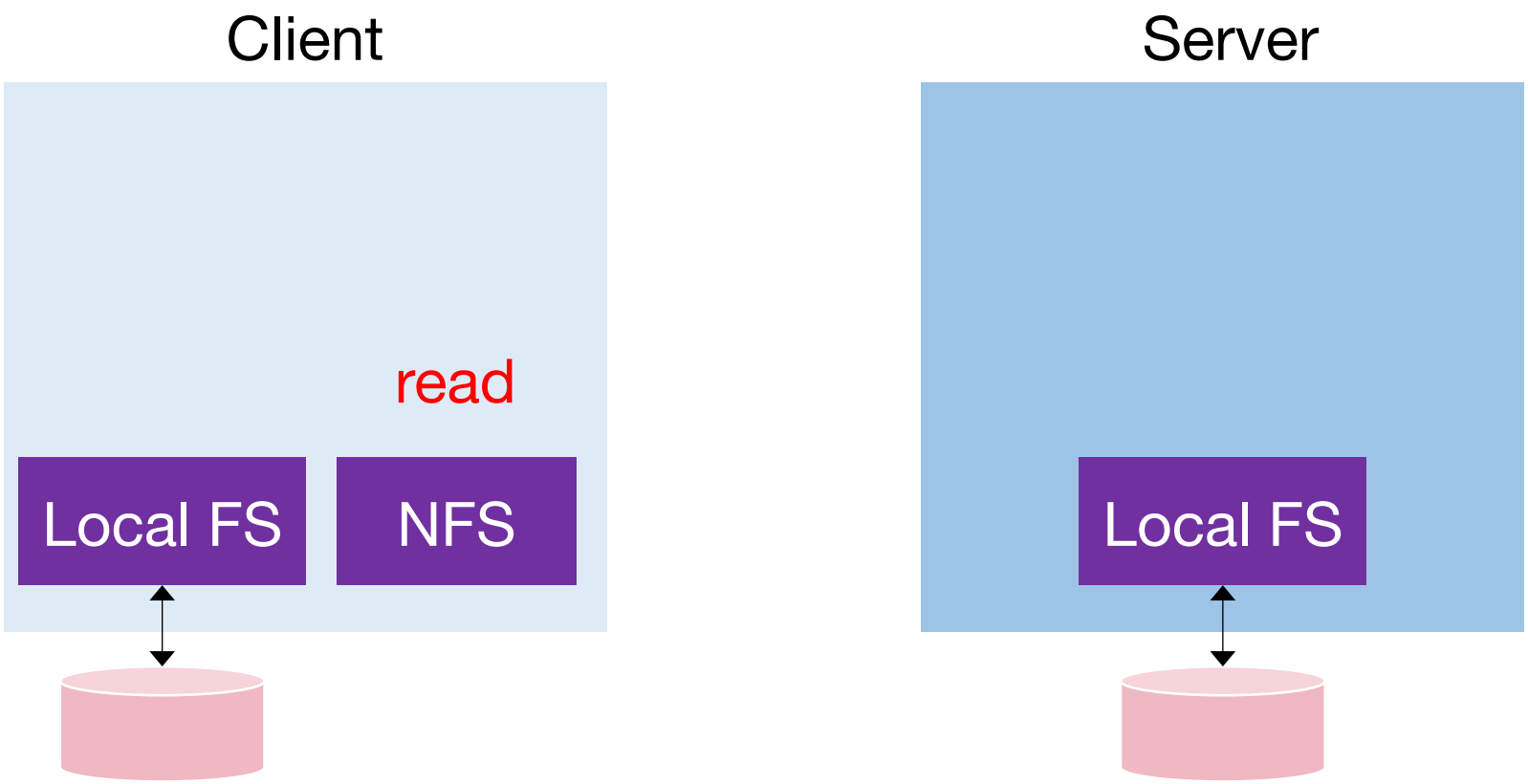

```
mount yue@nfs-1:... /home/yue/571
```


/dev/sda1 on /
/
/dev/sdb1 on /backups
backups
home
etc
bin
bak1
bak2
bak3
yue
.bashrc
571
/dev/sdc1 on /home/yue
proj1
proj2
yue@nfs-1:... on /home/yue/571

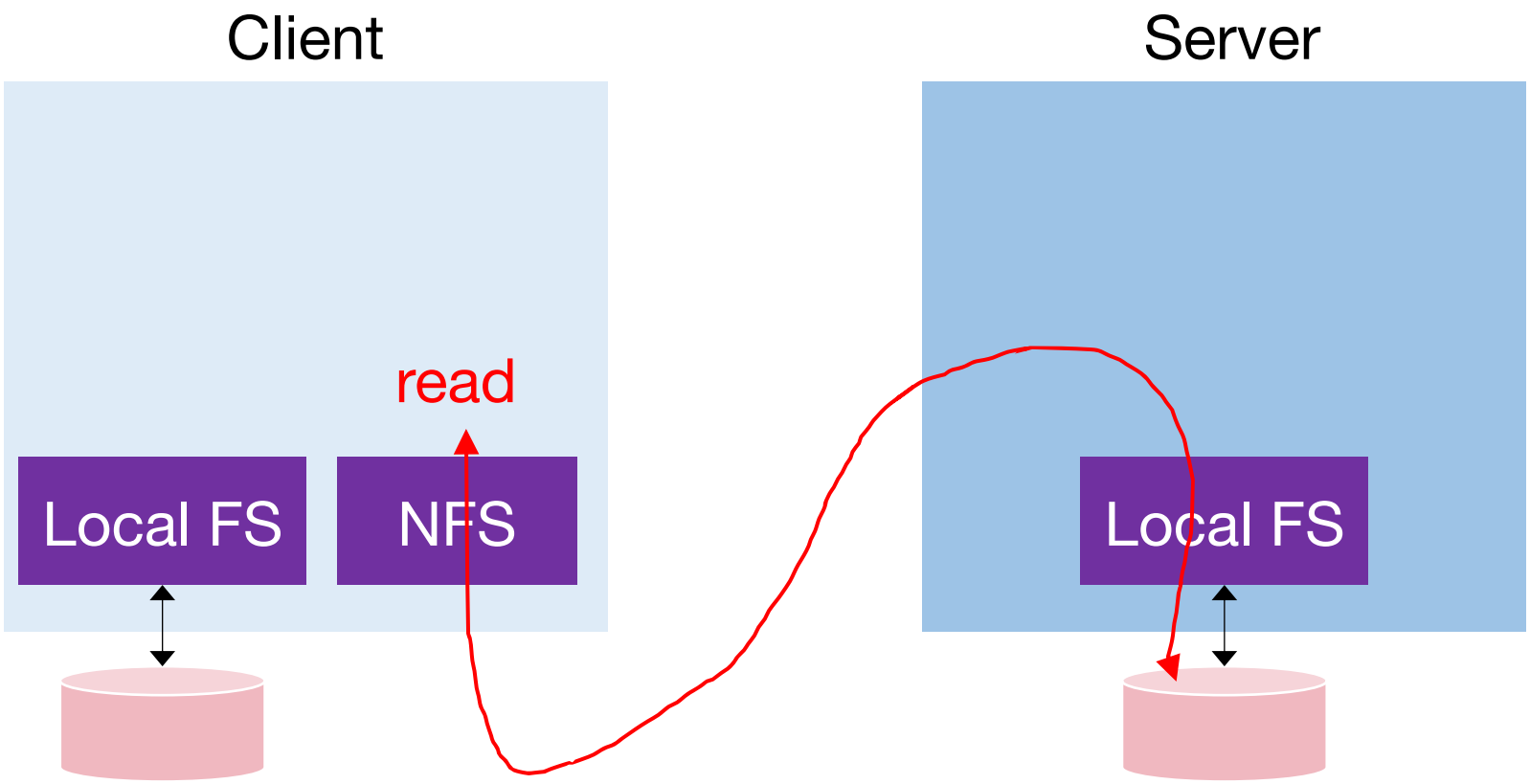
General strategy: Export FS



General strategy: Export FS



General strategy: Export FS



NFS agenda

- Architecture
- Network API
- Cache

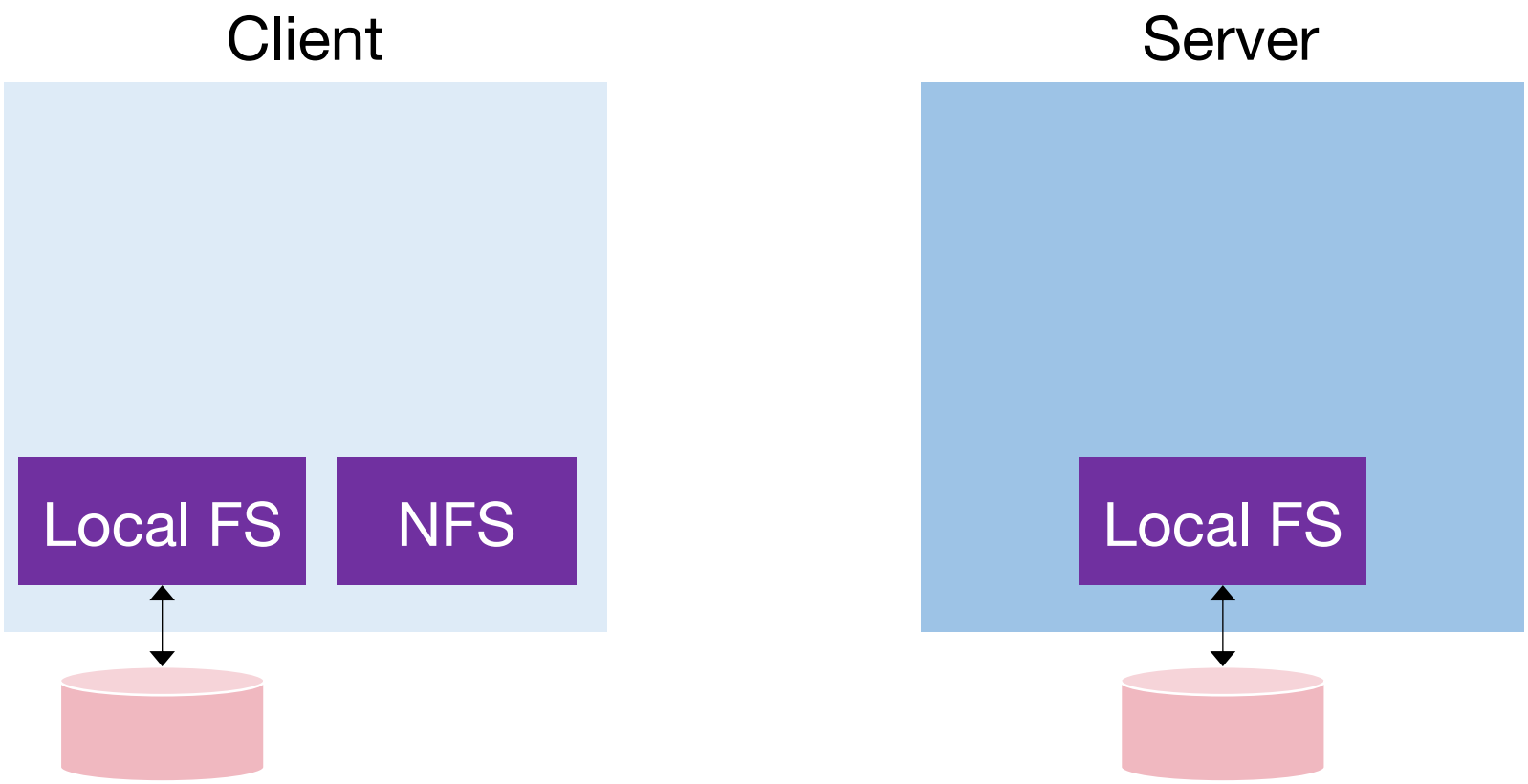
Strategy 1

- Wrap regular UNIX system calls using RPC

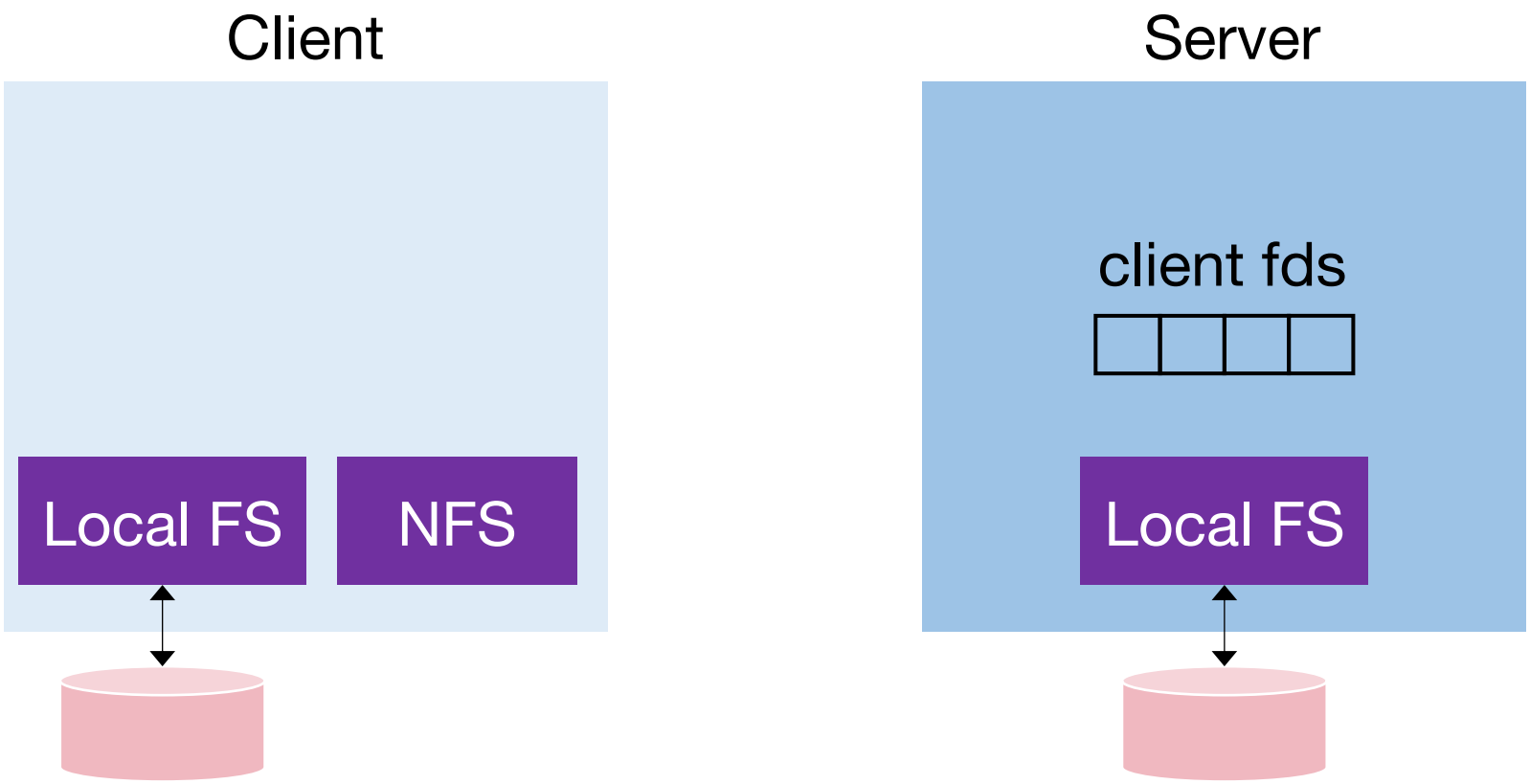
open() on client calls open() on server
open() on server returns fd back to client

read(fd) on client calls read(fd) on server
read(fd) on server returns data back to client

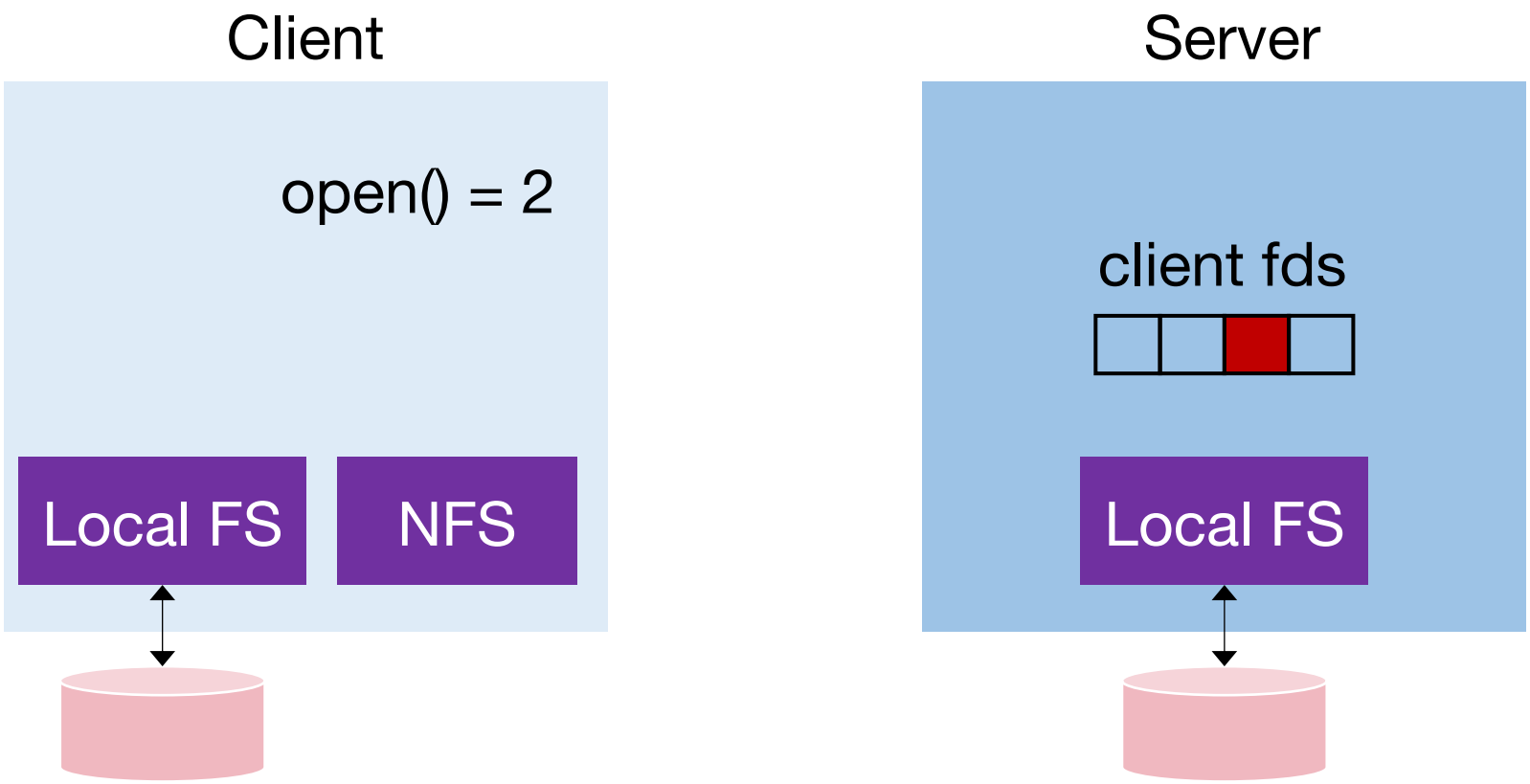
File descriptors



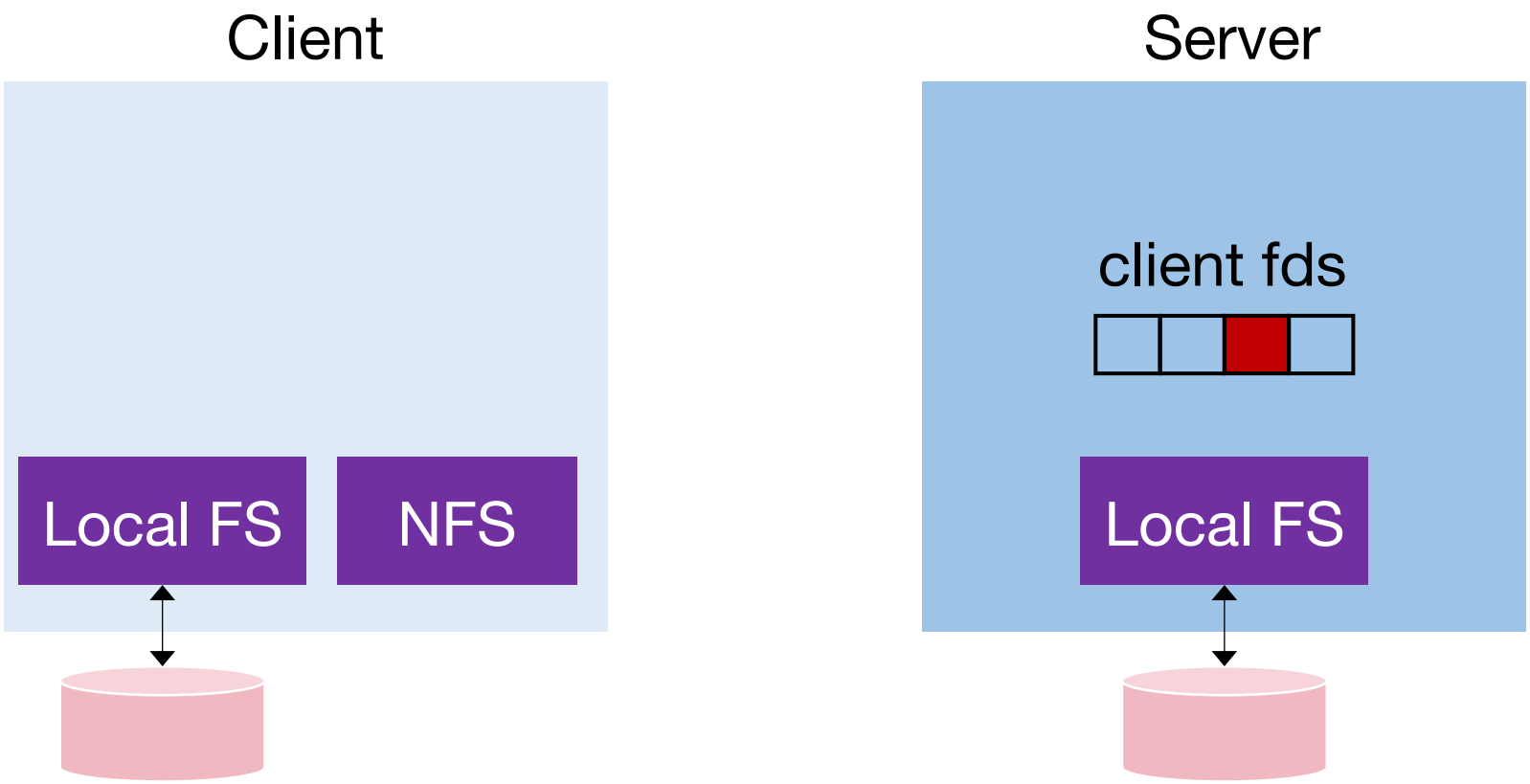
File descriptors



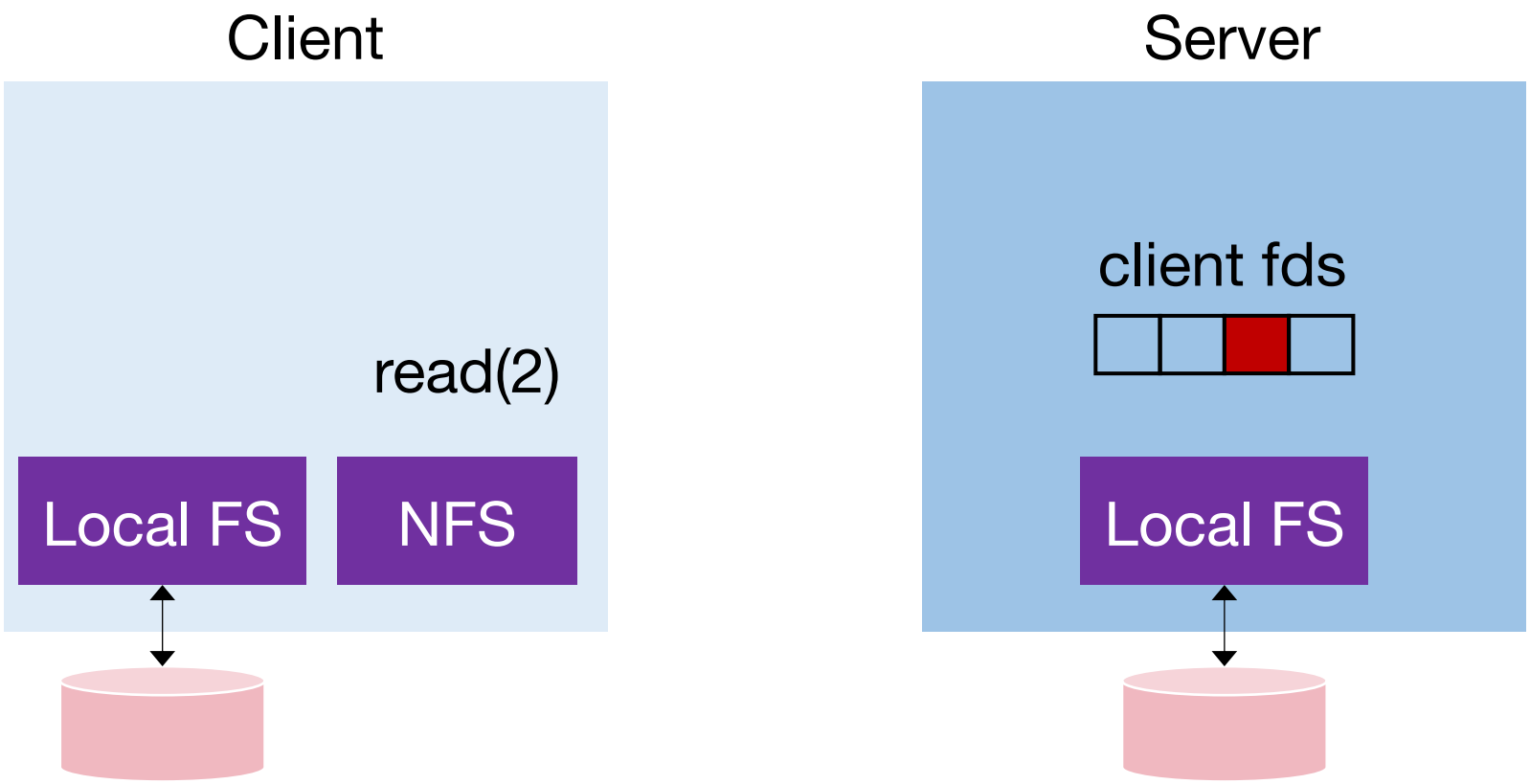
File descriptors



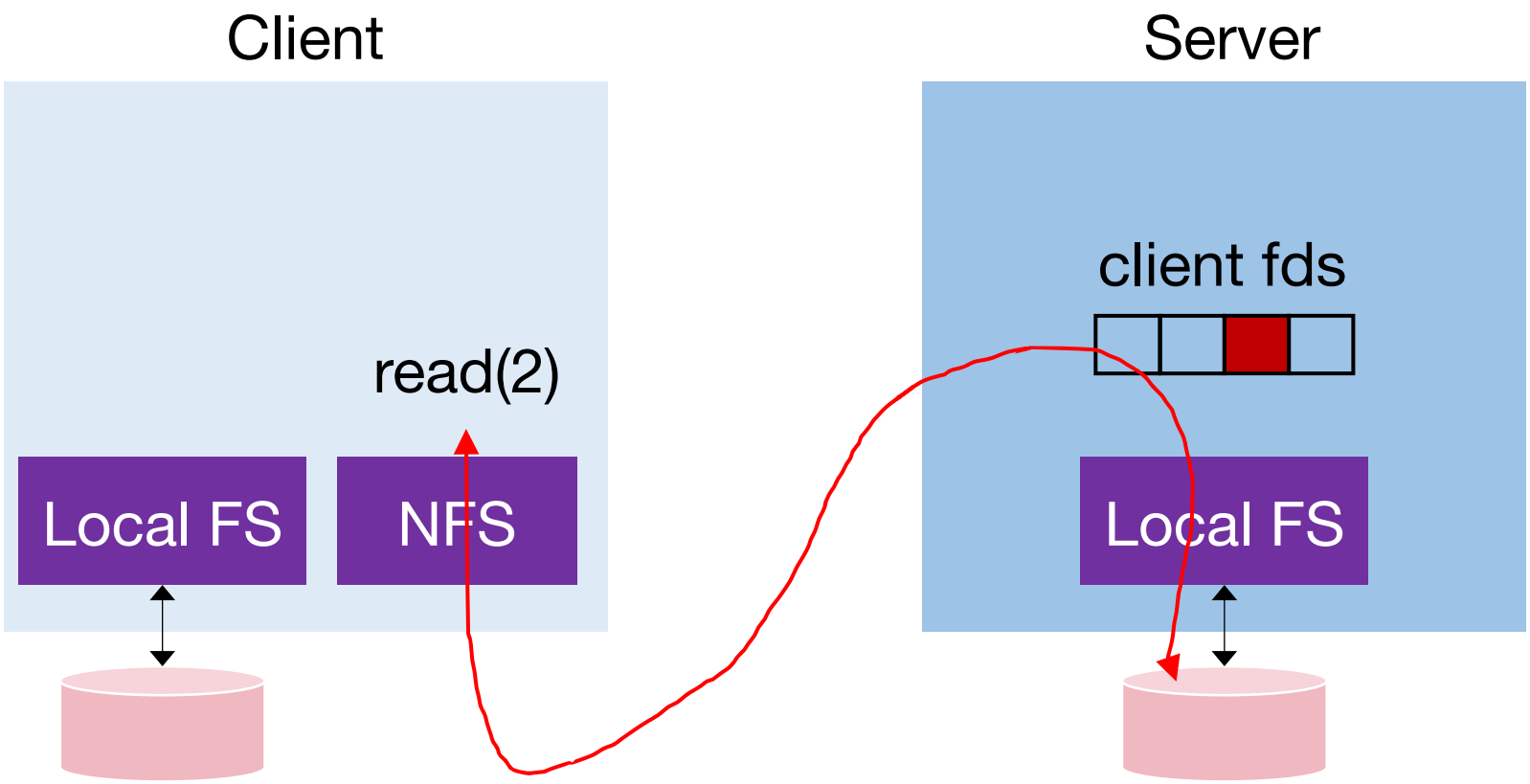
File descriptors



File descriptors



File descriptors



Strategy 1's problems

- What about crashes?

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
…
read(fd, buf, MAX);
```

crash!

Imagine server crashes and reboots during reads…

Strategy 1's problems

- What about crashes?

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
…
read(fd, buf, MAX);
```

crash!

Nice if this just looks like a slow read

Imagine server crashes and reboots during reads…

Potential solutions

1. Run some crash recovery protocol upon reboot
   - Complex

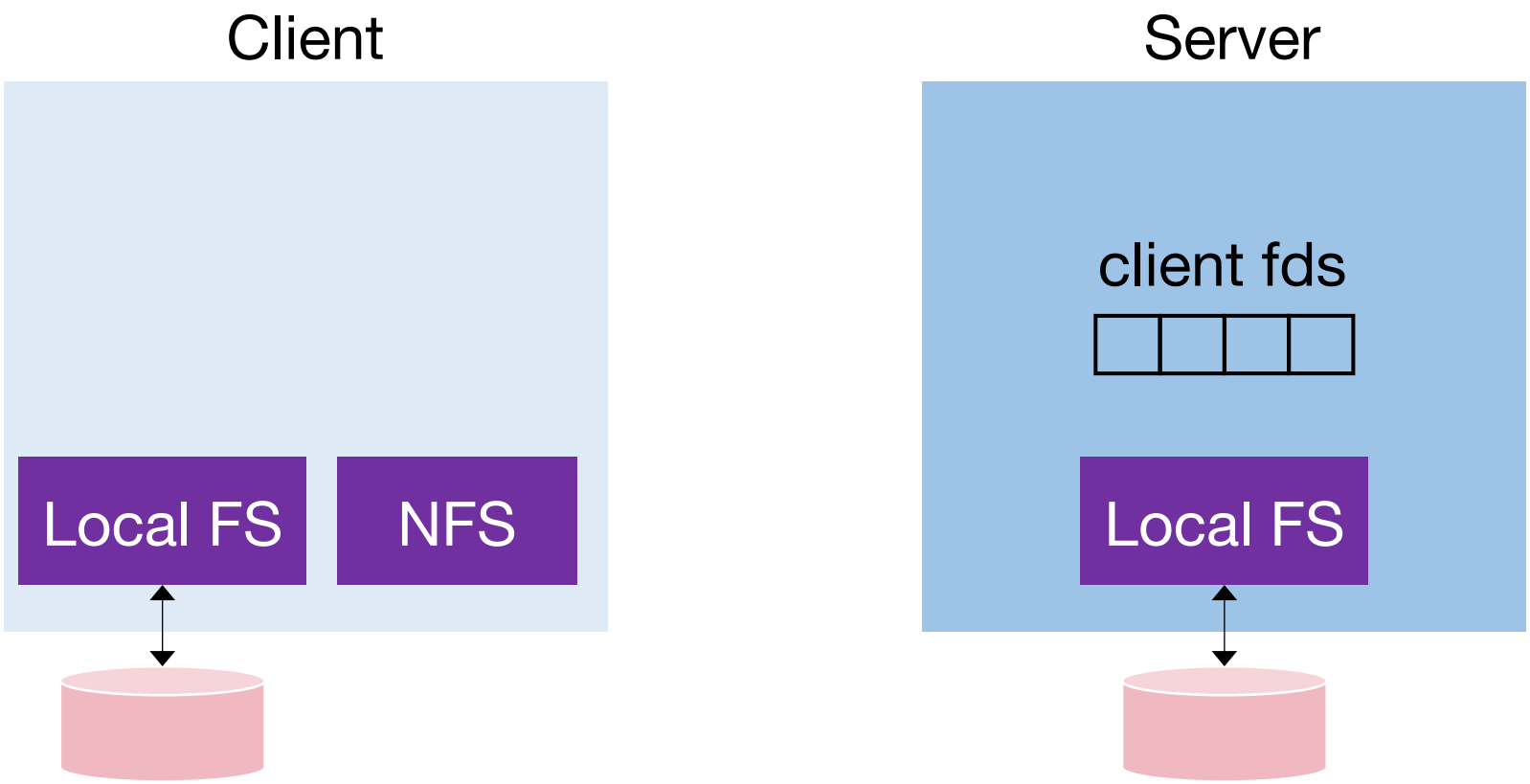
2. Persist fds on server disk
   - Slow
   - What if client crashes instead?

Strategy 2: Put all info in requests

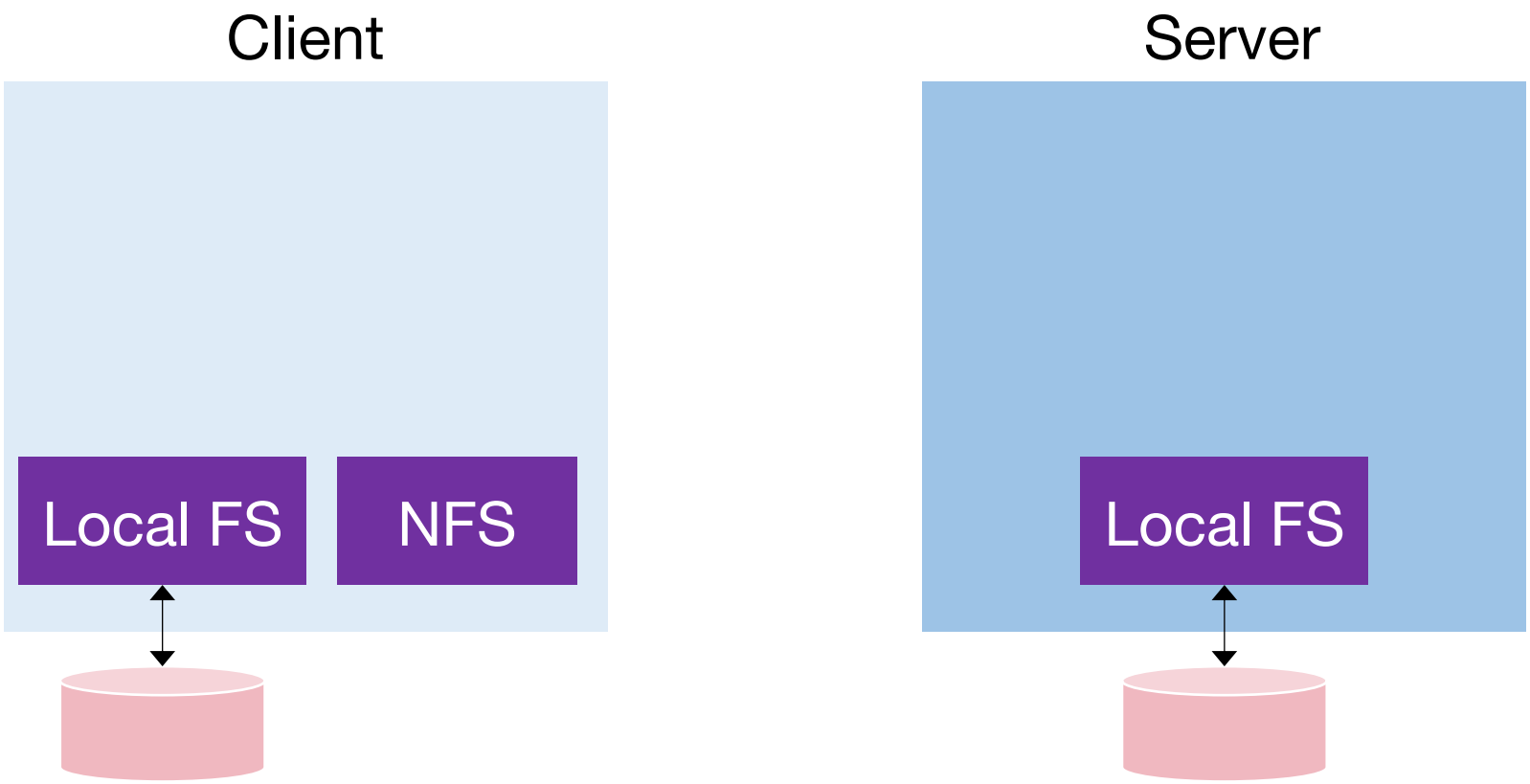
- Use "stateless" protocol!
  - Server maintains no state about clients
  - Server still keeps other state, of course

Eliminate file descriptors

Eliminate file descriptors



Eliminate file descriptors



Strategy 2: Put all info in requests

- Use "stateless" protocol!
  - Server maintains no state about clients
  - Server still keeps other state, of course

- Need API change. One possibility:

```
pread(char *path, buf, size, offset);
pwrite(char *path, buf, size, offset);
```

- Specify path and offset each time. Server needs not remember. Pros/cons?

Strategy 2: Put all info in requests

- Use "stateless" protocol!
  - Server maintains no state about clients
  - Server still keeps other state, of course

- Need API change. One possibility:

```
pread(char *path, buf, size, offset);
pwrite(char *path, buf, size, offset);
```

- Specify path and offset each time. Server needs not remember. Pros/cons? Too many path lookups

Strategy 3: inode requests

```
pread(char *path, buf, size, offset);
pwrite(char *path, buf, size, offset);
```

Strategy 3: inode requests

```
inode = open(char *path);
pread(inode, buf, size, offset);
pwrite(inode, buf, size, offset);
```

Strategy 3: inode requests

```
inode = open(char *path);
pread(inode, buf, size, offset);
pwrite(inode, buf, size, offset);
```

This is pretty good! Any correctness problems?

Strategy 3: inode requests

```
inode = open(char *path);
pread(inode, buf, size, offset);
pwrite(inode, buf, size, offset);
```

This is pretty good! Any correctness problems?

- What if file is deleted, and inode is reused?

Strategy 4: File handles

```
fh = open(char *path);
pread(fh, buf, size, offset);
pwrite(fh, buf, size, offset);
```

File handle = <volume ID, inode #, generation #>

Aside: Append?

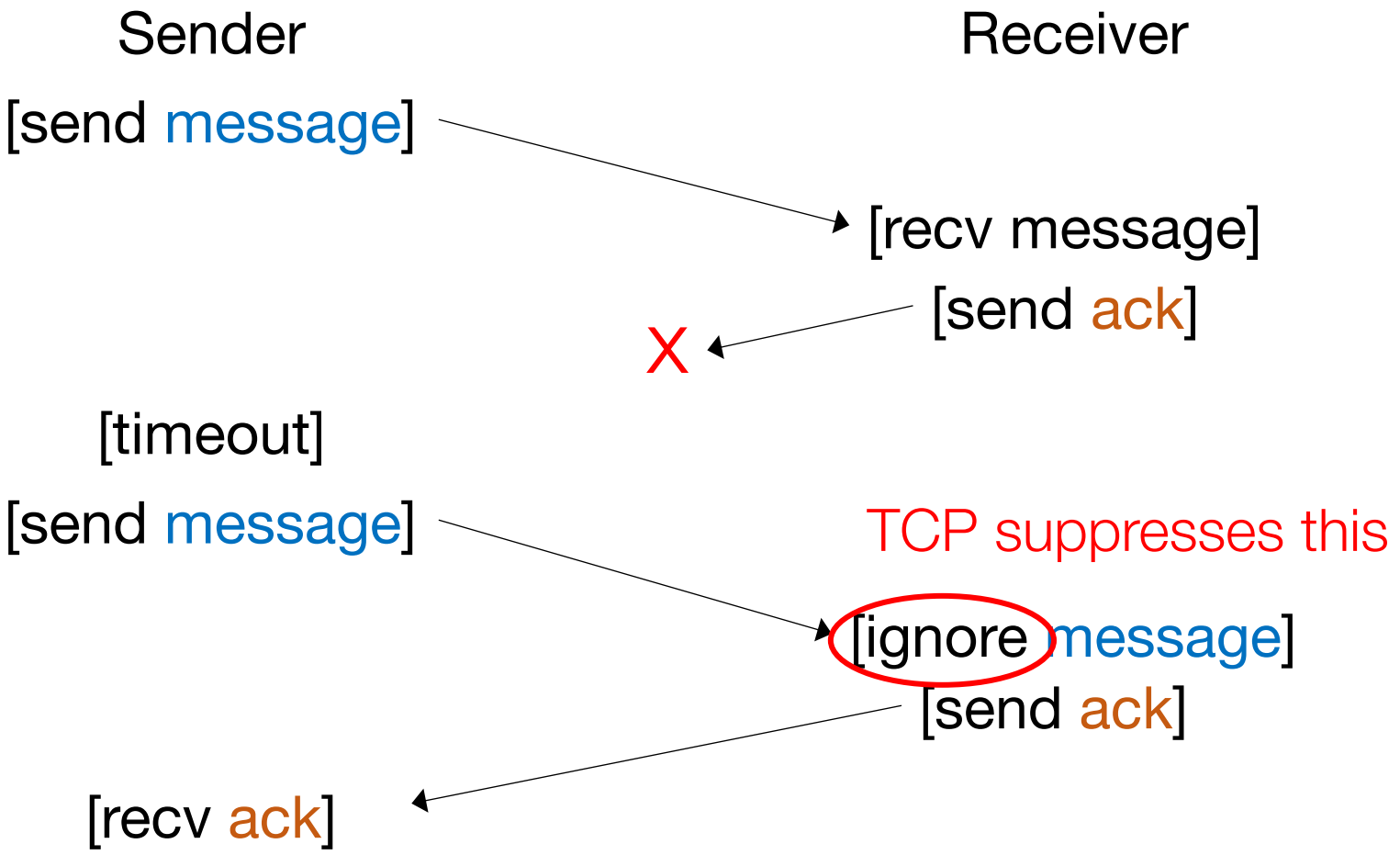
```
fh = open(char *path);
pread(fh, buf, size, offset);
pwrite(fh, buf, size, offset);
append(fh, buf, size);
```

Would append() be a good idea?

Problem: if our RPC library retries if no ACK or return, what happens when append is retried?

Solutions??

TCP remembers messages



Replica suppression is stateful

- TCP is stateful
  - If server crashes, it forgets what RPC's have been executed!

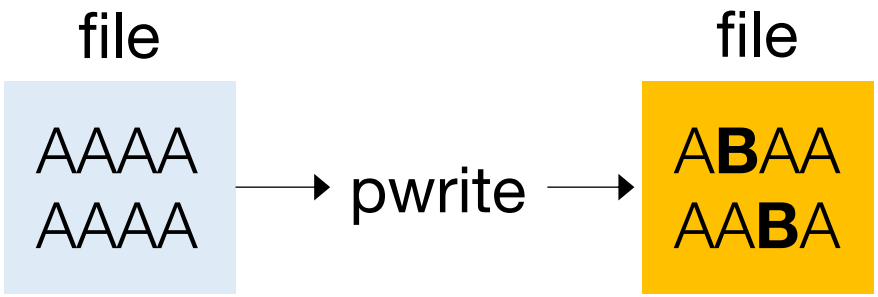
Replica suppression is stateful

- TCP is stateful
  - If server crashes, it forgets what RPC's have been executed!

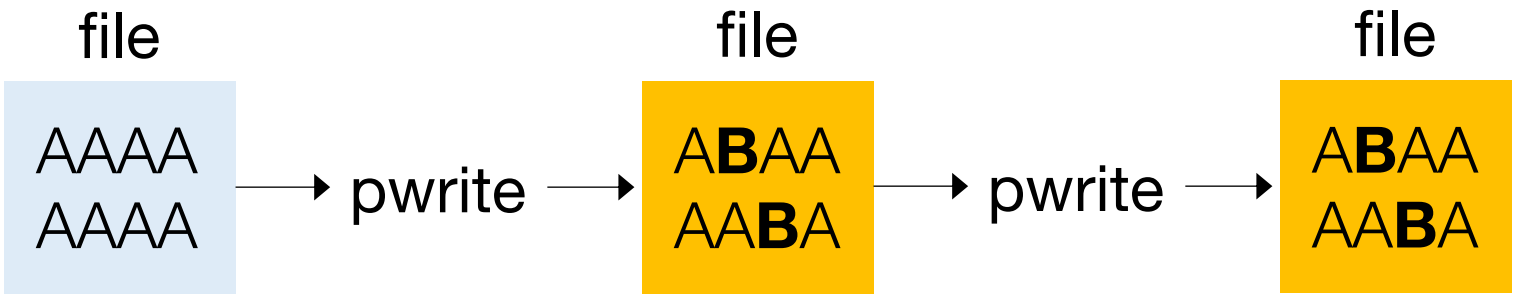
- Solution: design API so that there is no harm if executing a call more than once

- An API call that has this property is "idempotent": If f() is idempotent, then
  - f() has the same effect as f(); f(); … f(); f()

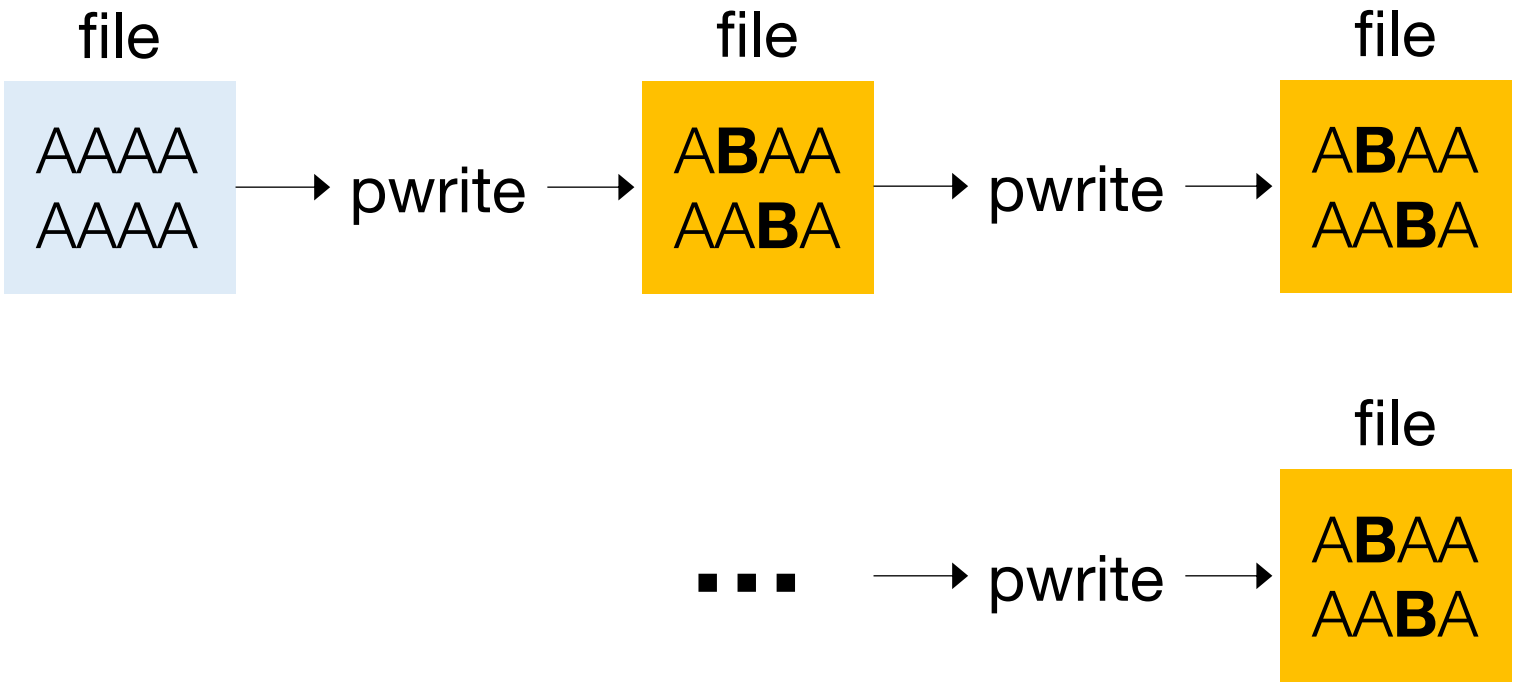
pwrite is idempotent



pwrite is idempotent

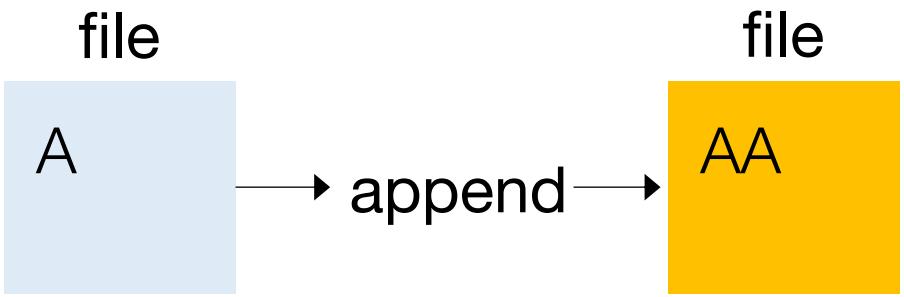


pwrite is idempotent

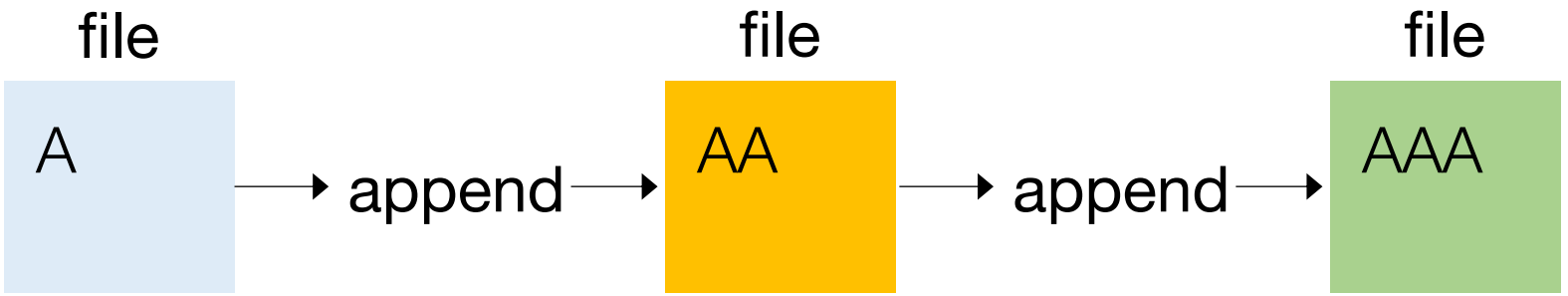


How about append?

Append is NOT idempotent



Append is NOT idempotent



Idempotence

- Idempotent
  - Any sort of read
  - pwrite
- Not idempotent
  - append
- What about these?
  - mkdir
  - creat

Strategy 4: File handles

```
fh = open(char *path);
pread(fh, buf, size, offset);
pwrite(fh, buf, size, offset);
append(fh, buf, size);
```

File handle = <volume ID, inode #, generation #>

NFS agenda

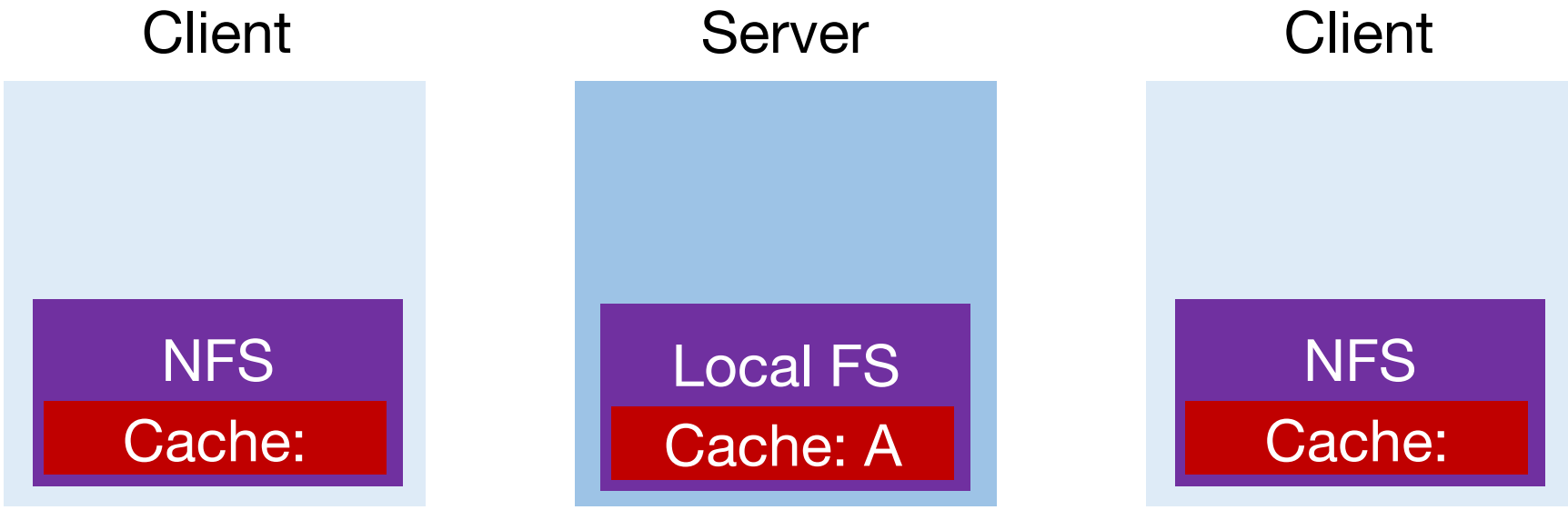
- Architecture
- Network API
- Cache

Cache

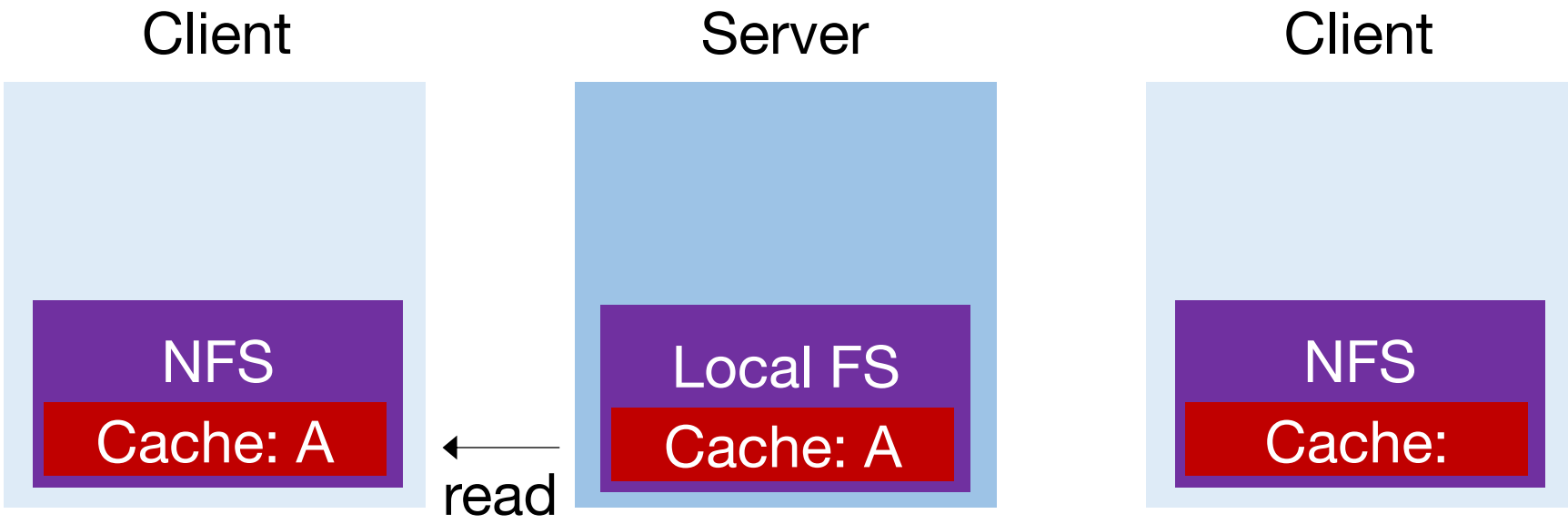
- We can cache data in three places
  - Server memory
  - Client memory
  - Client disk

- How to make sure all versions are in sync?

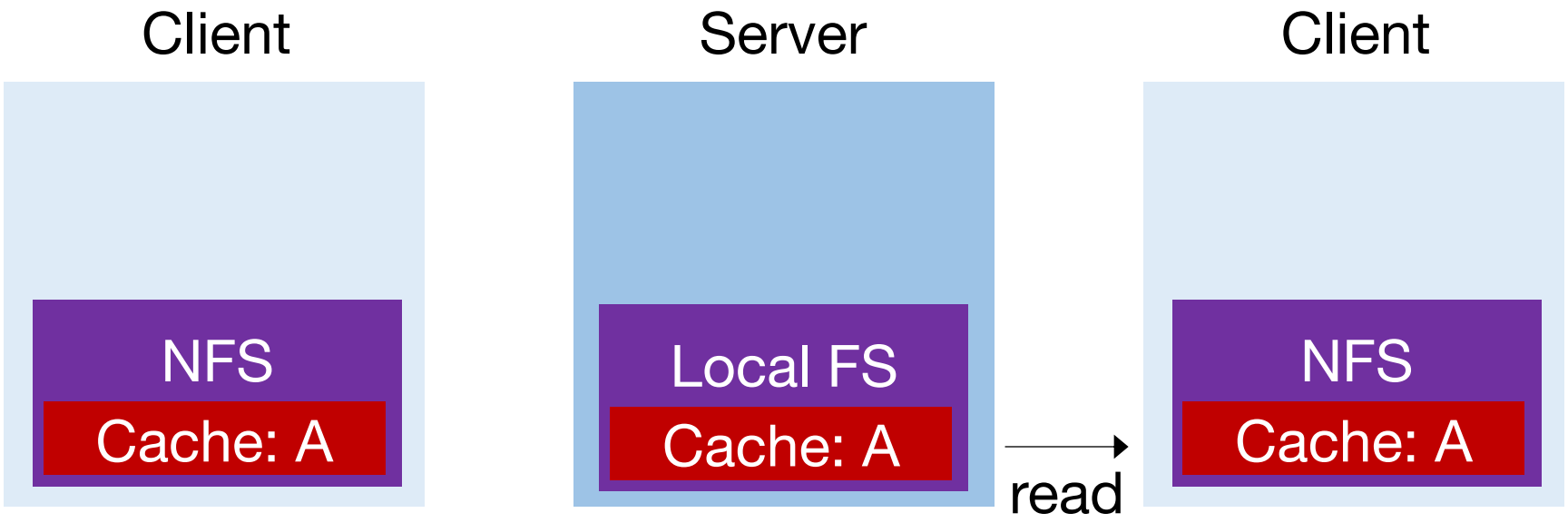
Cache



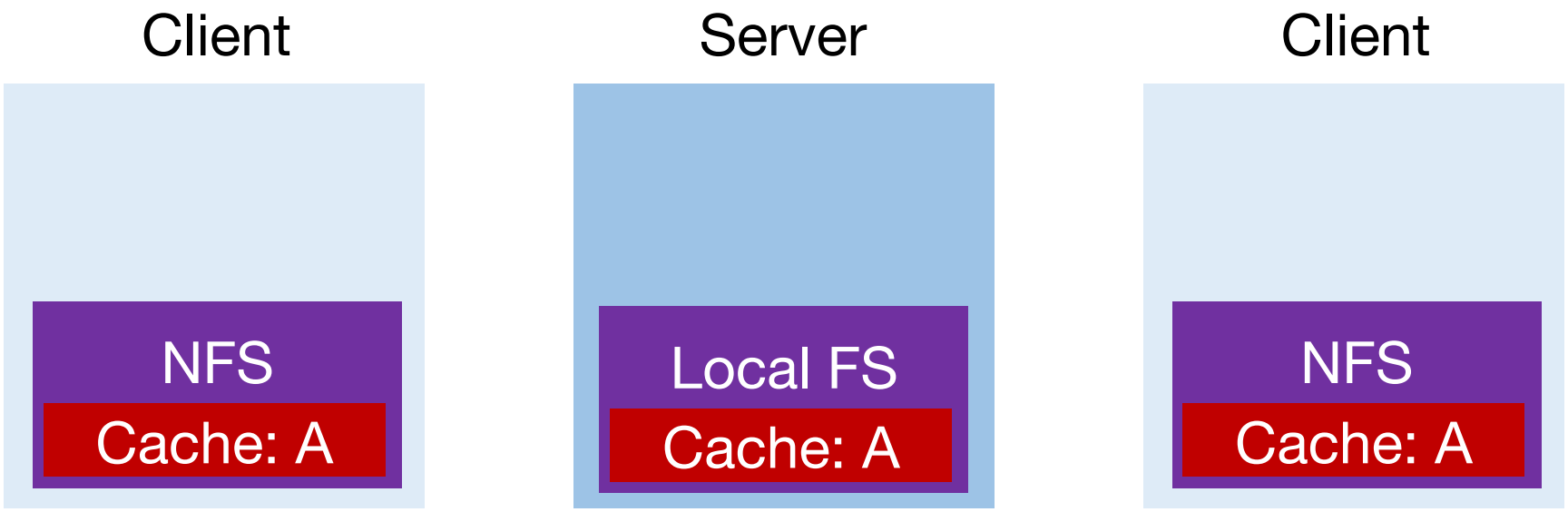
Cache


Client
Server
Client
NFS
Cache: A
Local FS
Cache: A
NFS
Cache:
read

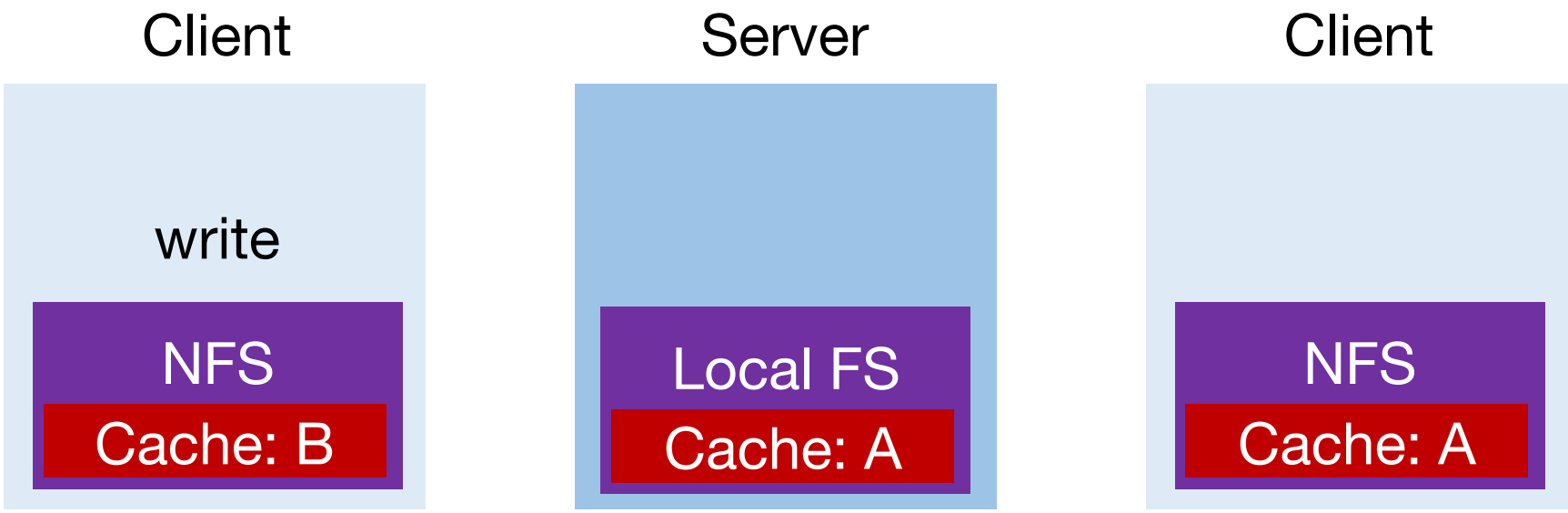
Cache


Client
Server
Client
NFS
Cache: A
Local FS
Cache: A
read
NFS
Cache: A

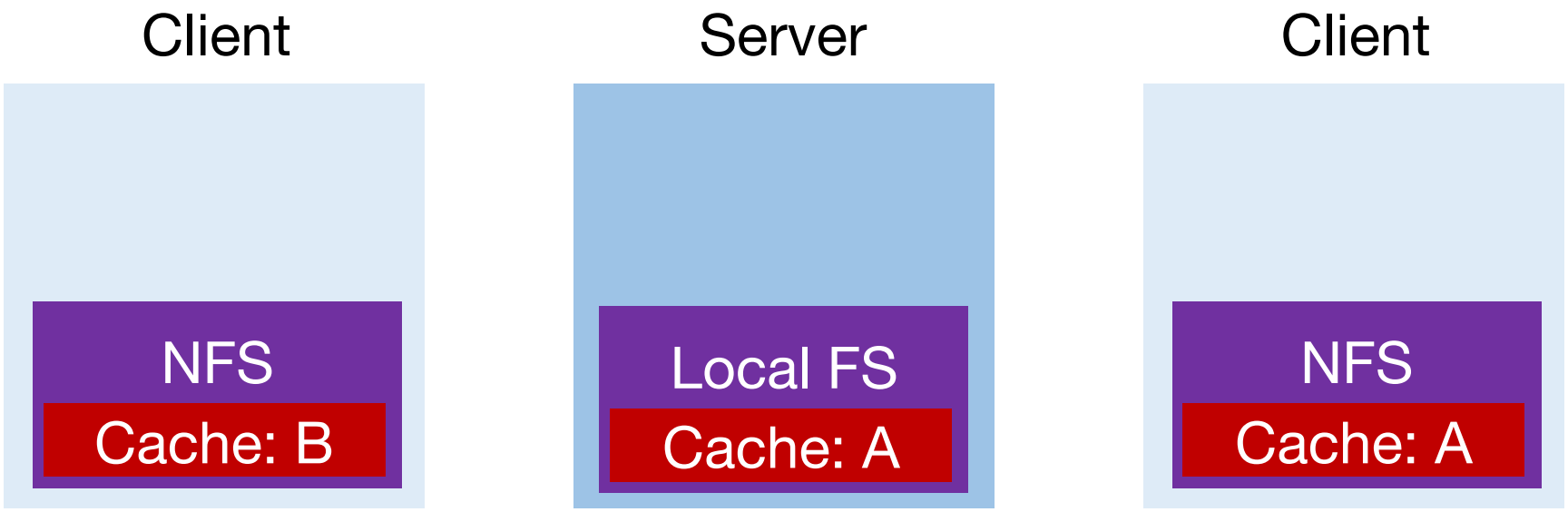
Cache



Cache

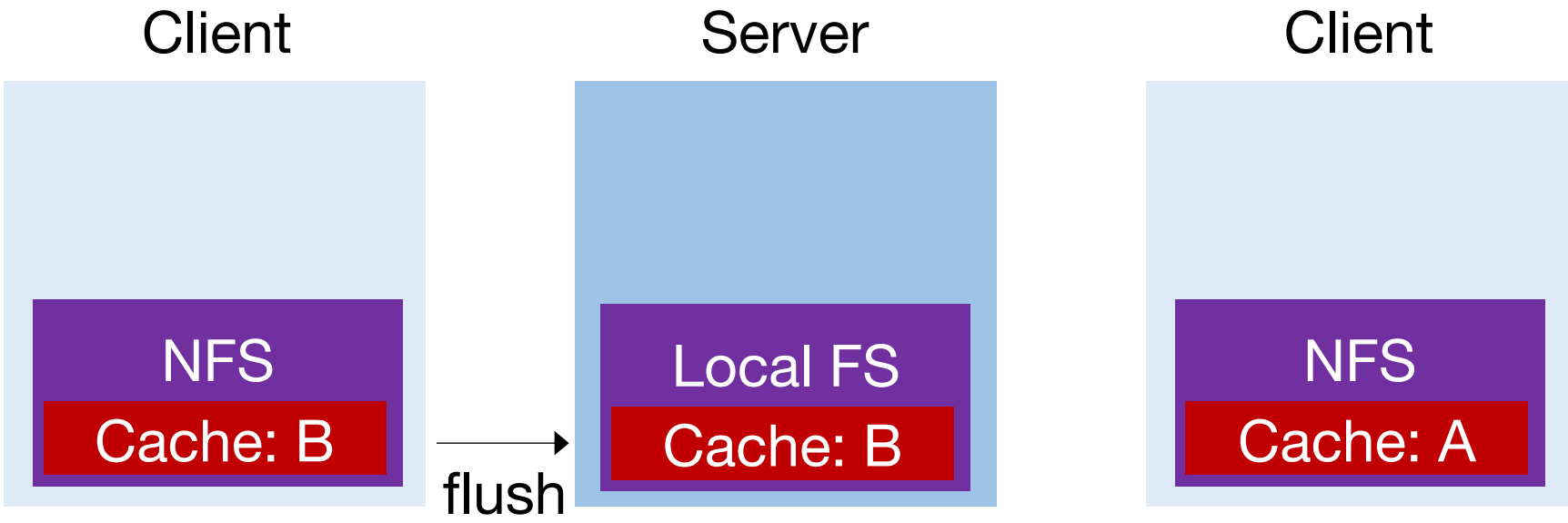

Client
Server
Client
write
NFS
Cache: B
Local FS
Cache: A
NFS
Cache: A

Cache

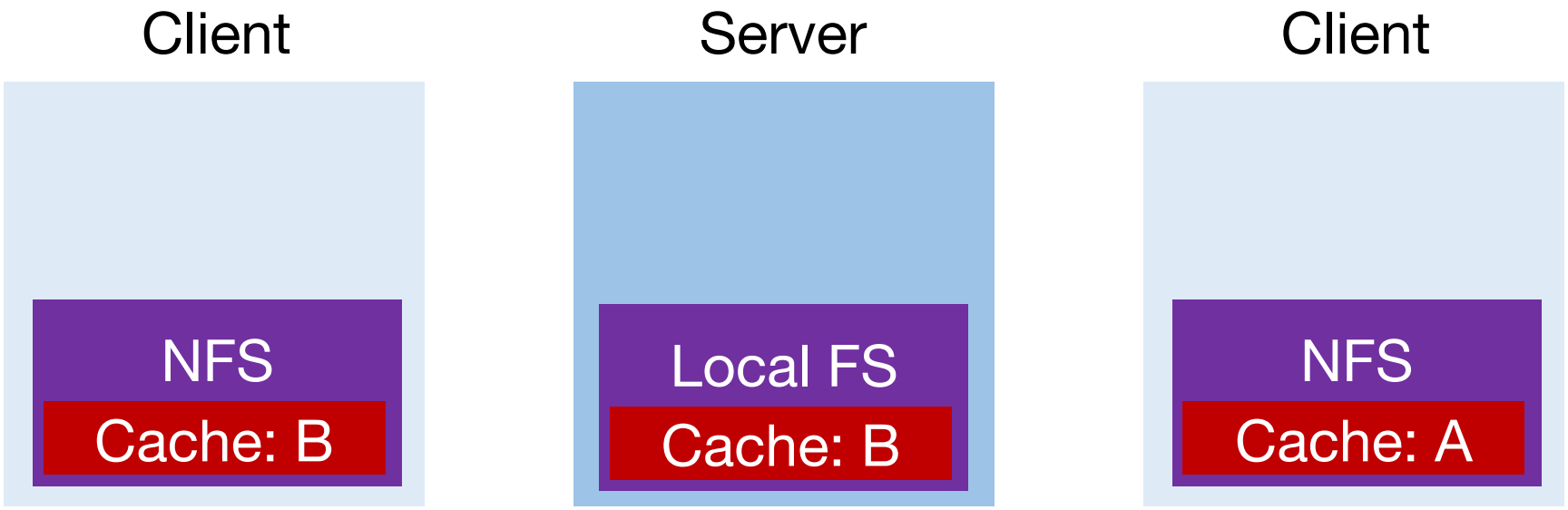


"Update visibility" problem: server doesn't have latest

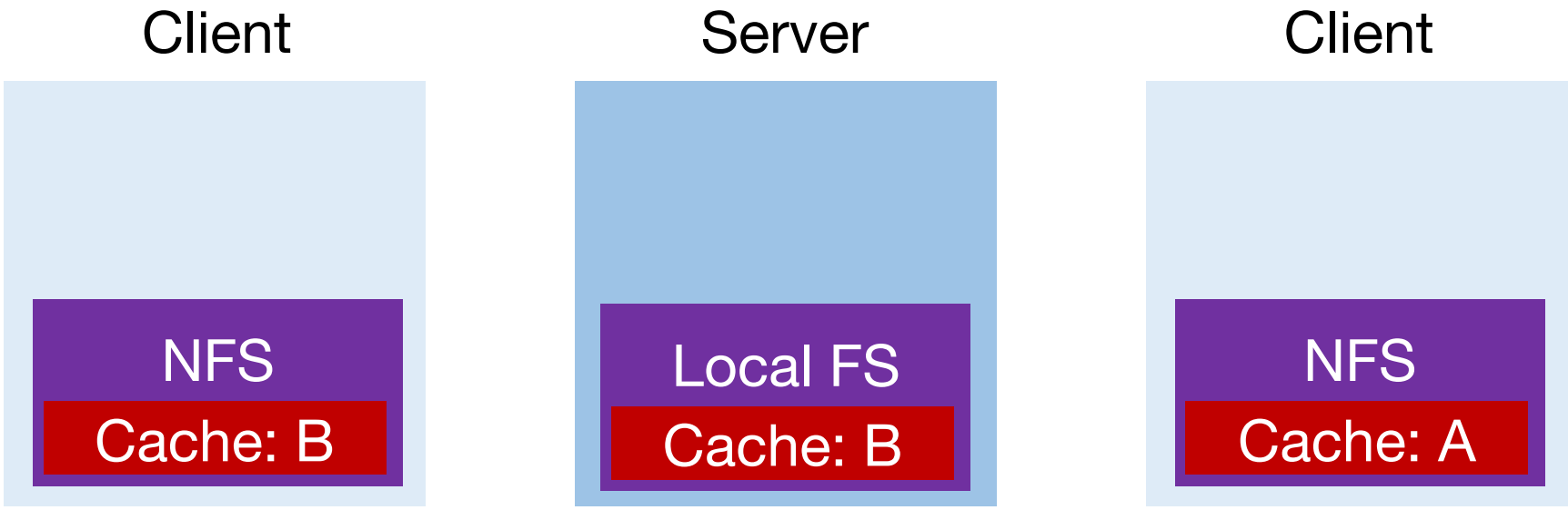
Cache


Client
Server
Client
NFS
Cache: B
flush
Local FS
Cache: B
NFS
Cache: A

Cache

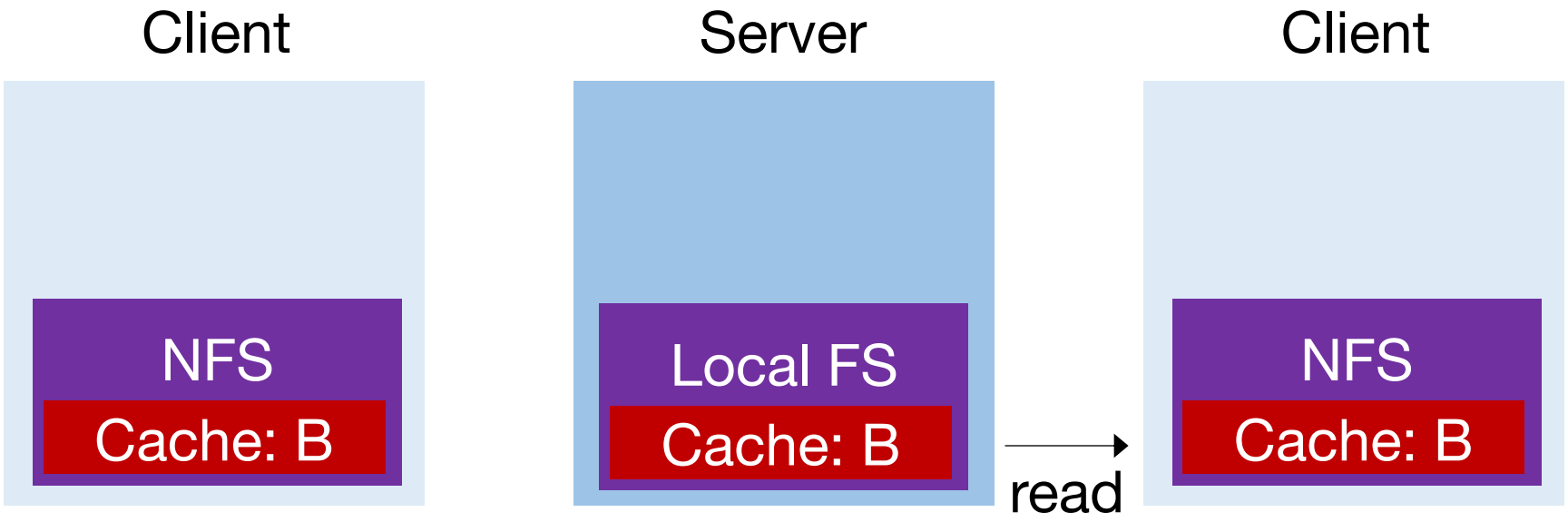


Cache



"Stale cache" problem: client doesn't have latest

Cache


Client
Server
Client
NFS
Cache: B
Local FS
Cache: B
read
NFS
Cache: B

Problem 1: Update visibility

- A client may buffer a write
- How can server and other clients see it?
- NFS solution: flush on fd close
  - Not quite like UNIX

Problem 2: Stale cache

- A client may have a cached copy that is obsolete
- How can we get the latest?

Problem 2: Stale cache

- A client may have a cached copy that is obsolete
- How can we get the latest?
- If we weren't trying to be stateless, server could push out update in a timely manner

Problem 2: Stale cache

- A client may have a cached copy that is obsolete
- How can we get the latest?
- If we weren't trying to be stateless, server could push out update in a timely manner
- NFS solution: clients recheck if cache is current before using it

Stale cache solution

- Client caches metadata records when data was fetched

- Before it is used, client does a **`stat`** request to server
  - Gets last modified timestamp
  - Compares to cache
  - Refetches if necessary

Measure then build

- NFS developers found `stat` accounted for 90% of server requests

- Why? Because clients frequently recheck cache

Reducing stat calls

- Solution: cache results of `stat` calls

- Why is this a terrible solution?

Reducing stat calls

- Solution: cache results of `stat` calls
- Why is this a terrible solution?
- Make the stat cache entries expire after a given time (say 3 seconds)
- Why is this better than putting expirations on the regular cache?

Summary

- Robust APIs are often:
  - **Stateless**: servers don't remember clients
  - **Idempotent**: doing things twice never hurts

- Supporting existing specs is a lot harder than building from scratch!

- Caching is hard! Caching is harder in distributed systems, especially with crashes.