Week 13

Handling Crashes and Performance (Part 2)

Oana Balmau

March 30, 2023

Class Admin

Week 13 **File Systems**	mar 27 C Review: Advanced debugging	mar 28 **Handling Crashes & Performance (1/2)** Optional reading: OSTEP Chapters 38, 43	mar 29	mar 30 **Handling Crashes & Performance (2/2)** • ~~Grades released for Exercises Sheet~~ • Practice Exercises Sheet: File Systems	mar 31
Week 14 **Advanced Topics**	apr 3 No lab. Work on Assignment 3 ~~**Memory Management Assignment Due**~~	apr 4 **Advanced topics: Virtualization** **On zoom. Do not come to class.** **Invited Speaker: Dr. Stella Bitchebe**	apr 5	apr 6 **Advanced topics: Operating Systems Research** **On zoom. Do not come to class.** **Invited Speaker: Dr. Stella Bitchebe** Grades released for Exercises Sheet	apr 7
Week 15 **Wrap-up**	apr 10 No Lab. Prepare for end-of-semester. **Memory Management Assignment Due**	apr 11 **End-of-semester Q&A**– not recorded **Last class!**	apr 12	~~apr 13~~ ~~**End-of-semester Q&A – not recorded.**~~	apr 14 Grades released for Memory Management Assignment

Next week, both lectures on Zoom, by invited speakers.
Zoom link: https://mcgill.zoom.us/j/85002140998

Distributed file systems

Distributed file systems

- What is a distributed system?

A distributed system is one where a machine I've never heard of can cause my program to fail

— [*Leslie Lamport*]

Distributed file systems

- What is a distributed system?

- More than 1 machine working together to solve a problem

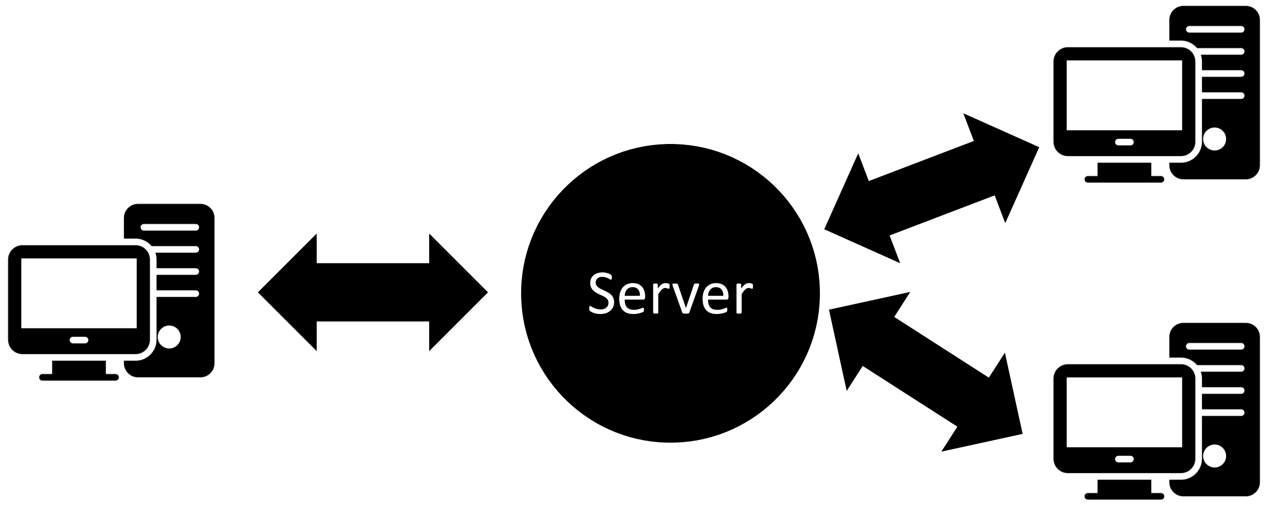
- Advantages
  - More computing power
  - More storage capacity
  - Fault tolerance
  - Data sharing

Two types of distributed systems

- Client/Server model
- Peer-to-peer model

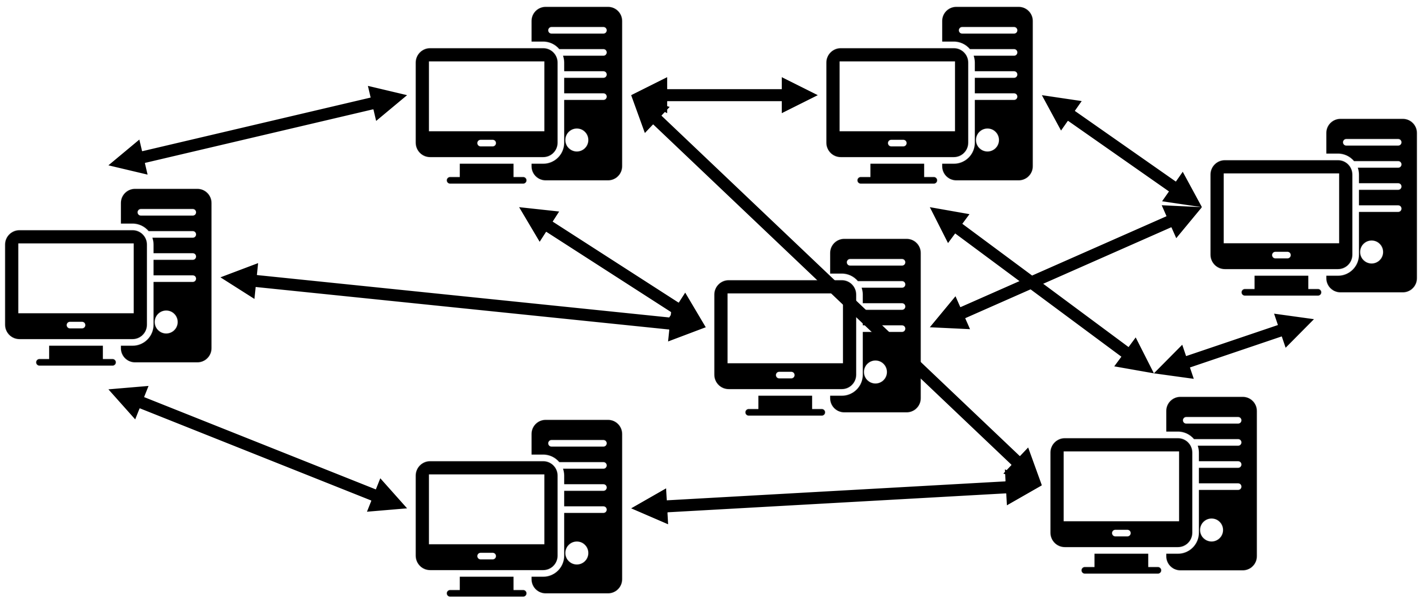
Client/Server model

- One or more server provides services to clients
- Clients makes remote procedure calls to server
- Server serves requests from clients



Peer-to-peer model

- Each computer acts as a peer
- No hierarchy or central point of coordination
- All-way communication between peers through gossiping



The promise of distributed systems

- Availability
- Fault-tolerance
- Scalability
- Transparency

Availability

Proportion of time system is in functioning condition

→ One machine goes down, use another

Fault-Tolerance

System has well-defined behaviour when fault occurs

→ Store data in multiple locations

Scalability

Ability to add resources to system to support more work

→ Just add machines when need more storage/processing power

Transparency

The ability of the system to mask its complexity behind a simple interface

Distributed File Systems

- File systems are great use case for distributed systems

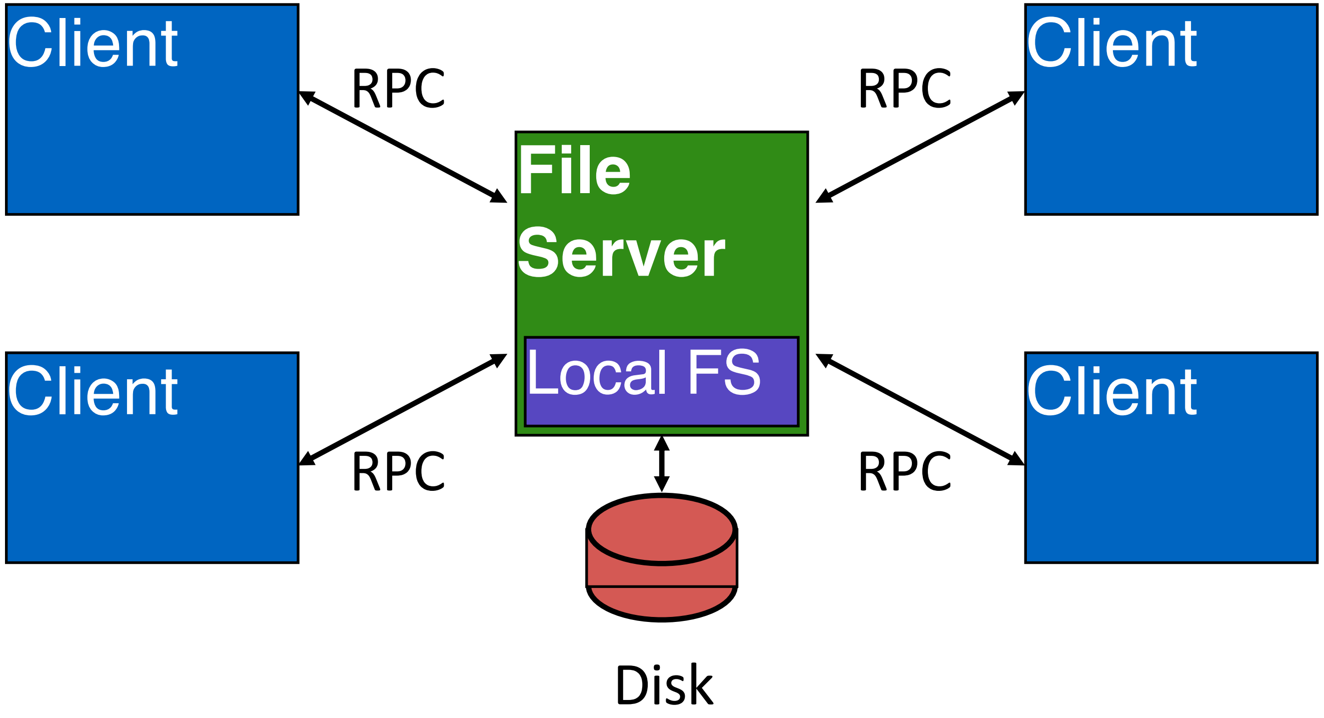
- Local FS:
  processes on same machine access shared files

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

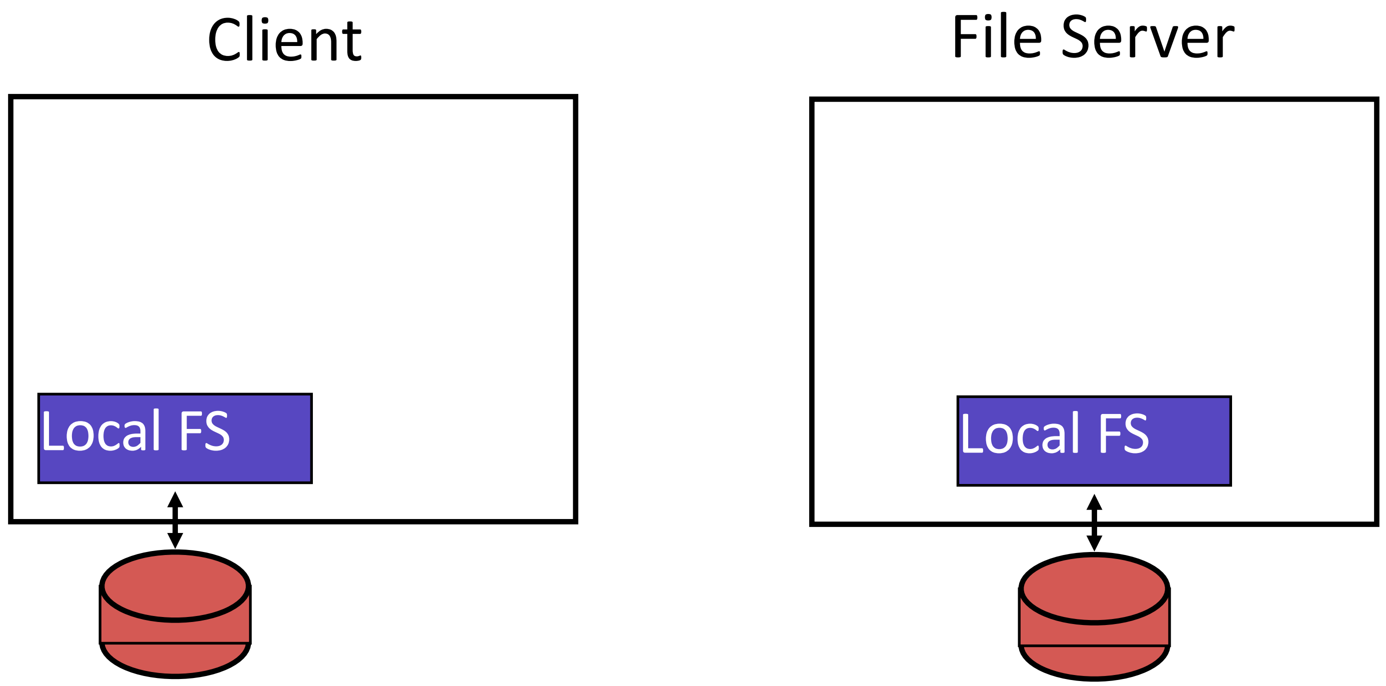
Goals for distributed file systems

- Fast + simple crash recovery
  - both clients and file server may crash
- Transparent access
  - can’t tell accesses are over the network
  - normal UNIX semantics
- Reasonable performance

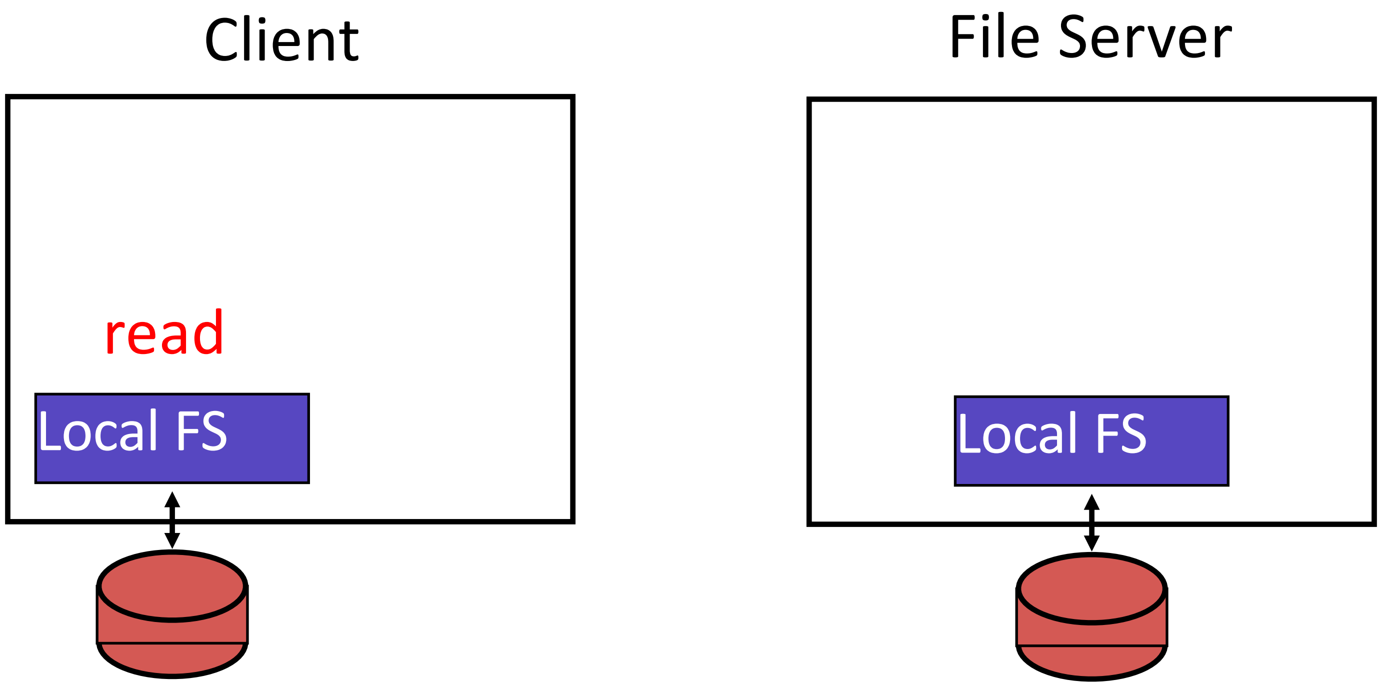
Networked File System



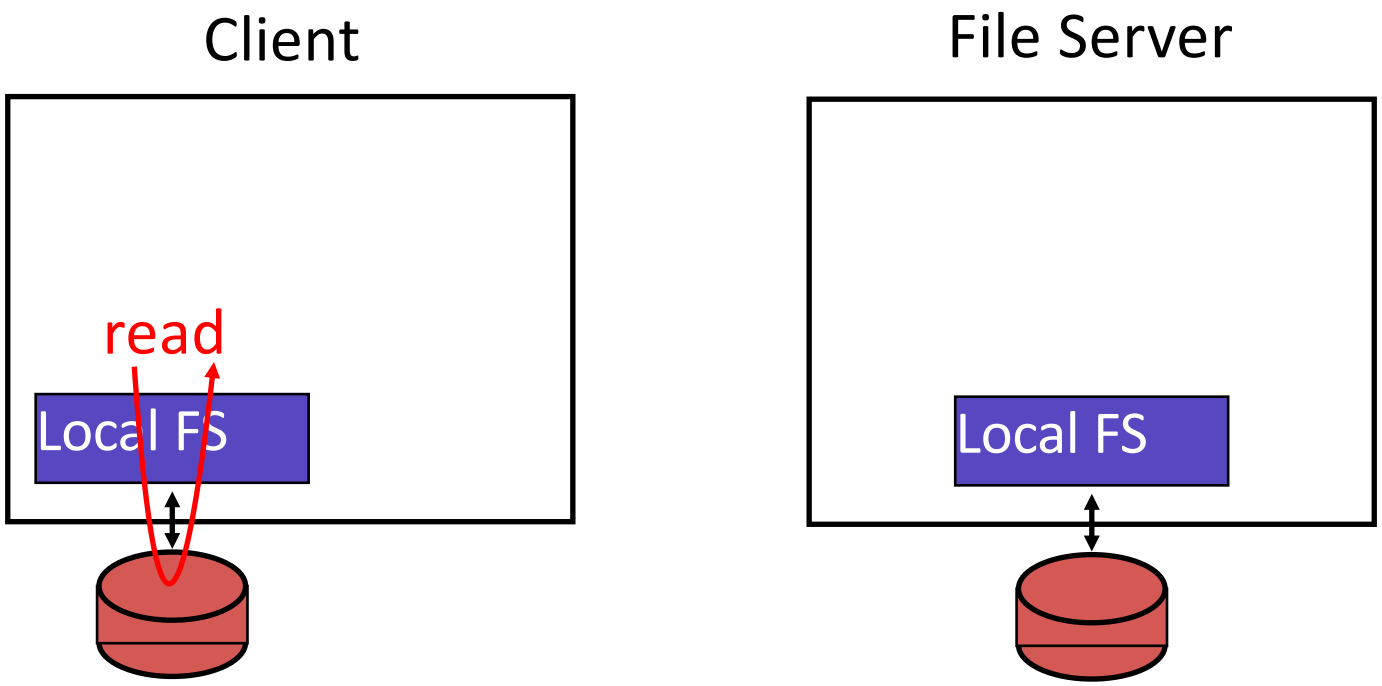
General Strategy: Export FS



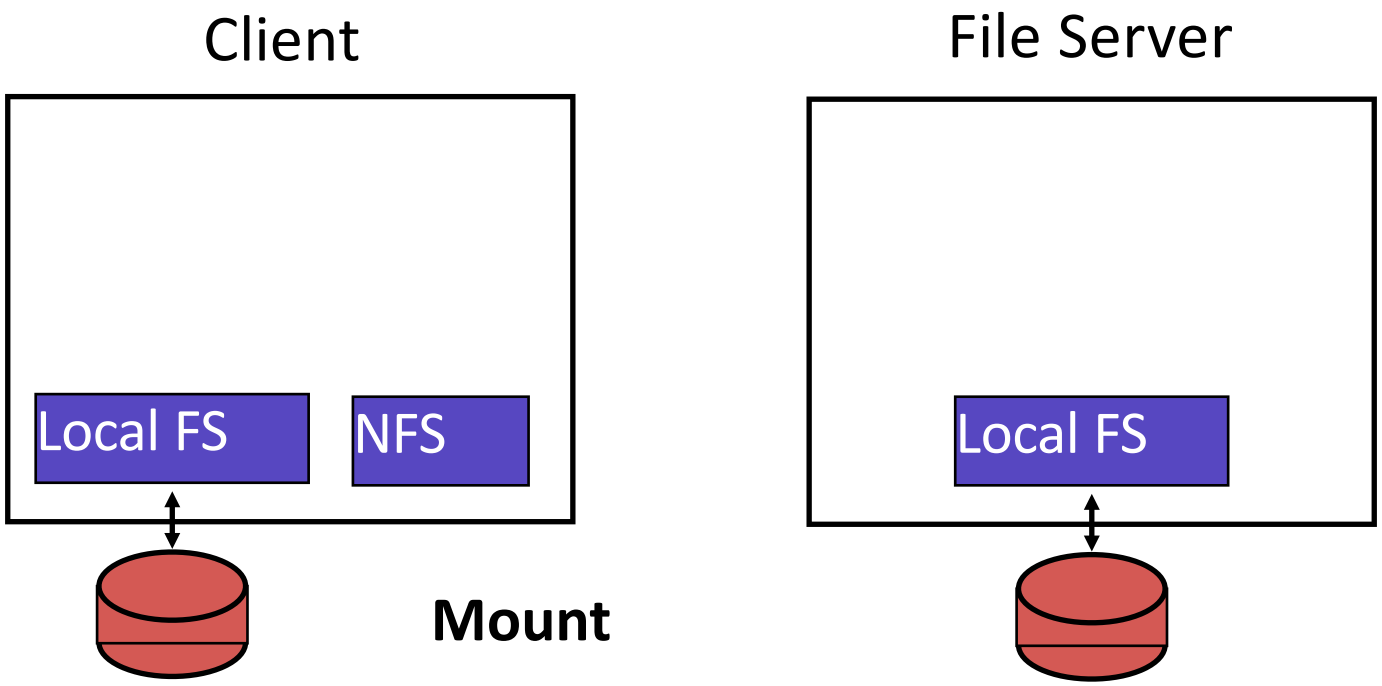
General Strategy: Export FS

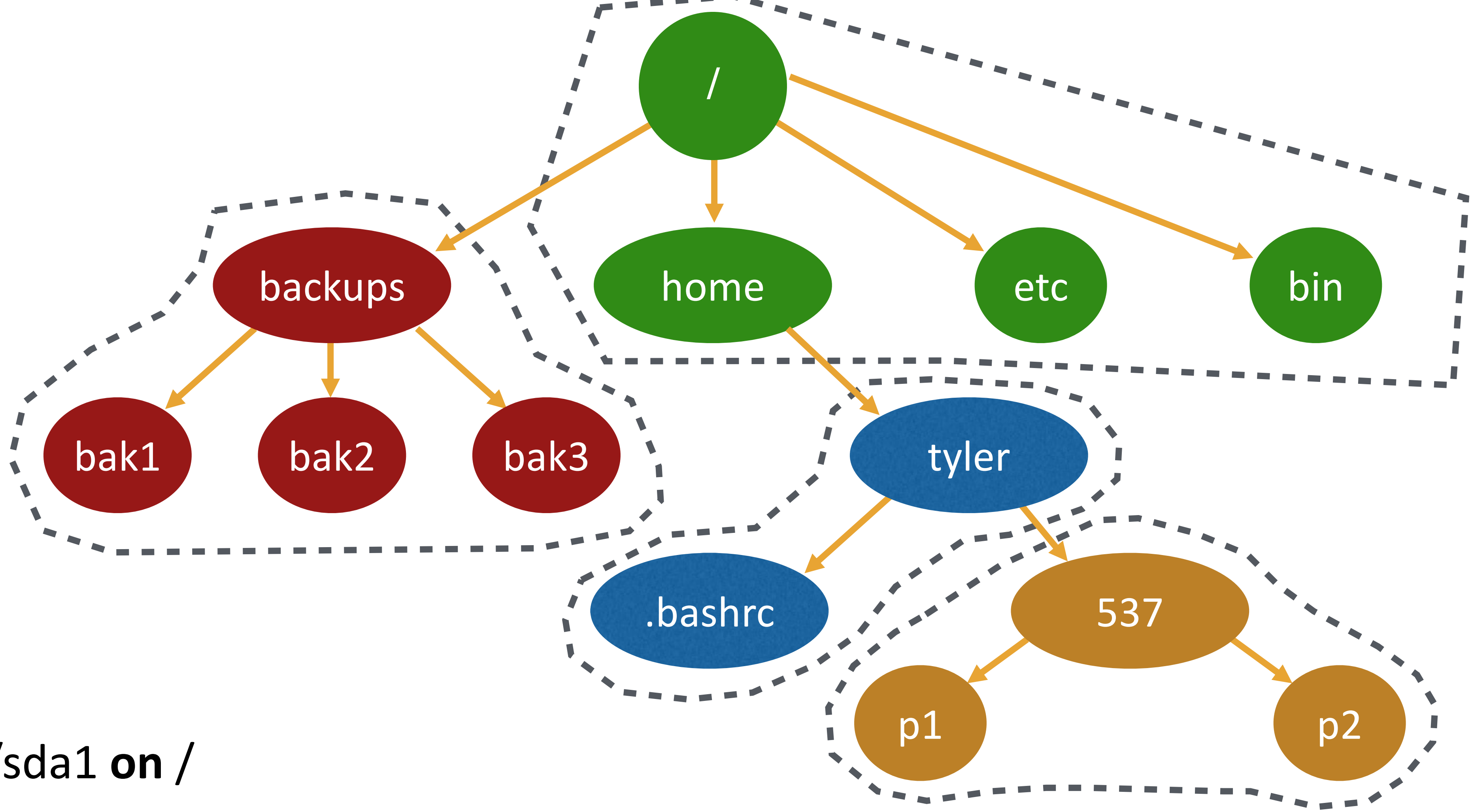


General Strategy: Export FS



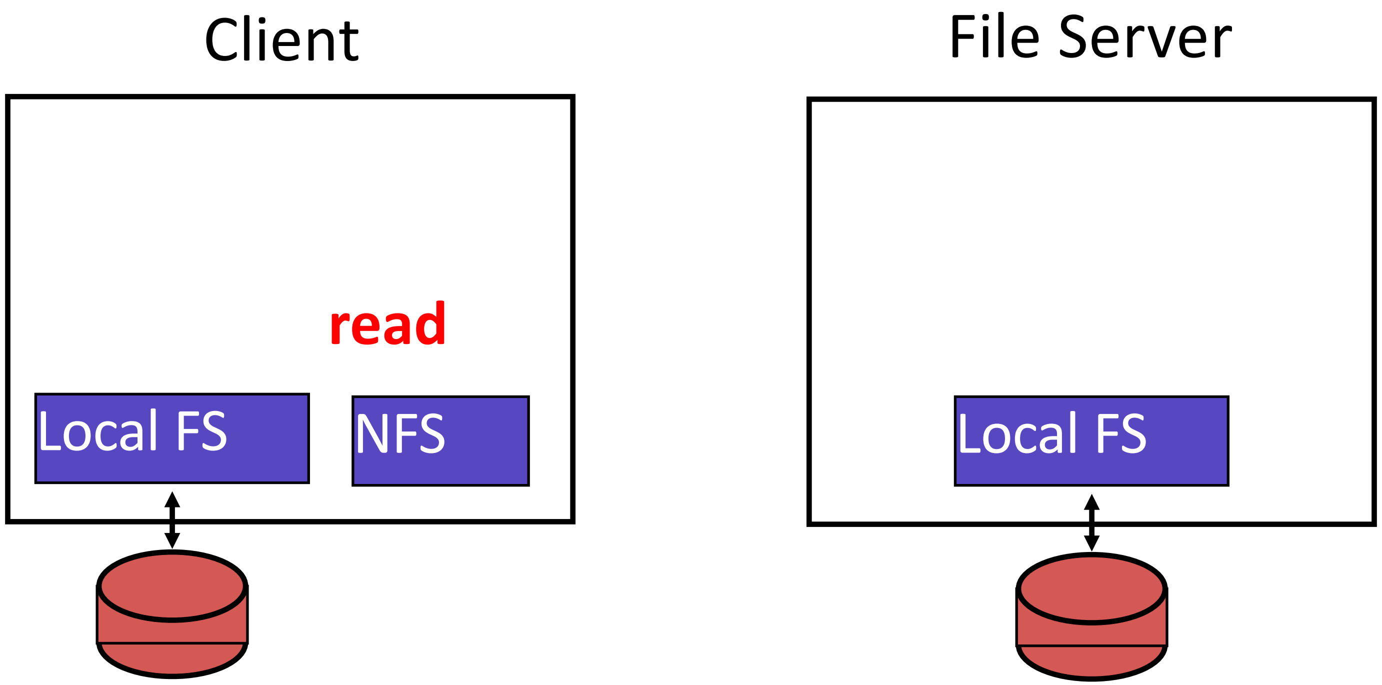
General Strategy: Export FS



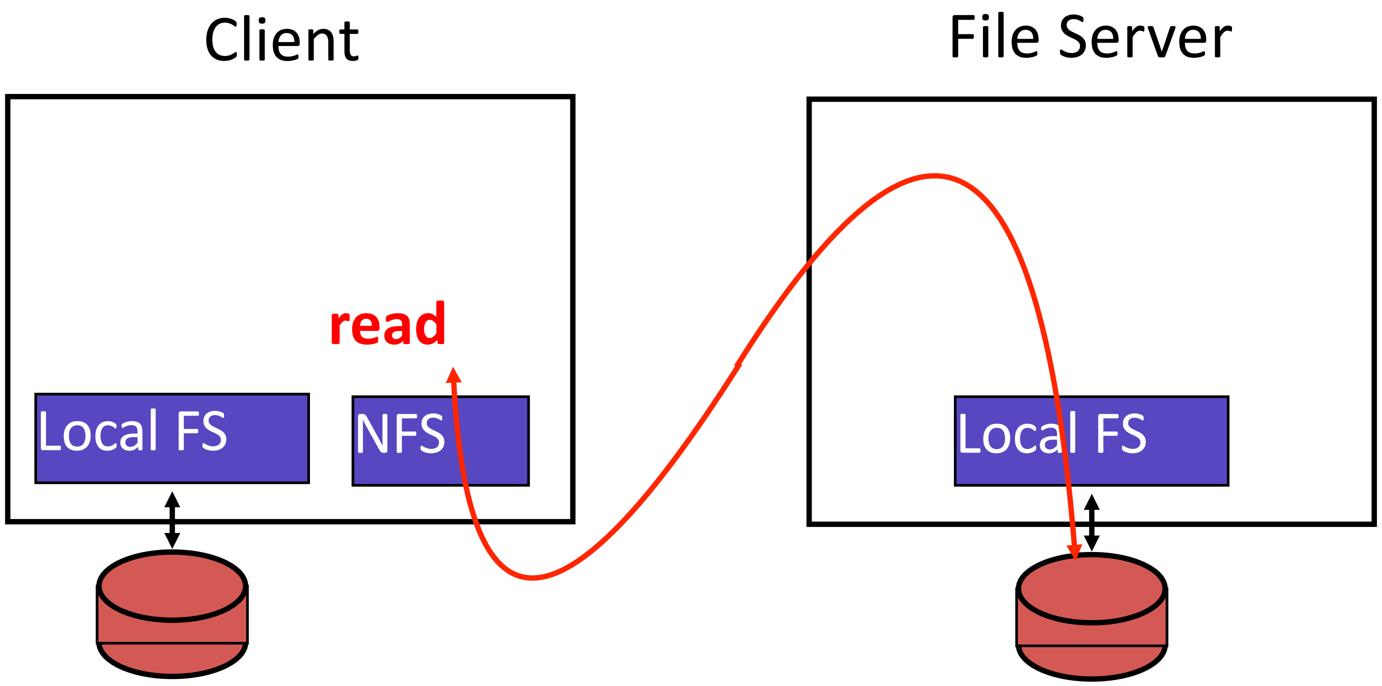


- /dev/sda1 **on** /
- /dev/sdb1 **on** /backups
- NFS **on** /home/tyler

General Strategy: Export FS



General Strategy: Export FS



Network FS API

- Want to support Unix API
  - Open(), read(), write(), close()

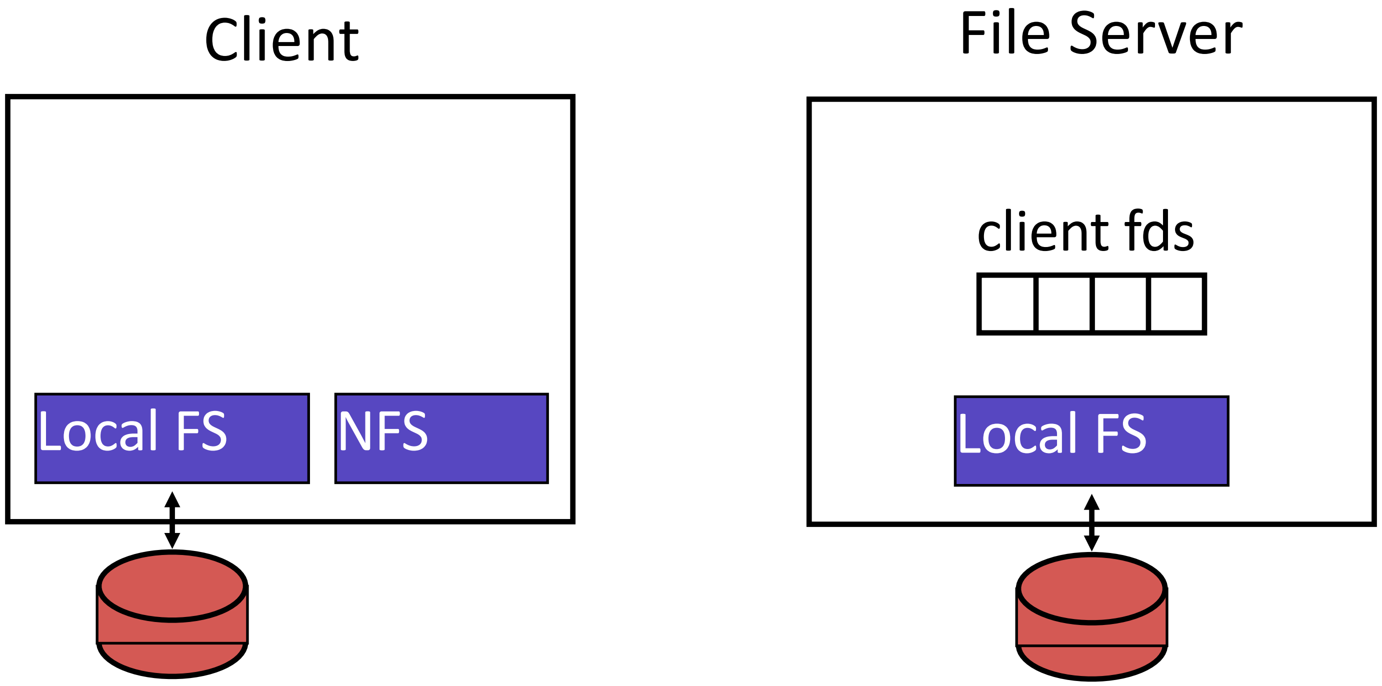
- Wrap regular UNIX system calls using RPC

Wrap regular UNIX system calls using RPC

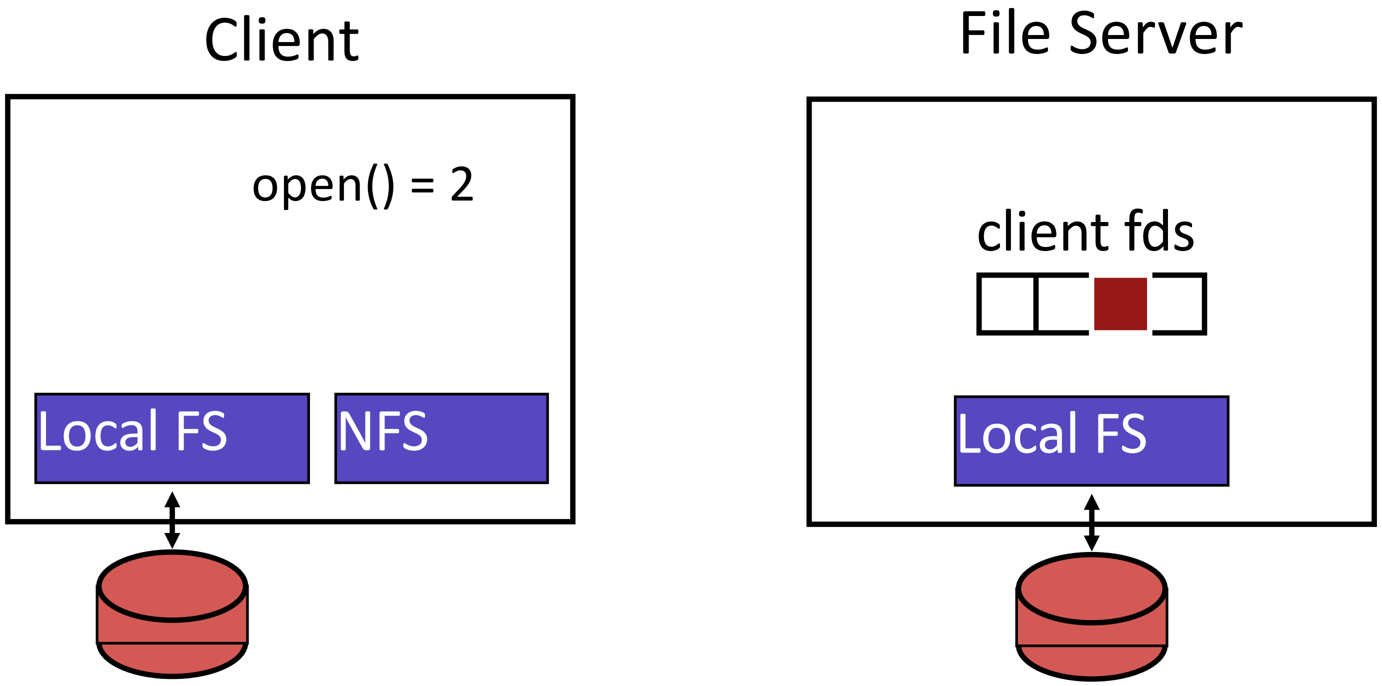
- open() on client calls open() on server
- open() on server returns fd back to client

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

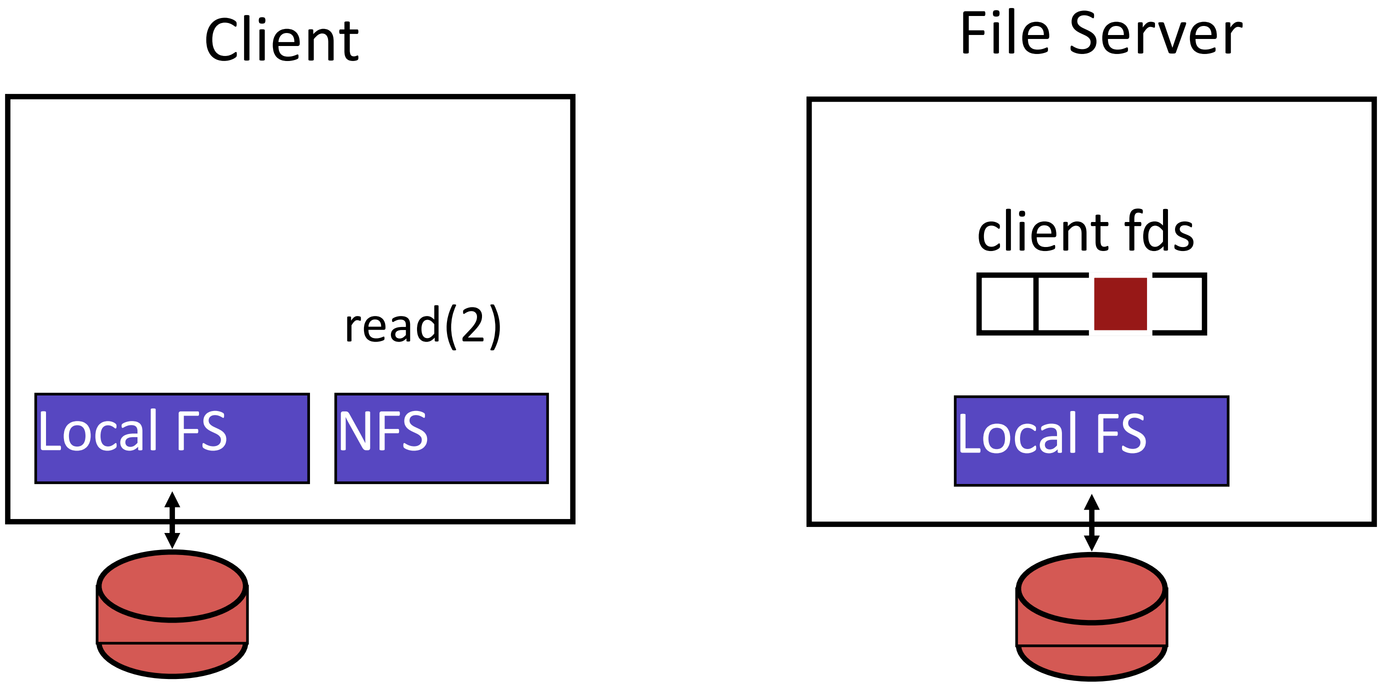
File Descriptors



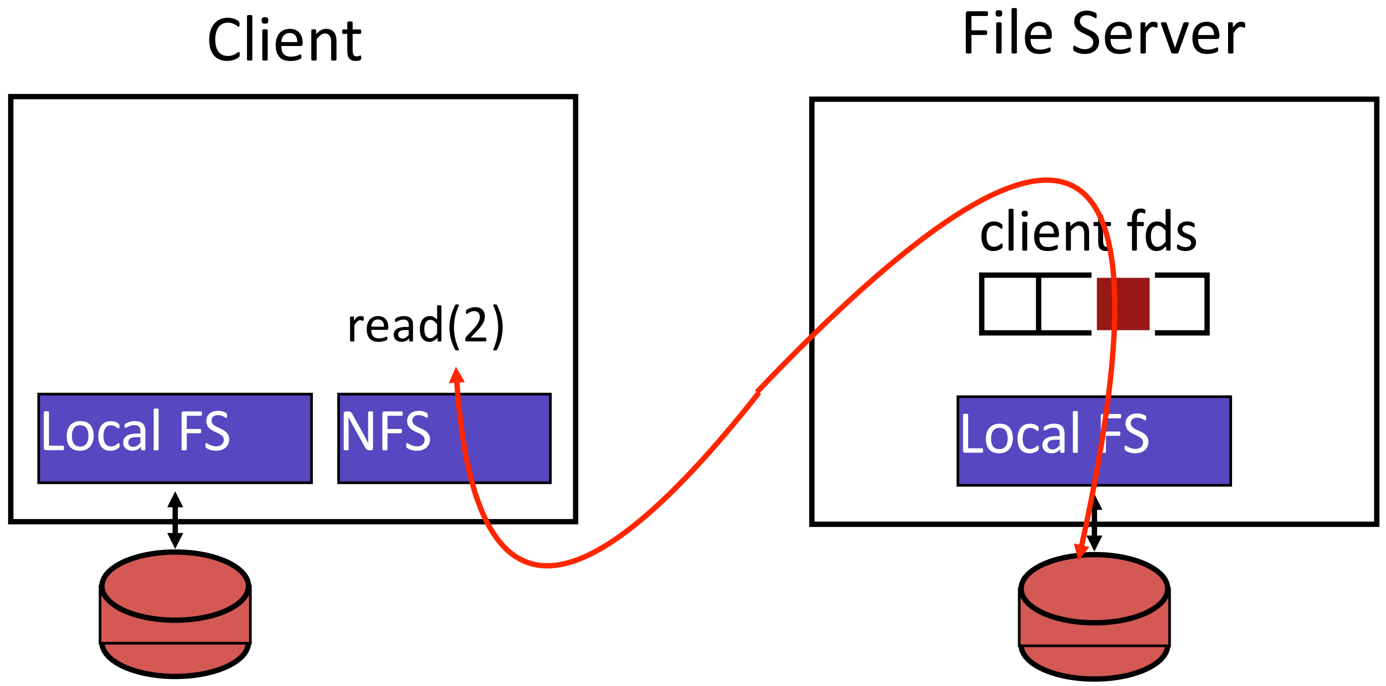
File Descriptors



File Descriptors



File Descriptors



Network FS Advantages

- Data sharing across clients
- Centralized administration
- Security

Network FS Problems

- Handling crashes
- Slow to go over the network for every request

Imagine server crashes and reboots during reads

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

← Server crash!

Want to act like a slow read from the client perspective

Potential solution

- Run crash recovery protocol on server-side

Problem with this solution: Too complex if server fails

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

Server crash!

Server needs to remember first read was done. Too complex to keep track of with many clients.

Problem with this solution cont'd: Too complex if client fails

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

client crash!

How will server know to get rid of fd?

Better solution

- Stateless protocol with

- Idempotent operations

Stateless protocol

- Server maintains no state about clients
  - All the relevant information is encoded in the client request
    - Remember: parallel to the handling of RPCs

Stateless protocol: Put all info in requests

Use stateless protocol. Server maintains no state about clients.

Need API change. One possibility:

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

Specify path and offset each time. Server need not remember anything from clients.

Pros? Server can crash and reboot transparently to clients.

Cons? Too many path lookups.

Stateless protocol: Use Inode instead of path

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

Any correctness problems?

If file is deleted, the inode could be reused.

- Problem: Inode not guaranteed to be unique over time.

Stateless protocol: Use File Handles (FH)

File Handle = <volume ID, inode #, **generation #**>
Opaque to client (client should not interpret internals)

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

Any correctness problems?

If file is deleted, the inode could be reused.

- Problem: Inode not guaranteed to be unique over time.

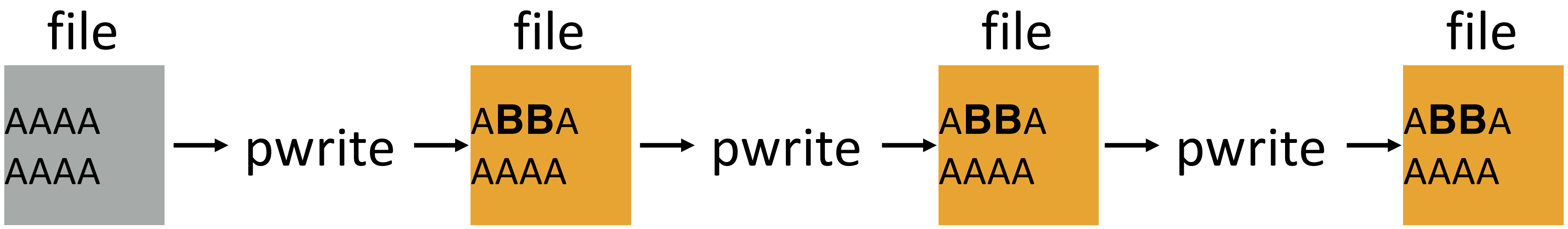
Idempotent Operations

- f() is idempotent if f() has the same effect of f(); f(); f(); …. f()
- Design API so that it is ok to execute functions more than once.
  - Why?
    - useful for correctness during crashes
    - Can retry in case of slowness/crash

pread is idempotent

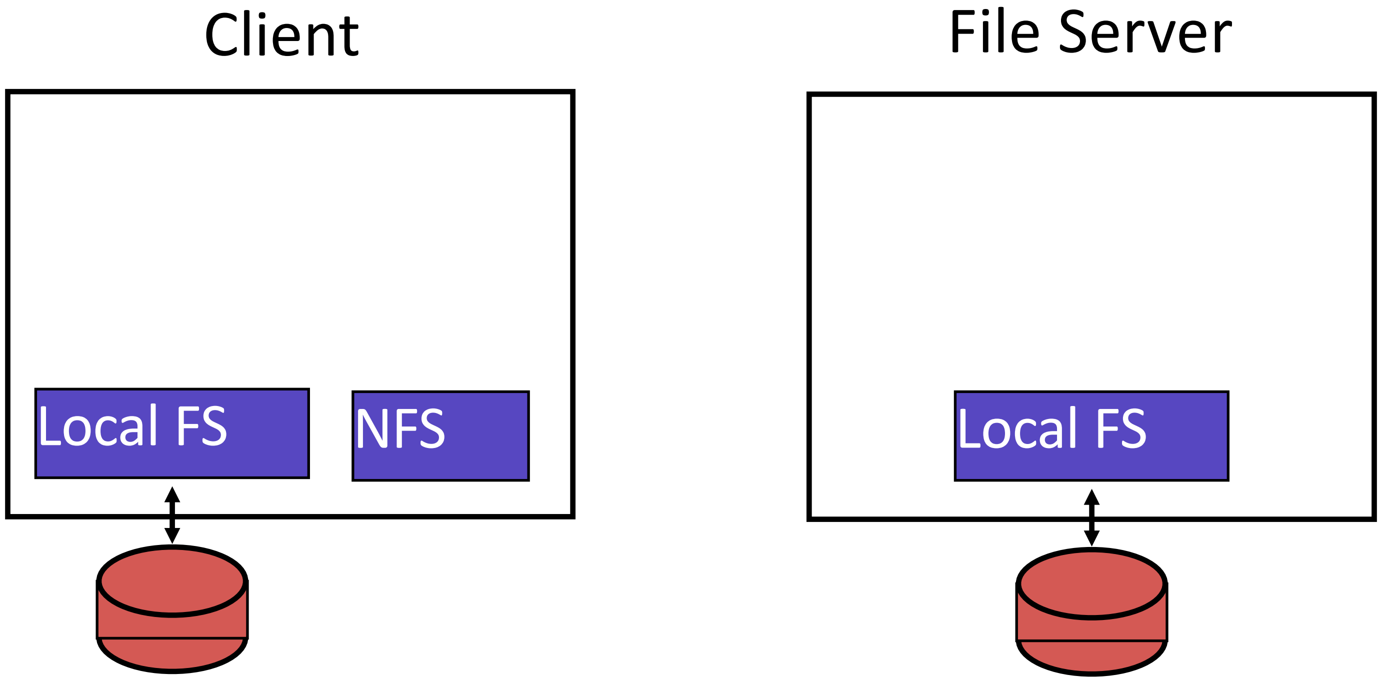
- **pread**(<u>fh</u>, <u>buf</u>, <u>size</u>, <u>offset</u>);
- Reading does not change the file
- What about pwrite?

pwrite is idempotent




```
pwrite(fh, buf, size, offset);
```

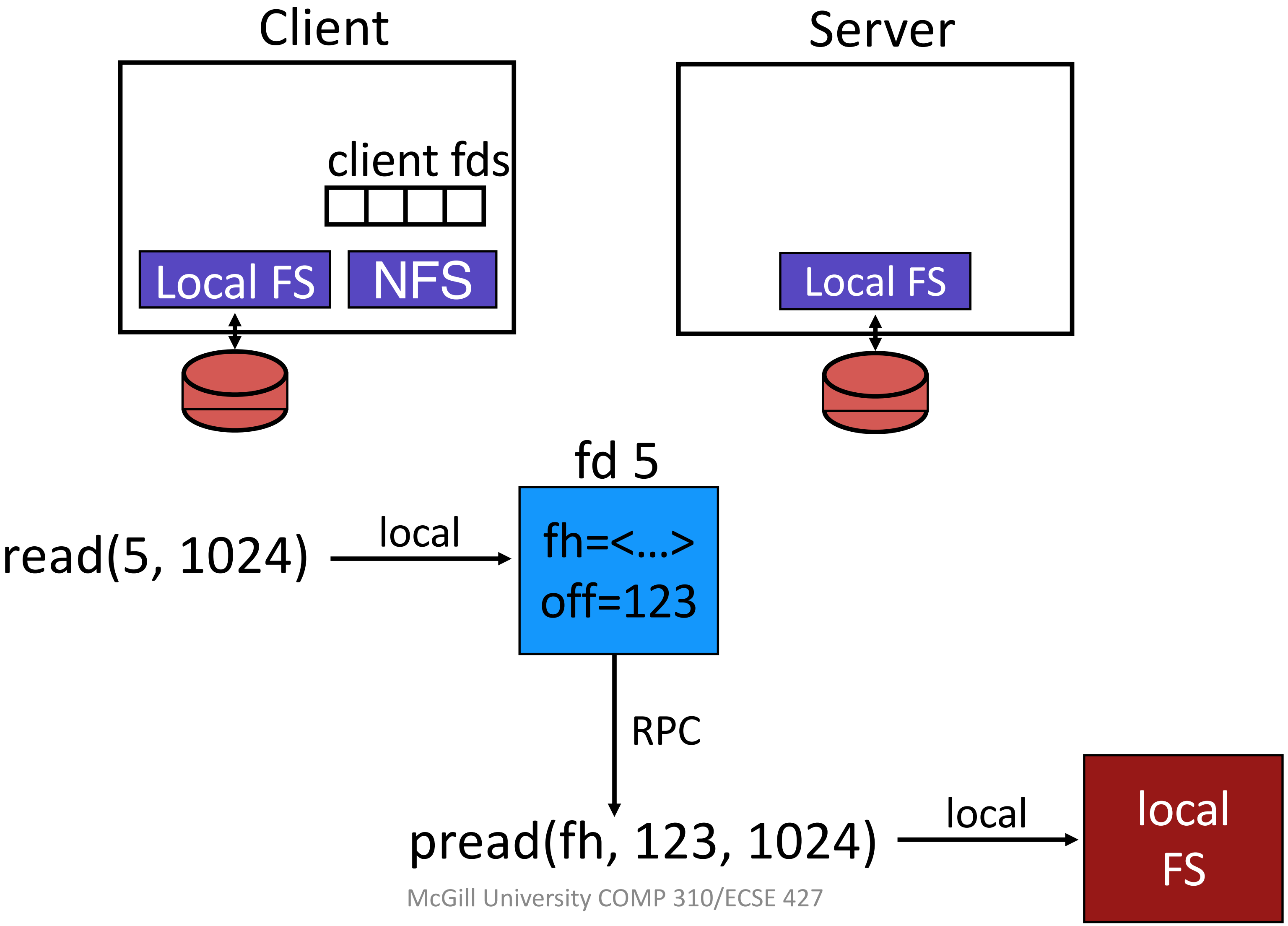
Putting it all together



Build normal UNIX API
On top of idempotent, RPC-based API

Handle RPCs

Putting it all together



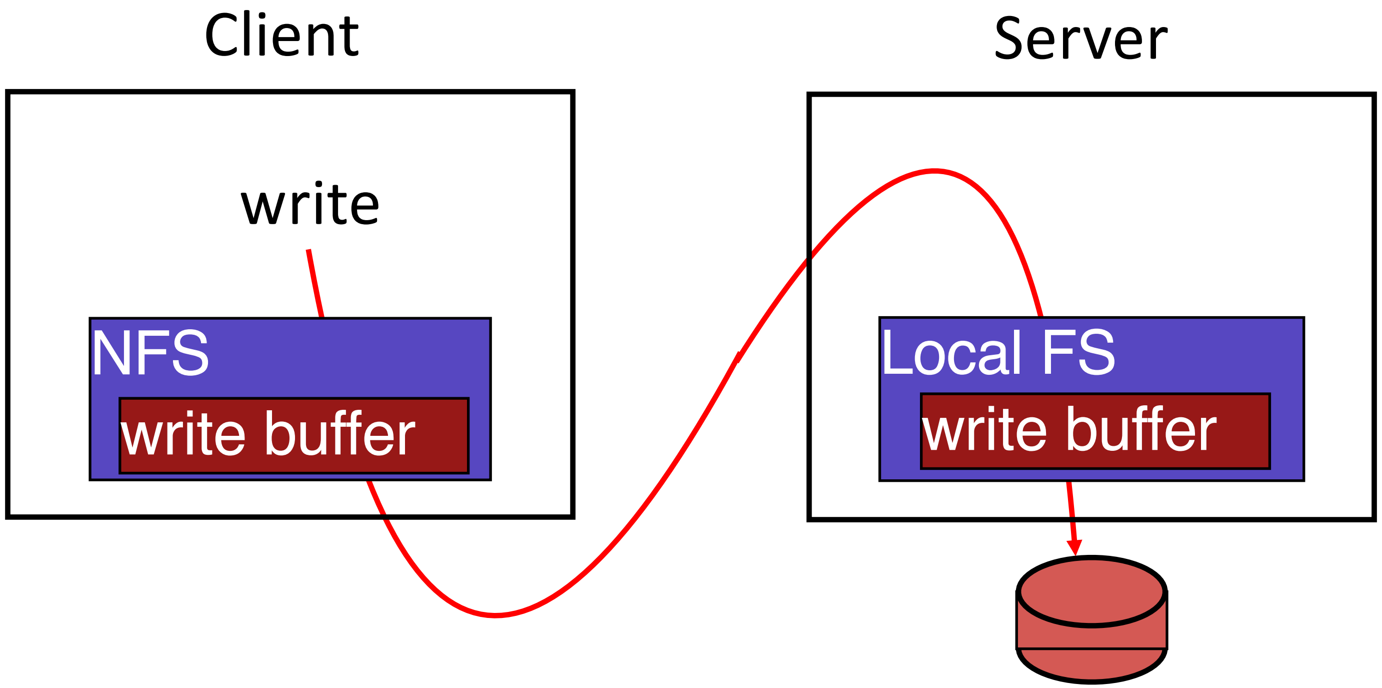
Network FS Problems

- Handling crashes
  - Solutions: stateless protocol, idempotent operations
- **Slow to go over the network for every request**

Slow to go over the network for every request

- Write buffering
- Caching

Write Buffers



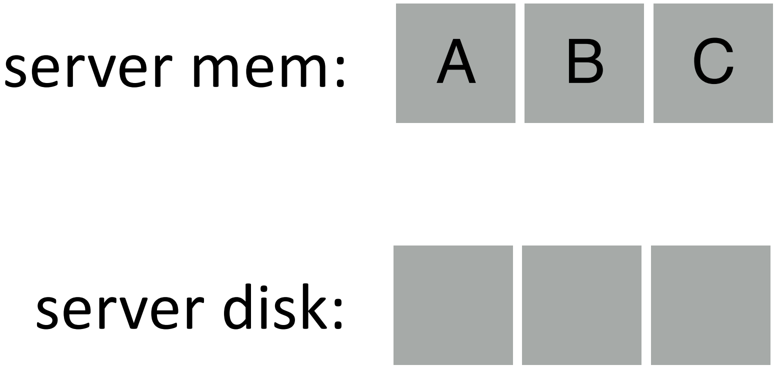
server acknowledges write before write is pushed to disk;

what happens if server crashes?

Server Write Buffer Lost

`client:`

```
write A to 0
write B to 1
write C to 2
```



server acknowledges write before write is pushed to disk

Server Write Buffer Lost

```
client:

 write A to 0
 write B to 1
 write C to 2
```

server mem:

A	B	C

server disk:

A	B	C

server acknowledges write before write is pushed to disk

Server Write Buffer Lost

```
client:

 write A to 0
 write B to 1
 write C to 2

 write X to 0
```

server mem:	X	B	C

server disk:	A	B	C

server acknowledges write before write is pushed to disk

Server Write Buffer Lost

```
client:

 write A to 0
 write B to 1
 write C to 2

 write X to 0
```

server mem: X B C

server disk: X B C

server acknowledges write before write is pushed to disk

Server Write Buffer Lost

```
client:

 write A to 0
 write B to 1
 write C to 2

 write X to 0
 write Y to 1
```

server mem:	X	Y	C
server disk:	X	B	C

server acknowledges write before write is pushed to disk

Server Write Buffer Lost

```
client:

write A to 0
write B to 1
write C to 2

write X to 0
write Y to 1
```

server mem:

server disk: X B C

crash!

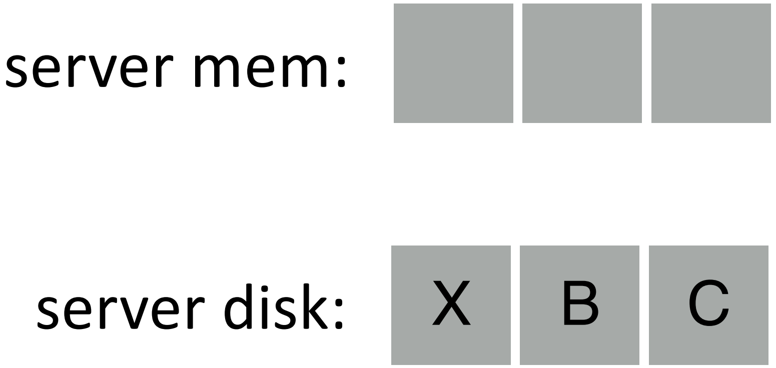
server acknowledges write before write is pushed to disk

Server Write Buffer Lost

```
client:

 write A to 0
 write B to 1
 write C to 2

 write X to 0
 write Y to 1
```



server acknowledges write before write is pushed to disk

Server Write Buffer Lost

```
client:

 write A to 0
 write B to 1
 write C to 2

 write X to 0
 write Y to 1
 write Z to 2
```

server mem: | | | Z |

server disk: | X | B | C |

server acknowledges write before write is pushed to disk

Server Write Buffer Lost

client:

```
write A to 0
write B to 1
write C to 2

write X to 0
write Y to 1
write Z to 2
```

server mem:

		Z

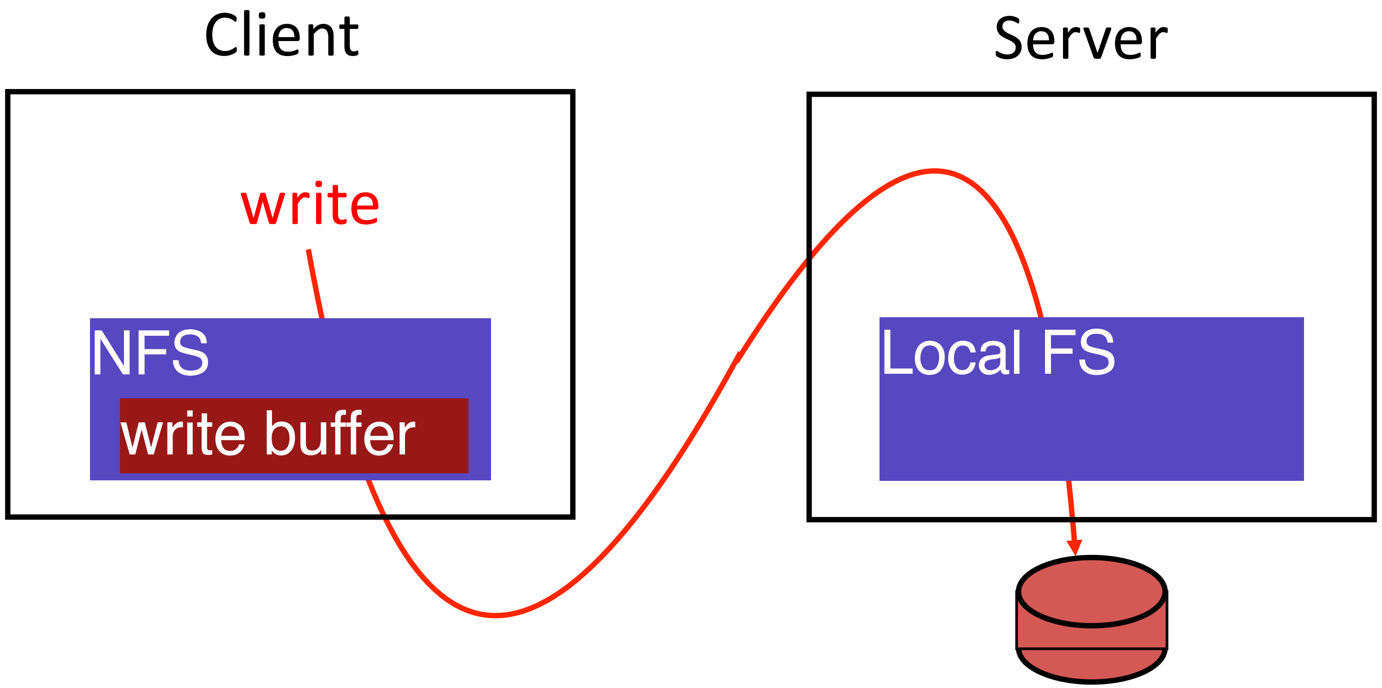
server disk:

X	B	C

Problem:

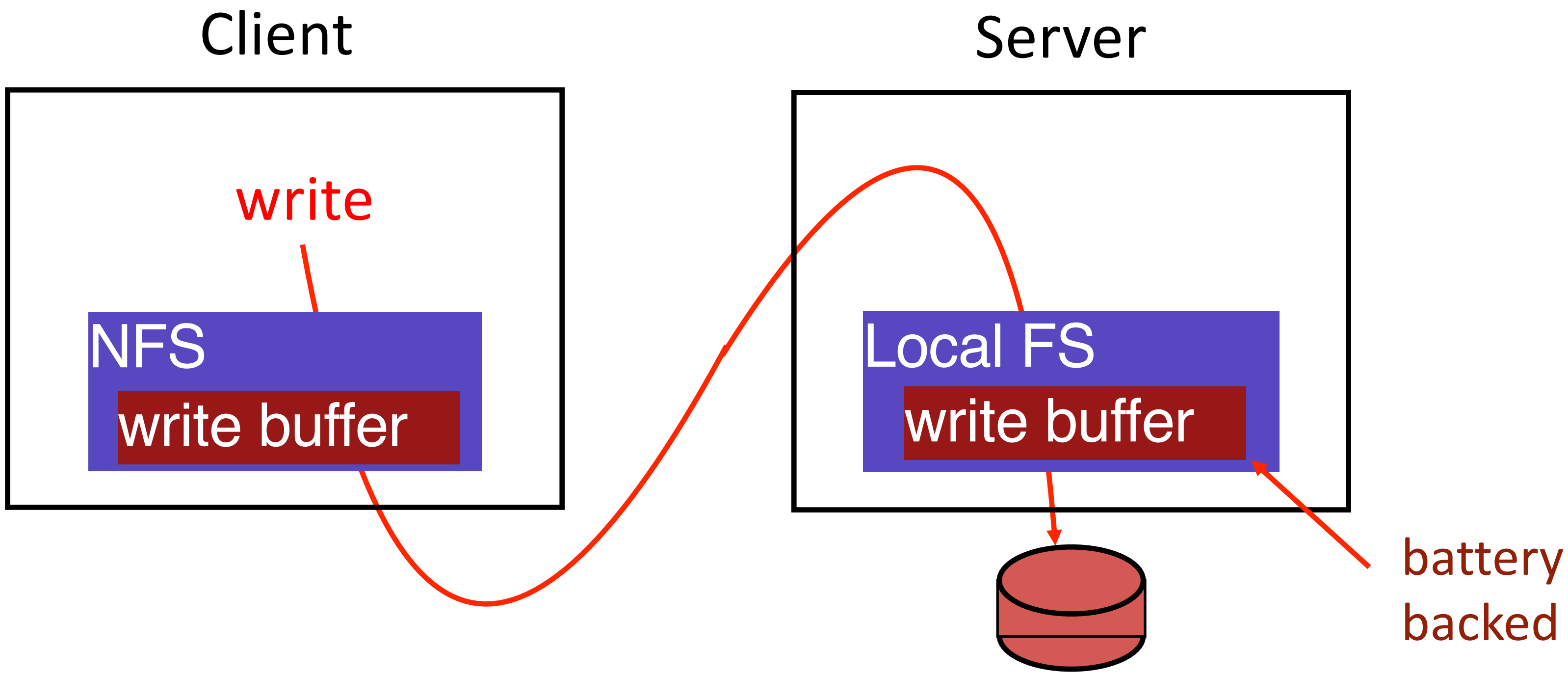
No write failed, but disk state doesn't match any point in time

Solution



Don’t use server write buffer
(persist data to disk before acknowledging write)
Problem: Slow!

Better Solution



Use persistent write buffer (more expensive)
Many systems implement this.

Caching

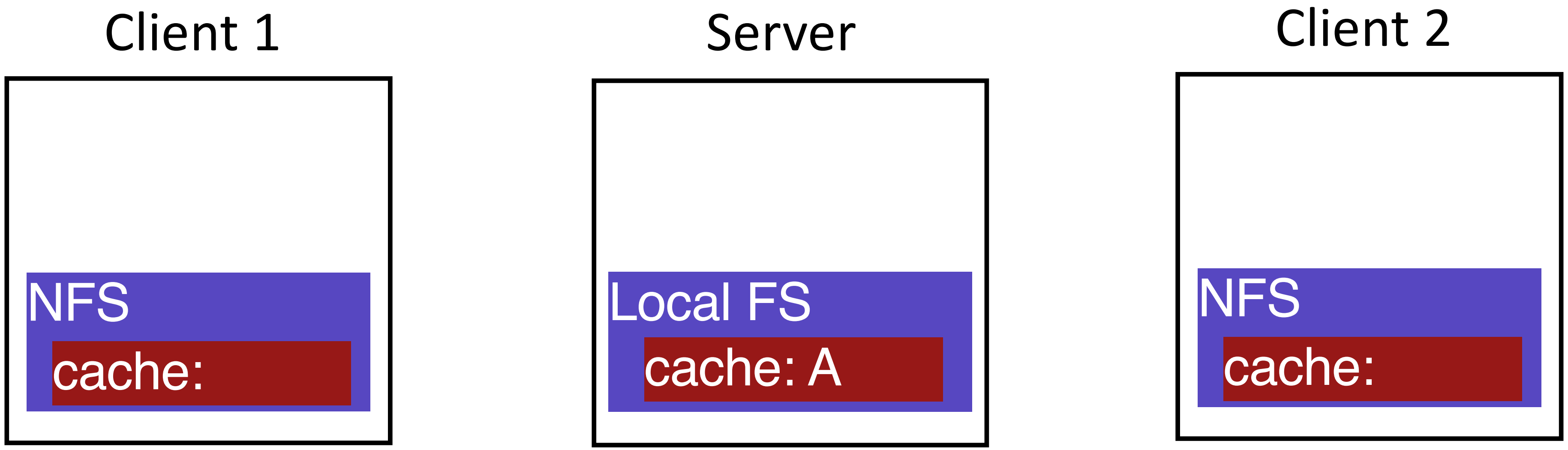
Cache Consistency

Networked FS can cache data in three places:

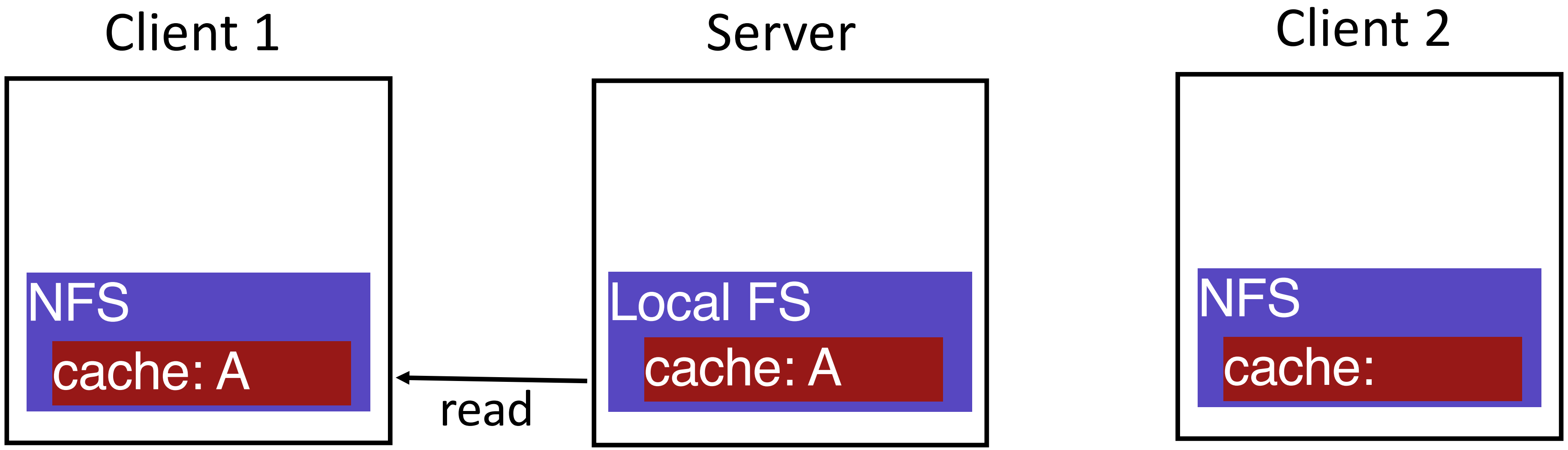
- server memory
- client disk
- client memory

How to make sure all versions are in sync?

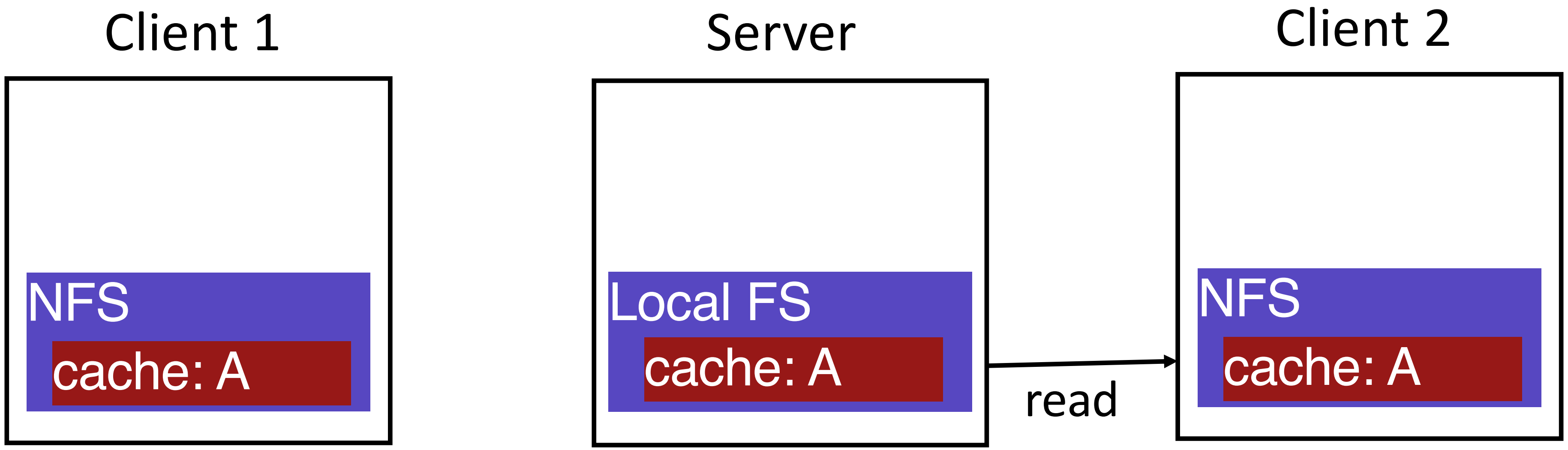
Distributed Cache



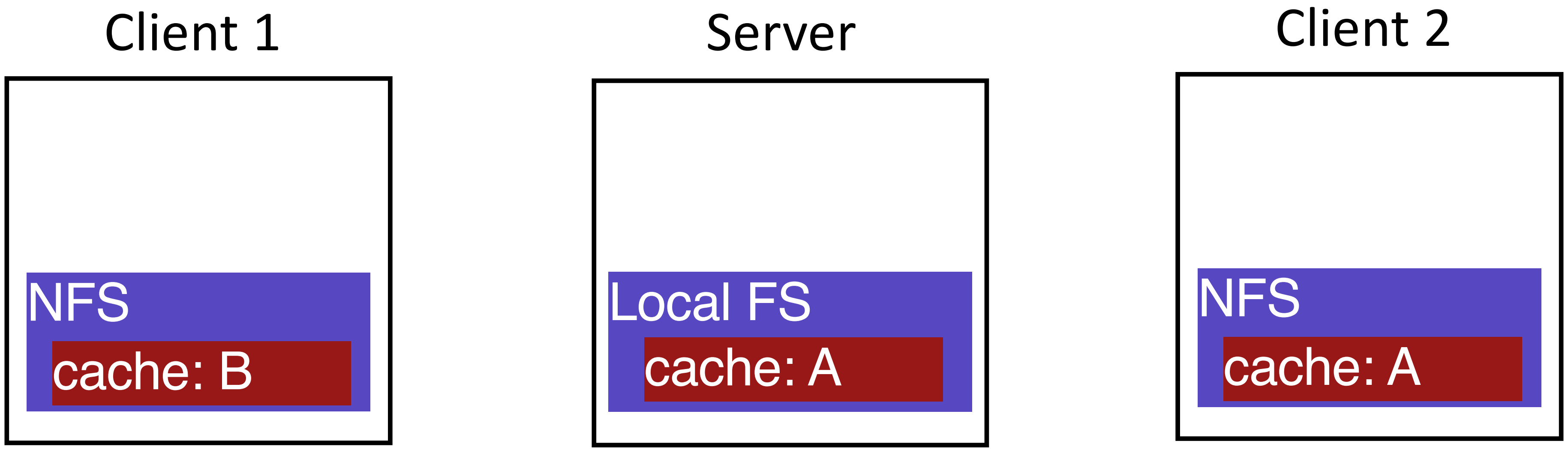
Distributed Cache



Distributed Cache



Update Visibility problem



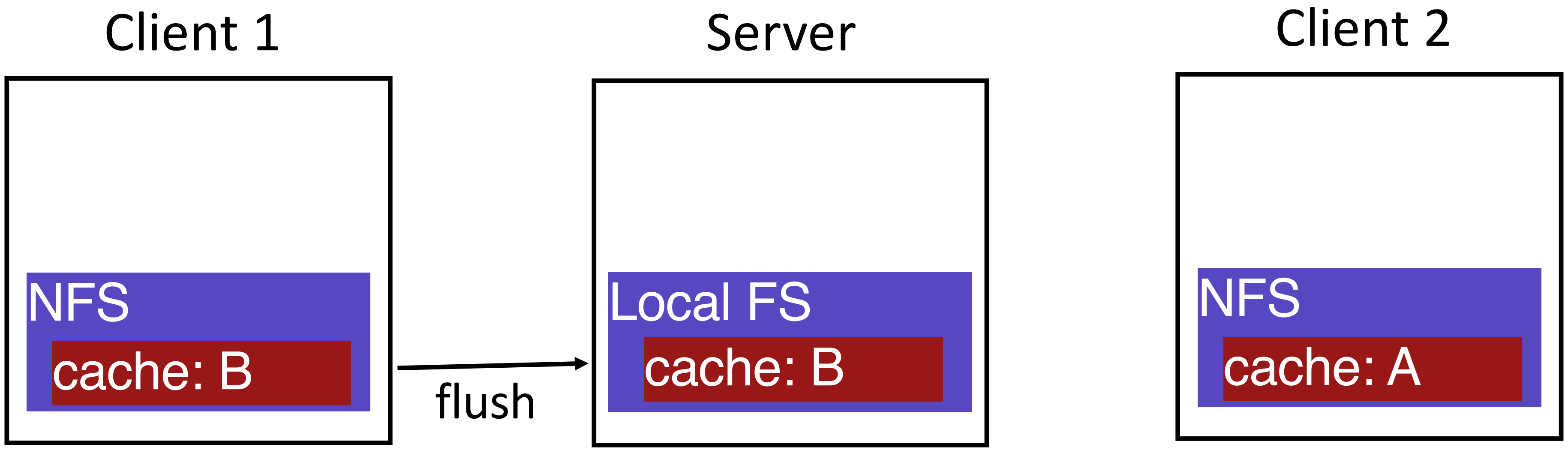
Server doesn’t have latest version.

What happens if Client 2 (or any other client) reads data?
Sees old version (different semantics than local FS)

Update Visibility Solution

- Client periodically flushes.
- When to flush?
  - on fd close

Stale Cache Problem



Client 2 doesn't have latest version

What happens if Client 2 reads data?
Sees old version (different semantics than local FS)

Stale Cache Solution

Clients recheck if cached copy is current before using data.

- 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)

- Scalability issue
  - In practice >90% of the requests are STAT calls

Networked FS Summary

- NFS handles client and server crashes very well
  - **stateless**: servers don't remember clients
  - **idempotent**: doing things twice never hurts

- Caching and write buffering in distributed systems.
- Scalability limitations as more clients call stat() on server

Google File System (GFS)

Google File System (GFS)

Google workload characteristics

- huge files (GBs); usually read in their entirety
- almost all writes are appends
- concurrent appends common
- high throughput is valuable
- low latency is not

Computing environment:

- 1000s of machines
- Machines sometimes fail (both permanently and temporarily)

Why not use Networked FS?

Scalability : Must store > 100s of Terabytes of file data

NFS only exports a local FS on <u>one machine</u> to other clients

GFS solution: store data on many server machines

Failures: Must handle temporary and permanent failures

NFS only recovers from temporary failure

- not permanent disk/server failure
- recovery means making reboot invisible
- technique: retry (stateless and idempotent protocol helps)

GFS solution: replication and failover (like RAID)

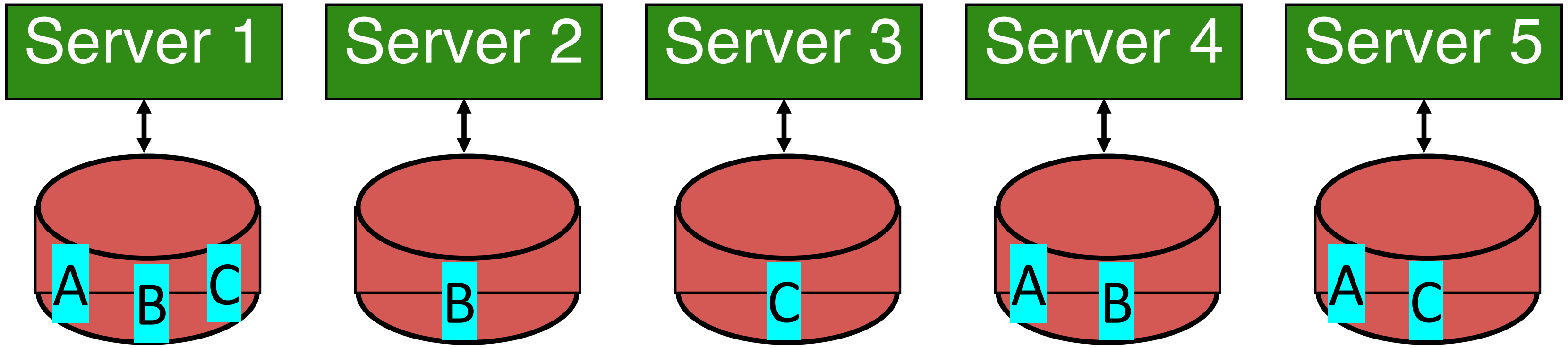
New File System: GFS

Google published details in SOSP2003

- Has evolved since then…

Open source implementation: Hadoop Distributed FS (HDFS)

1) Replication



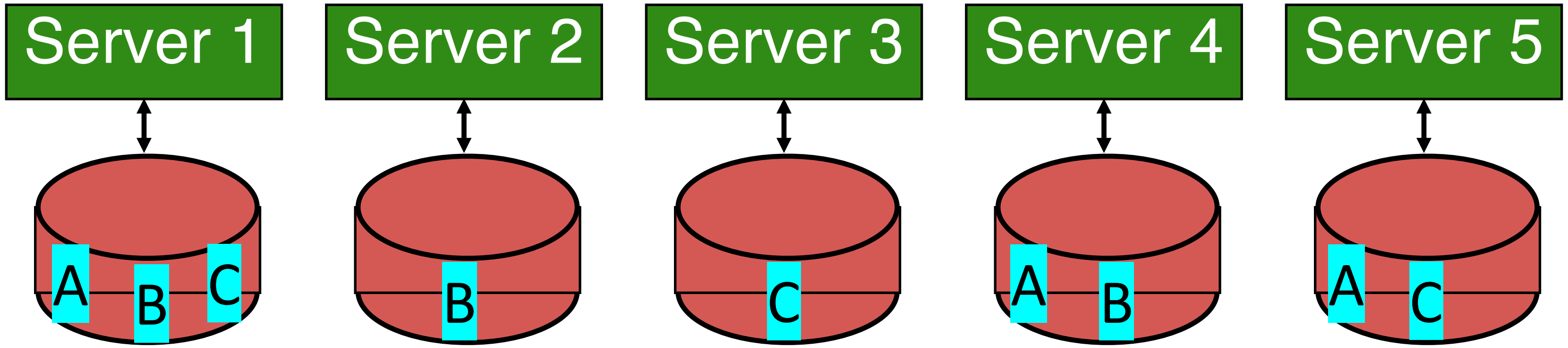
Each server holds "chunks" of data

Less structured than RAID (no static computation to determine locations)

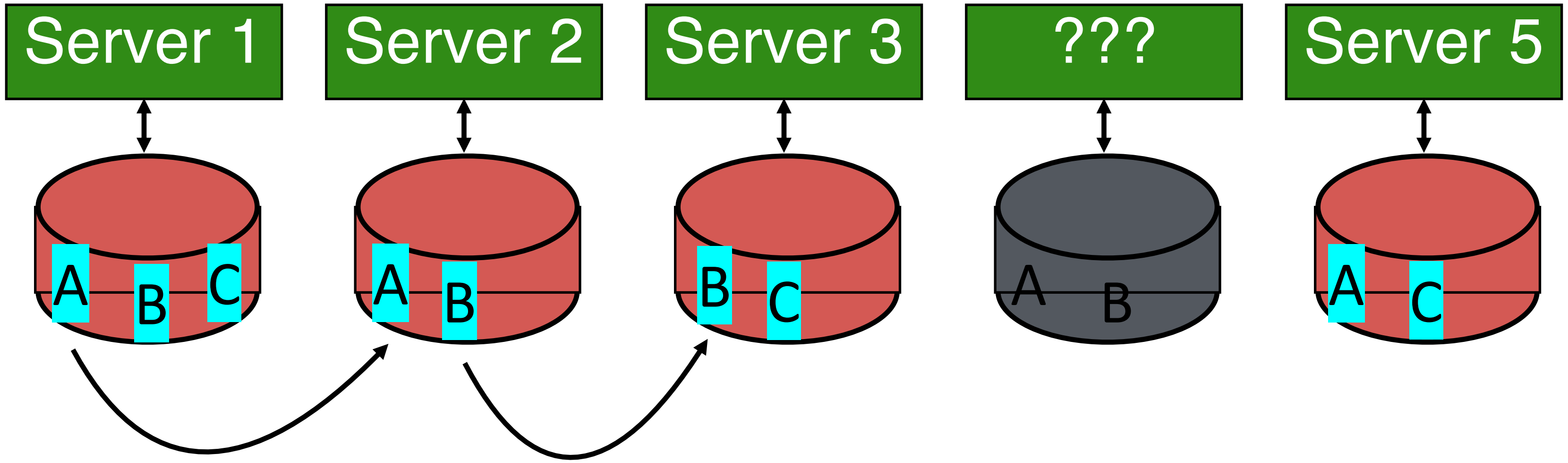
- machines come and go
- capacity may vary
- different data may have different replication levels (e.g., 3 vs 5 copies)

Problem: How to map logical to physical locations?

2) Recovery



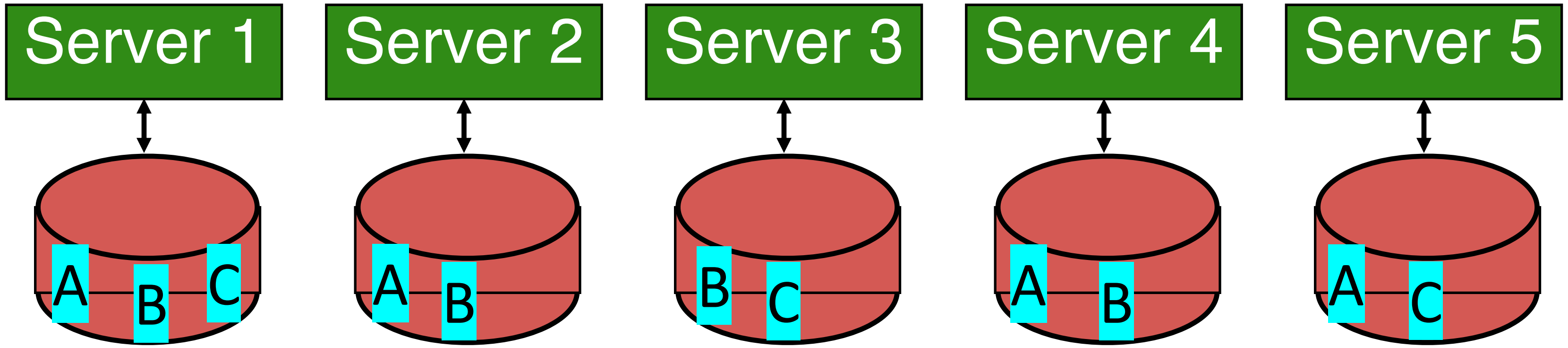
2) Recovery



Machine may come back, or may be dead forever

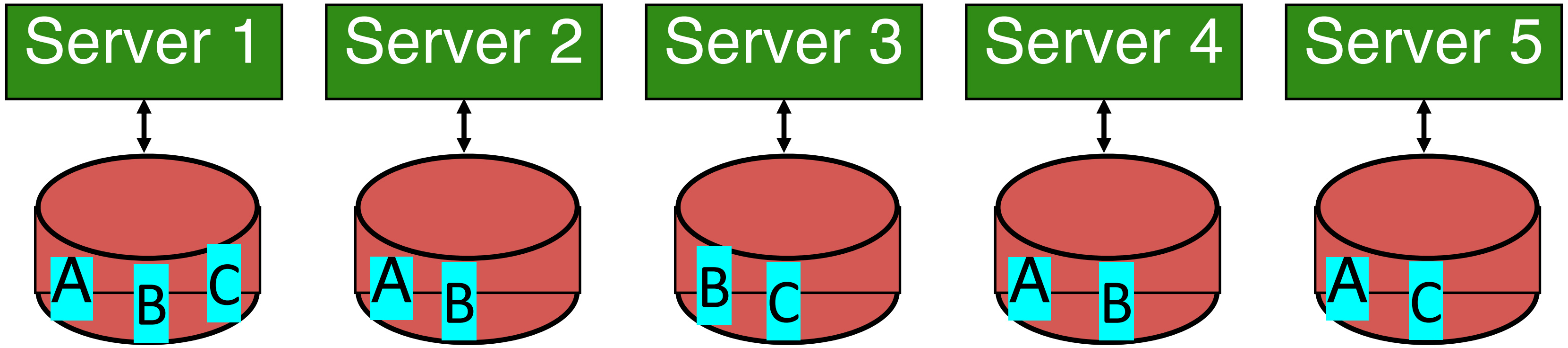
Must identify and replicate lost data on other servers

2) Recovery



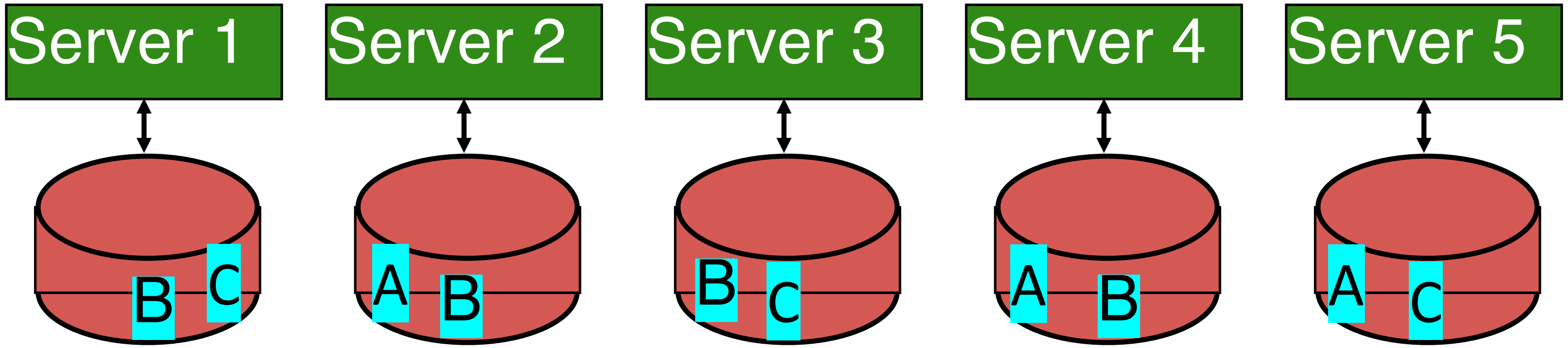
Machine may come back; disk space wasted with extra replicas

2) Recovery



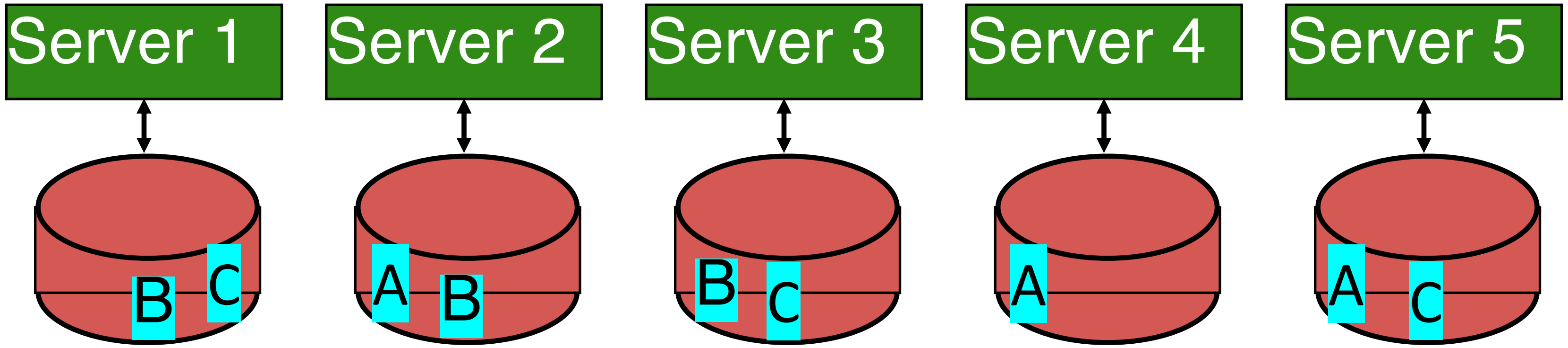
Identify number of replicas and choose to remove extras

2) Recovery



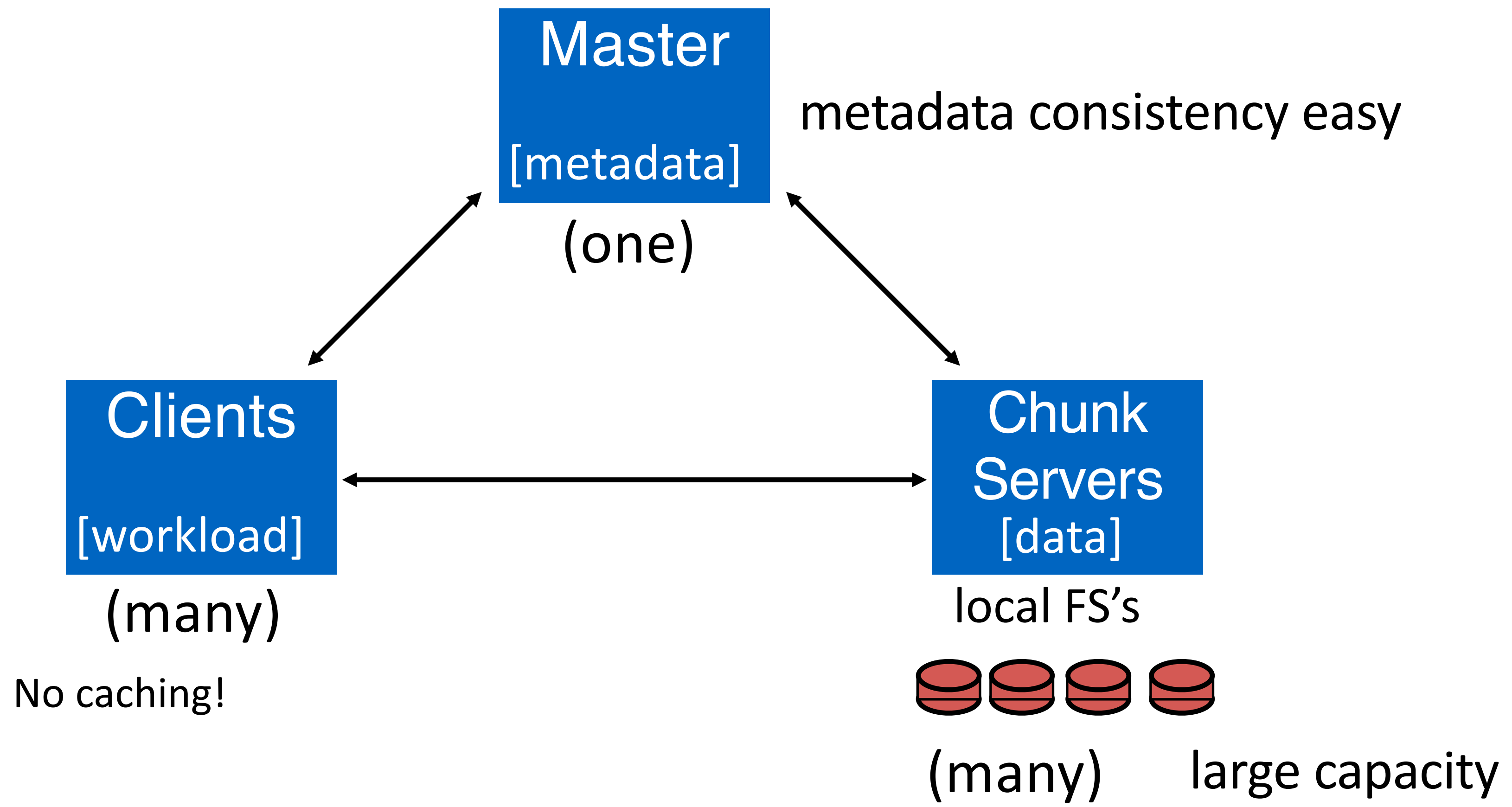
Identify number of replicas and choose to remove extras

Observation

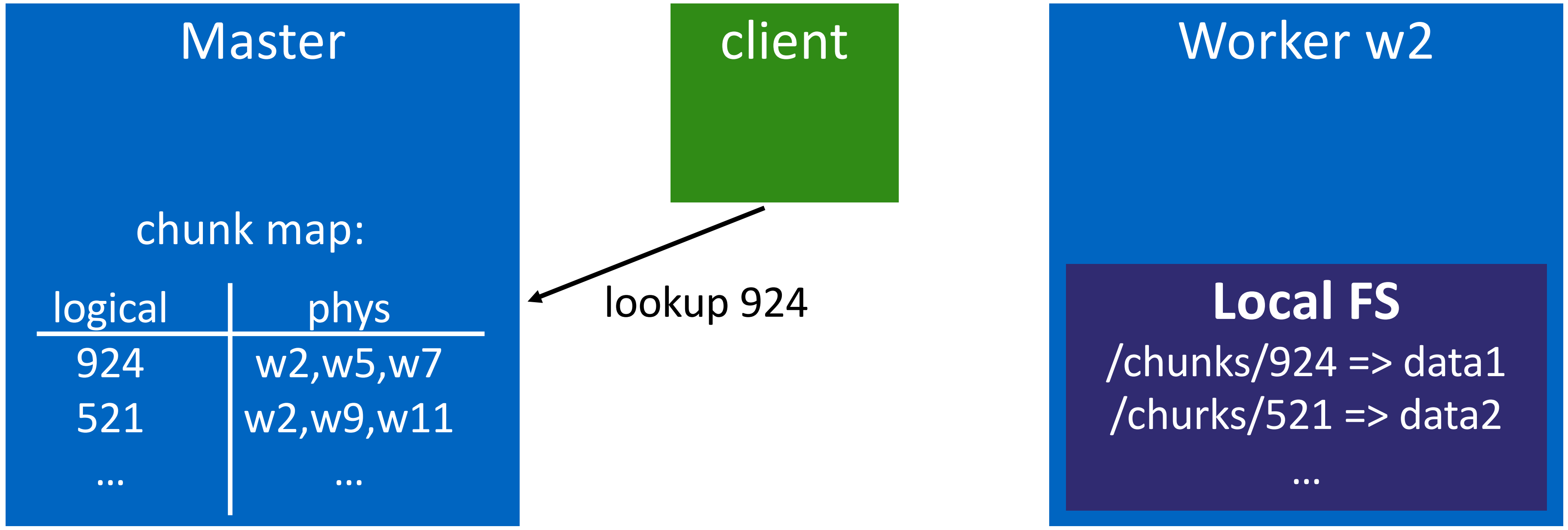


Finding copies of data + maintaining replicas is difficult without **global view** of data

GFS Architecture

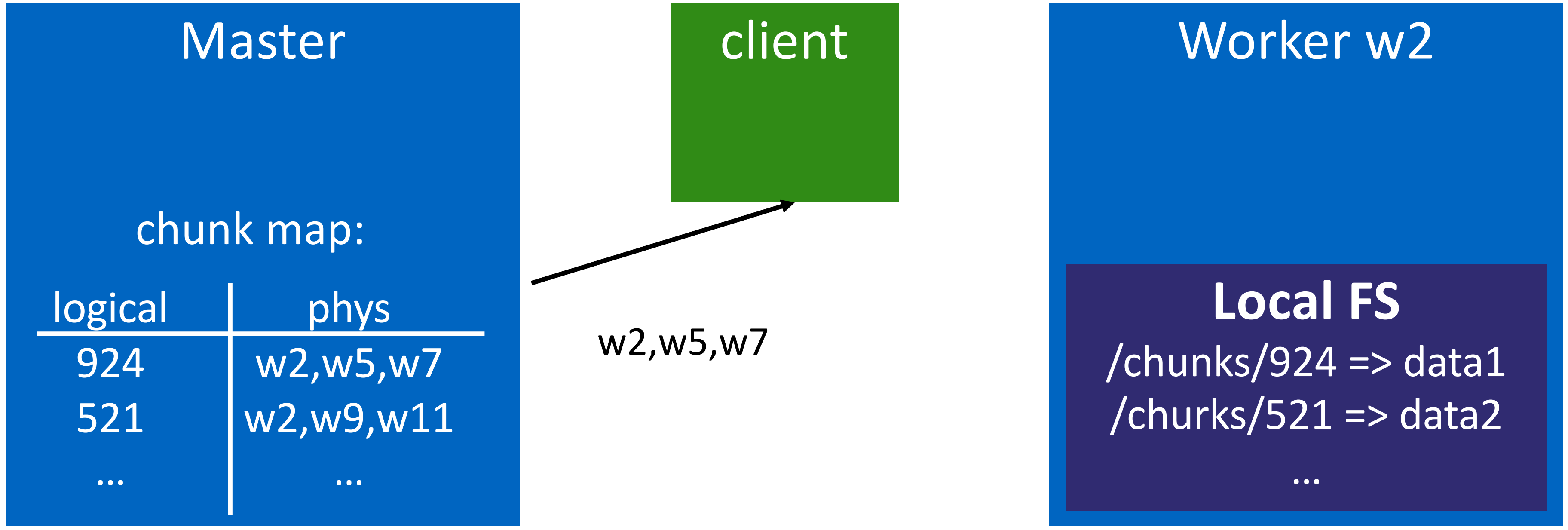


Master Metadata



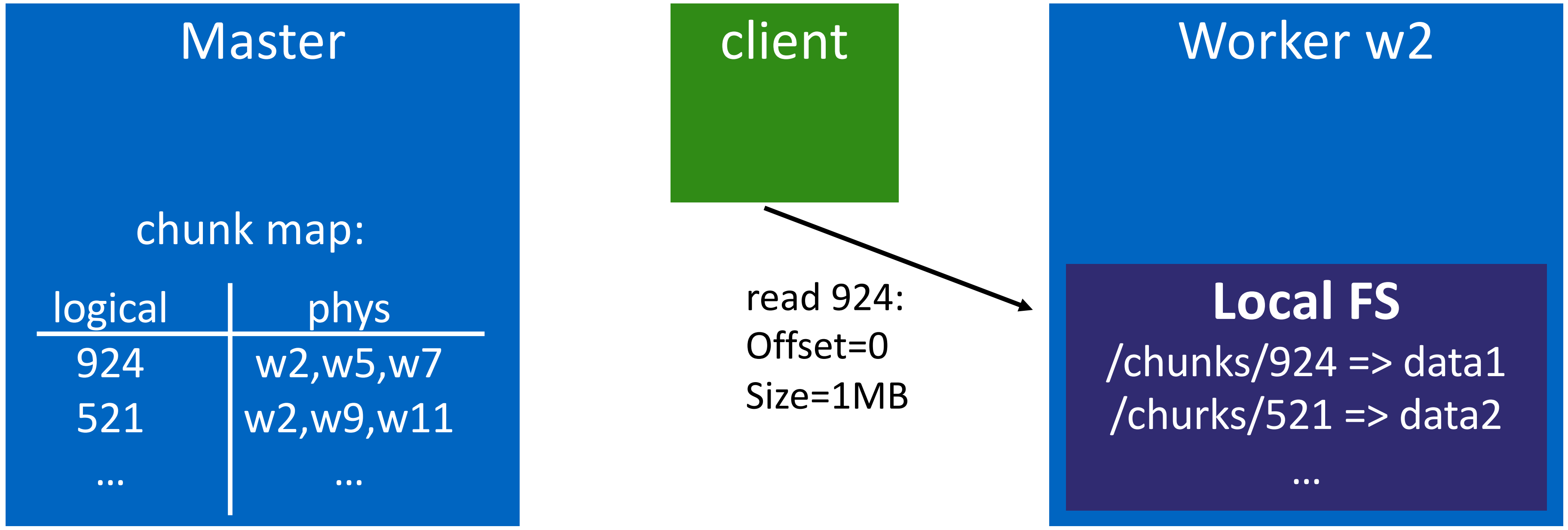
Client wants to read a chunk (identified with unique id num)
How does it find where that chunk lives?

Client reads a chunk

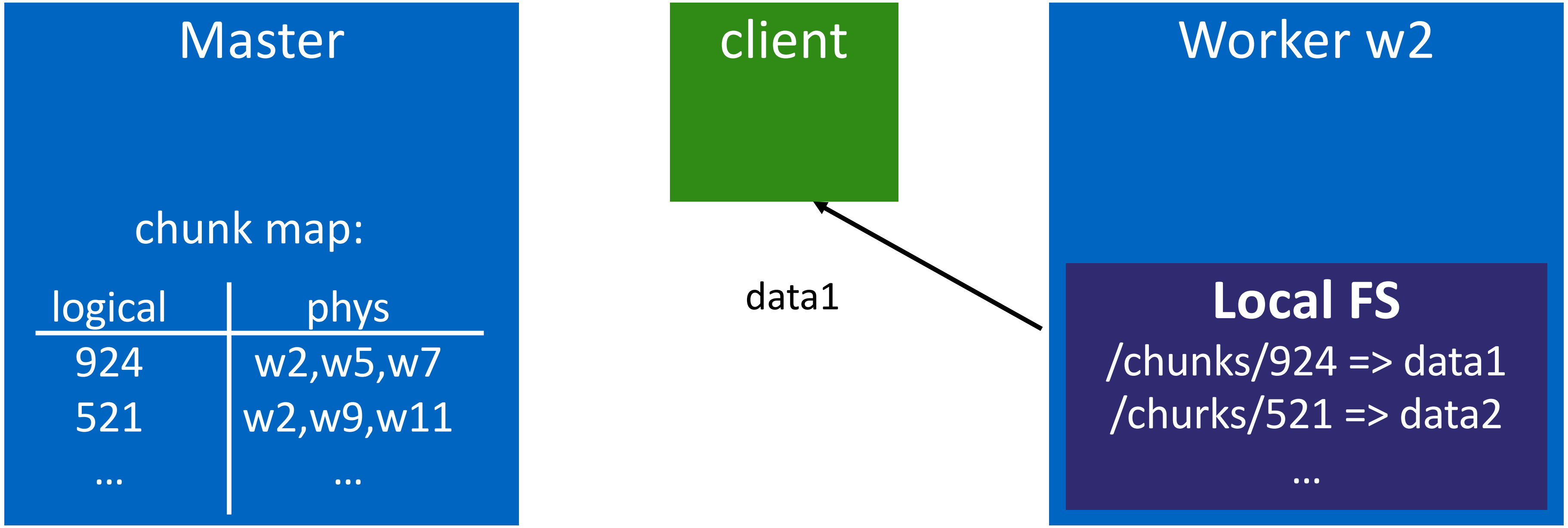


Client can read from any of the listed replicas

Master Metadata



Master Metadata



Master **is not bottleneck** because not involved in most reads.
One master can handle many clients.

What if Master crashes?

Two data structures to worry about

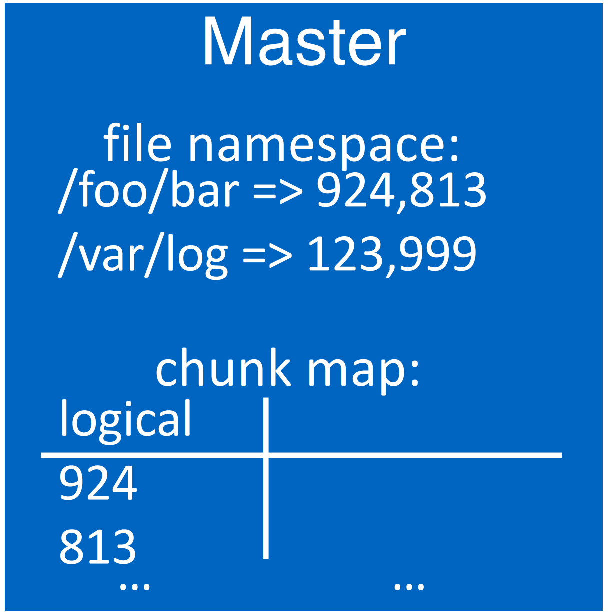
How to make **namespace** persistent?
Write updates to namespace to multiple logs

Where should these logs be located?

- Local disk (disk is never read except for crash)
- Disks on backup masters
- Shadow read-only masters (may lag state, temporary access)

Result: High availability when master crashes!

What about **chunk map**?



Chunk Map Consistency

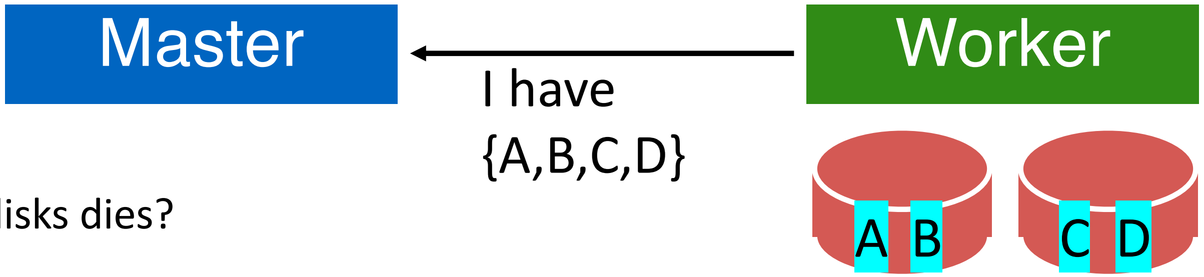
Don't persist chunk map on master

Approach:

After crash (and periodically), master asks each chunkserver which chunks it has

What if chunk server dies too?

Doesn't matter, that worker can't serve chunks anyway



What if one of chunk server's disks dies?

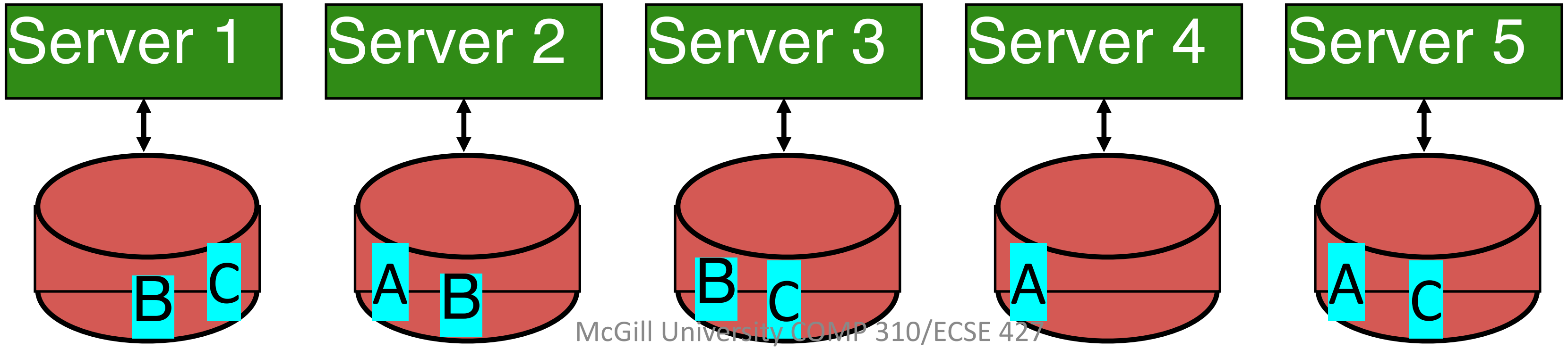
Chunkserver Consistency

How does GFS ensure physical chunks on different chunkservers are consistent with one another?

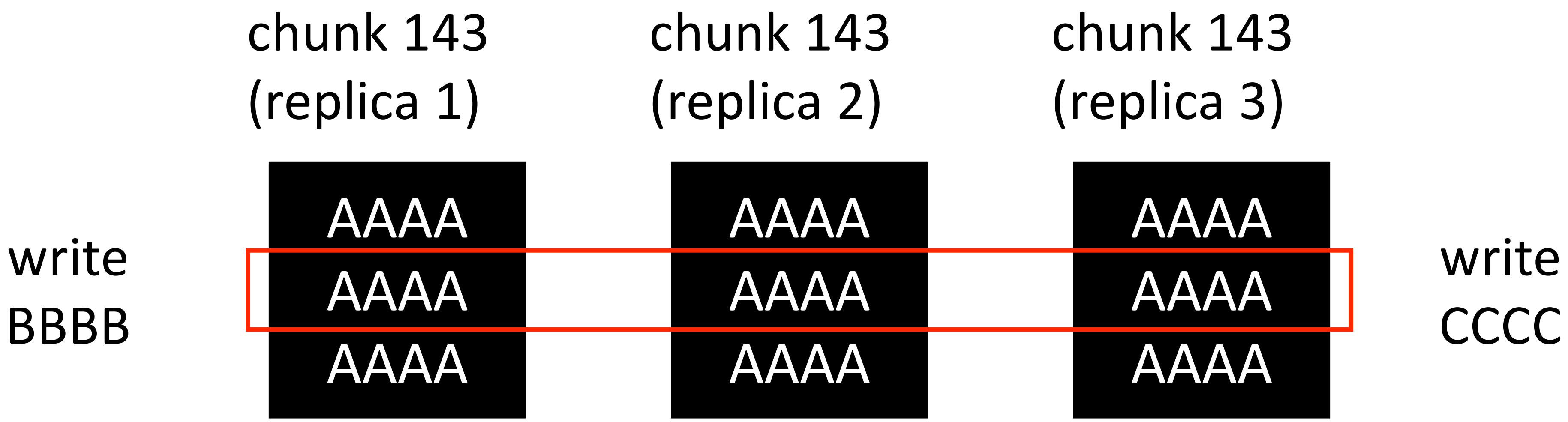
Corruption: delete chunks that violate **checksum**

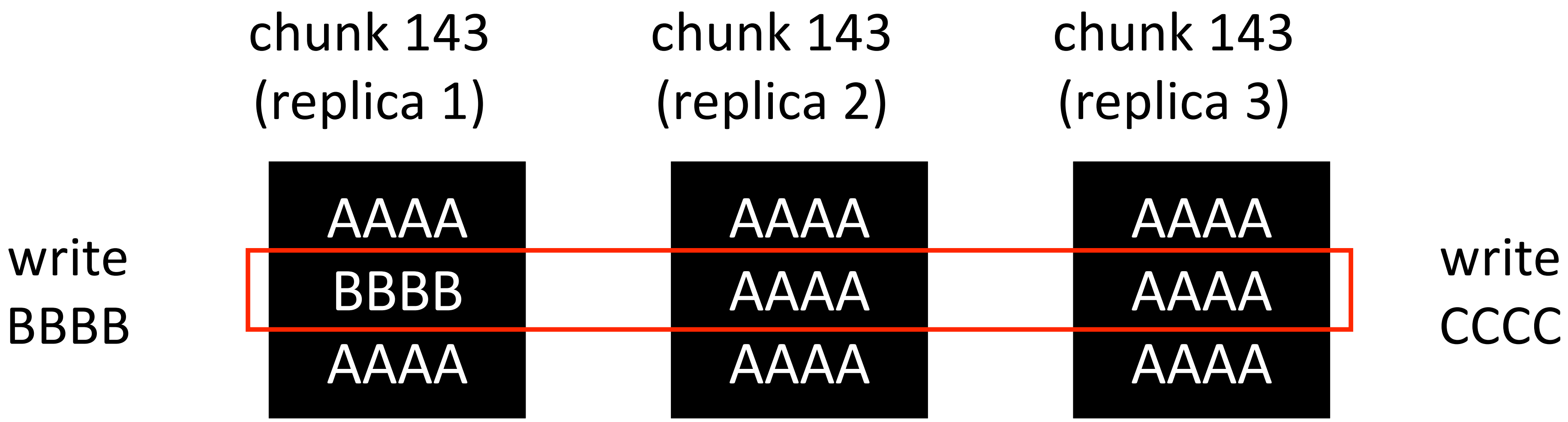
- Master eventually sees chunk has < desired replication

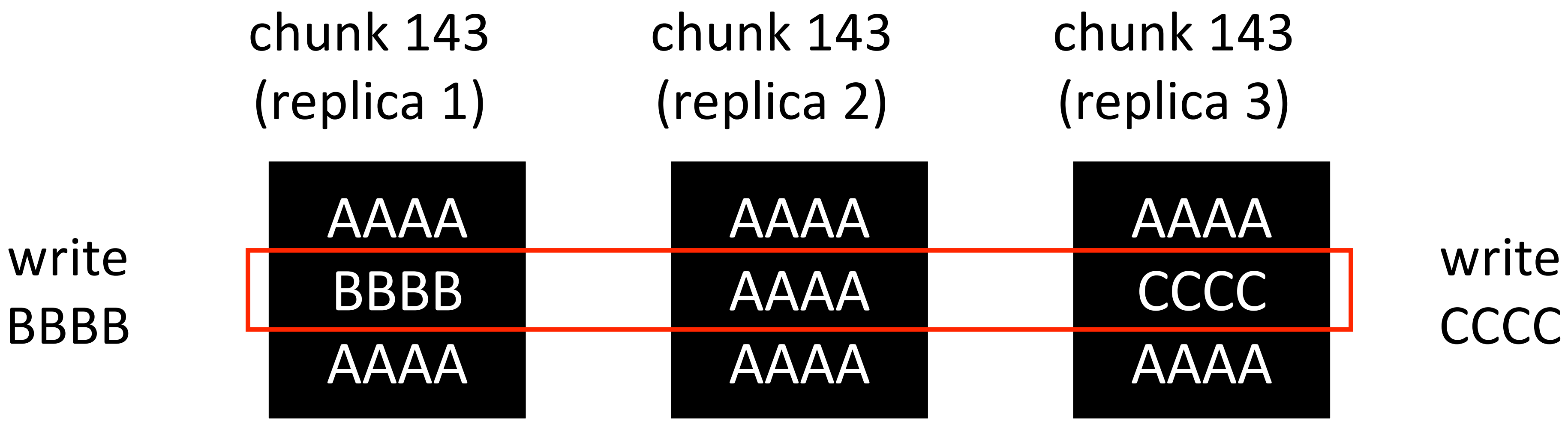
What about concurrent writes (or appends) from different clients? (e.g., multiple producers)

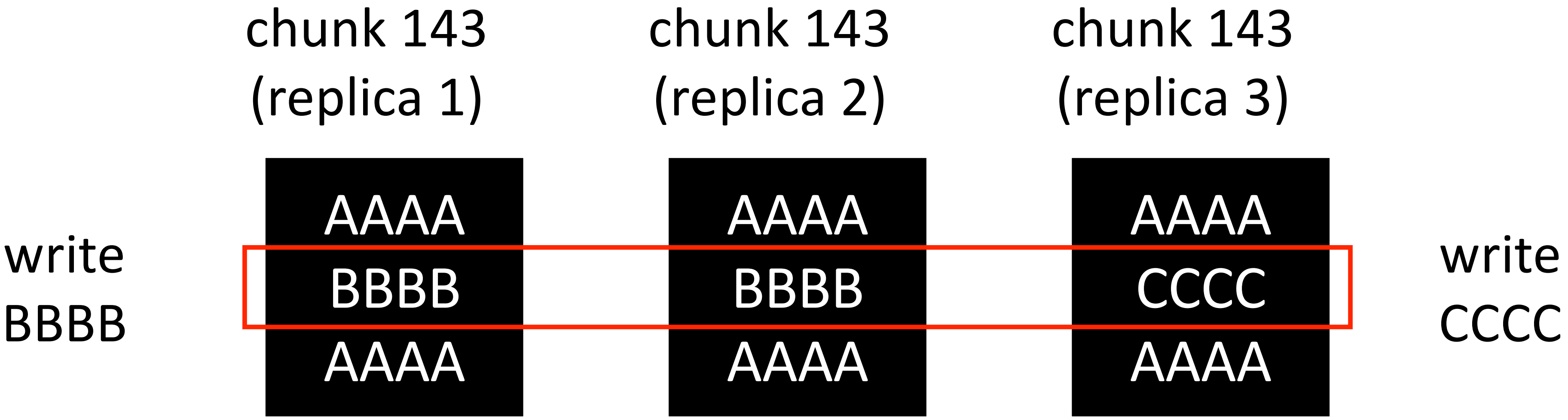


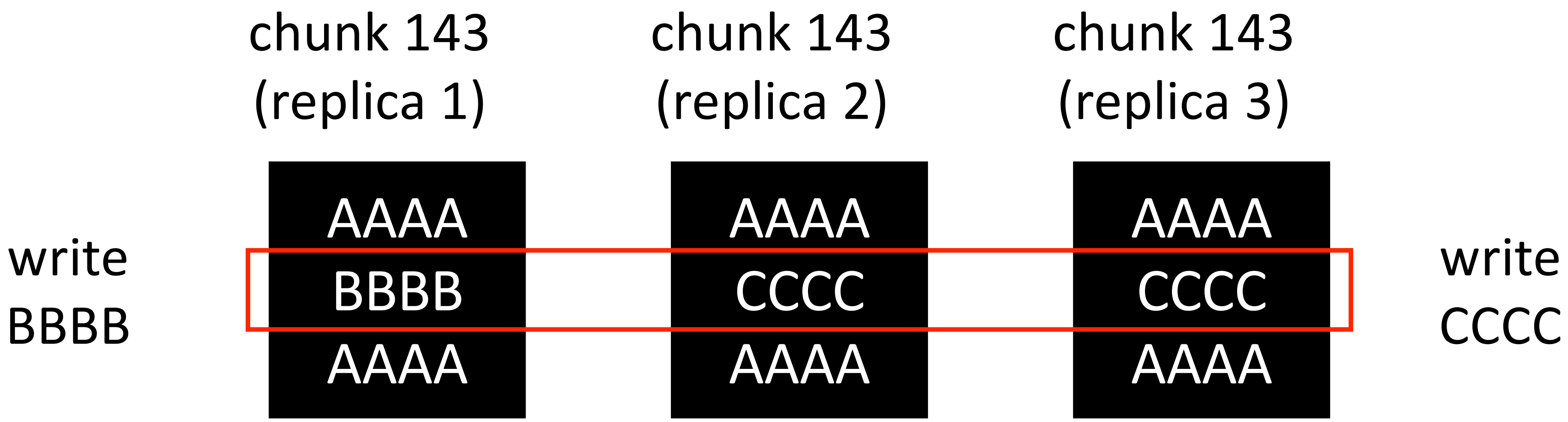
chunk 143 (replica 1)	chunk 143 (replica 2)	chunk 143 (replica 3)
AAAA AAAA AAAA	AAAA AAAA AAAA	AAAA AAAA AAAA

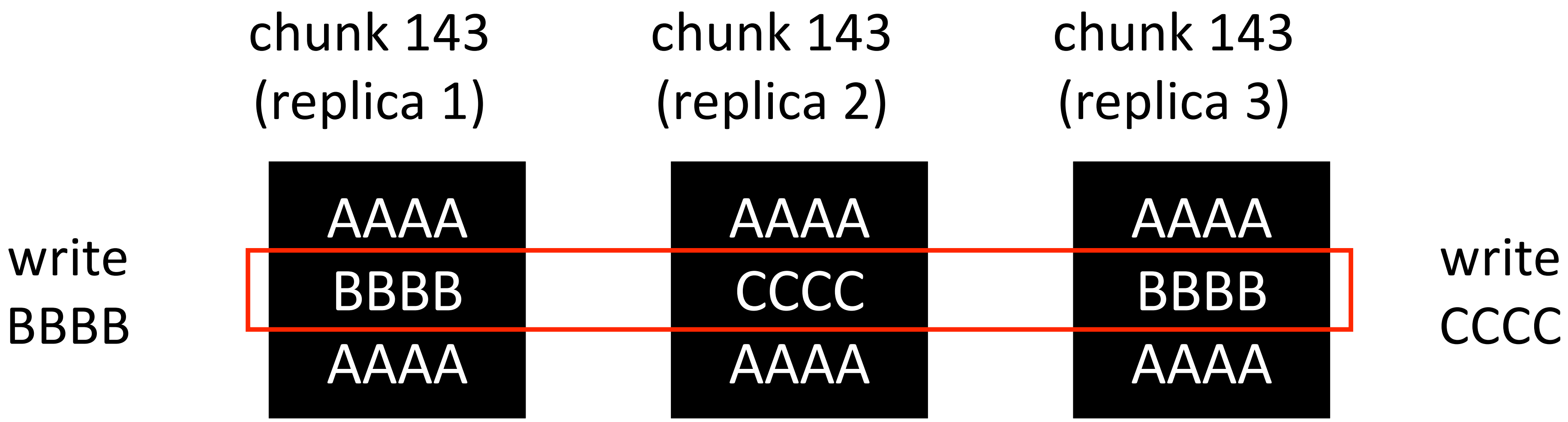

chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
write
BBBB
write
CCCC


chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
AAAA
BBBB
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
AAAA
write
BBBB
write
CCCC

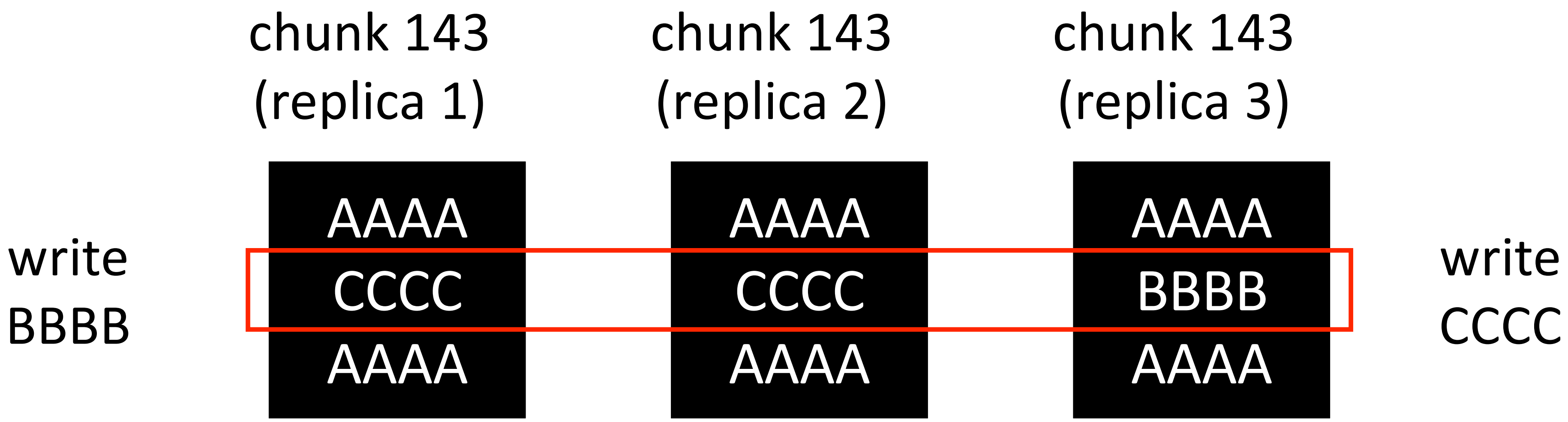

chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
write
BBBB
AAAA
BBBB
AAAA
AAAA
AAAA
AAAA
AAAA
CCCC
AAAA
write
CCCC


chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
write
BBBB
AAAA
BBBB
AAAA
AAAA
BBBB
AAAA
AAAA
CCCC
AAAA
write
CCCC


write
BBBB
chunk 143
(replica 1)
AAAA
BBBB
AAAA
chunk 143
(replica 2)
AAAA
CCCC
AAAA
chunk 143
(replica 3)
AAAA
CCCC
AAAA
write
CCCC



chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
write
BBBB
AAAA
BBBB
AAAA
AAAA
CCCC
AAAA
AAAA
BBBB
AAAA
write
CCCC


chunk 143
(replica 1)
chunk 143
(replica 2)
chunk 143
(replica 3)
write
BBBB
AAAA
CCCC
AAAA
AAAA
CCCC
AAAA
AAAA
BBBB
AAAA
write
CCCC

chunk 143 (replica 1)

AAAA
CCCC
AAAA

chunk 143 (replica 2)

AAAA
CCCC
AAAA

chunk 143 (replica 3)

AAAA
BBBB
AAAA

chunk 143 (replica 1)	chunk 143 (replica 2)	chunk 143 (replica 3)
AAAA	AAAA	AAAA
CCCC	CCCC	BBBB
AAAA	AAAA	AAAA

Chunks disagree,
but all checksums are correct,
all writes suceeeded,
and no machines ever failed!!

Chunkserver Consistency

GFS must “serialize” writes across chunkservers

- Decide an order of writes and ensure order is followed by every chunkserver

How to decide on an order?

- don’t want to overload master
- let one replica be primary and decide order of writes from clients

Primary Replica

- Master chooses primary replica for each logical chunk

- What if primary dies?

- Give primary replica a **lease** that expires after some time (1 minute)
  - If master wants to reassign primary, and it can't reach old primary, just wait 1 minute

GFS Summary

- Fight failure with replication
- Metadata consistency is hard, centralize to make it easier
- Data consistency is easier, distribute it for scalability

Thank you to TA team!!

- Alice Chang
- Zhongjie Wu
- Shakiba Bolbolian Khah
- Clare Jang
- Sebastian Rolon
- Jiaxuan Chen
- Aayush Kapur
- Murray Kornelsen
- Mohammad Rahman
- Emmanuel Wilson

- *Check out my lab's work* *http://discslab.cs.mcgill.ca/*

- *Reach out to me for research projects!*

- *Consider taking my course* ***COMP-513 Advanced Computer Systems*** *Offered in Fall 2023*

Further Reading

Operating Systems: Three Easy Pieces by R. & A. Arpaci-Dusseau

Chapters 49.

https://pages.cs.wisc.edu/~remzi/OSTEP/

Credits:

Some slides adapted from the OS courses of Profs. Remzi and Andrea Arpaci-Dusseau (University of Wisconsin-Madison), Prof. Willy Zwaenepoel (University of Sydney), and Prof. Youjip Won (Hanyang University).