[537] NFS

Chapters 47
Tyler Harter
11/19/14

File-System Case Studies

Local
- **FFS**: Fast File System
- **LFS**: Log-Structured File System

Network
- **NFS**: Network File System
- **AFS**: Andrew File System

File-System Case Studies

Local
- **FFS**: Fast File System
- **LFS**: Log-Structured File System

Network
- **NFS**: Network File System [today]
- **AFS**: Andrew File System

Communication Review

Challenges

Communication implies multiple components.

Need to deal with partial failure.

No global view!
- are router buffers full?
- why didn’t ACK come back?

Communication Abstractions

Raw messages

Reliable messages

PL: RPC call

OS: global FS

Raw Messages: UDP

Read/write over sockets.

IP addr + port identify endpoints.

Messages are unreliable.

Communication Abstractions

Raw messages

Reliable messages

PL: RPC call

OS: global FS

Reliable Messages: TCP

Timeout/retry.

Retry

Receiver sends ACK message back.

Upon no ACK, sender retries.

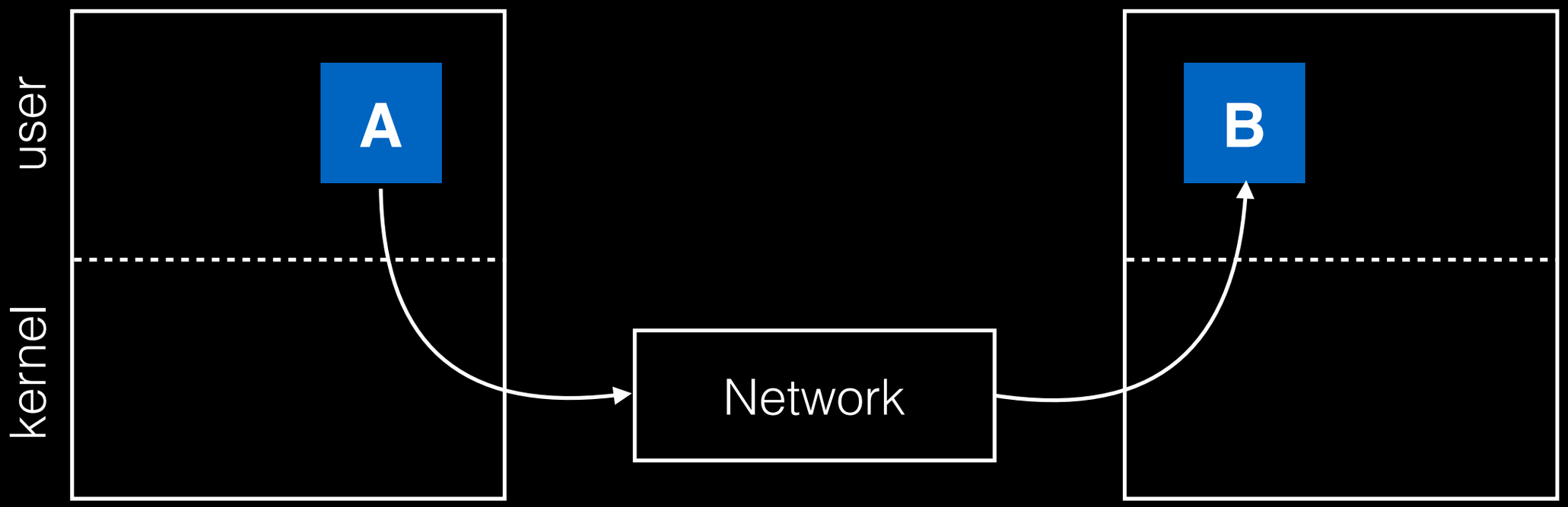
How long to wait? Use adaptive approach.

Reliable Messages: TCP

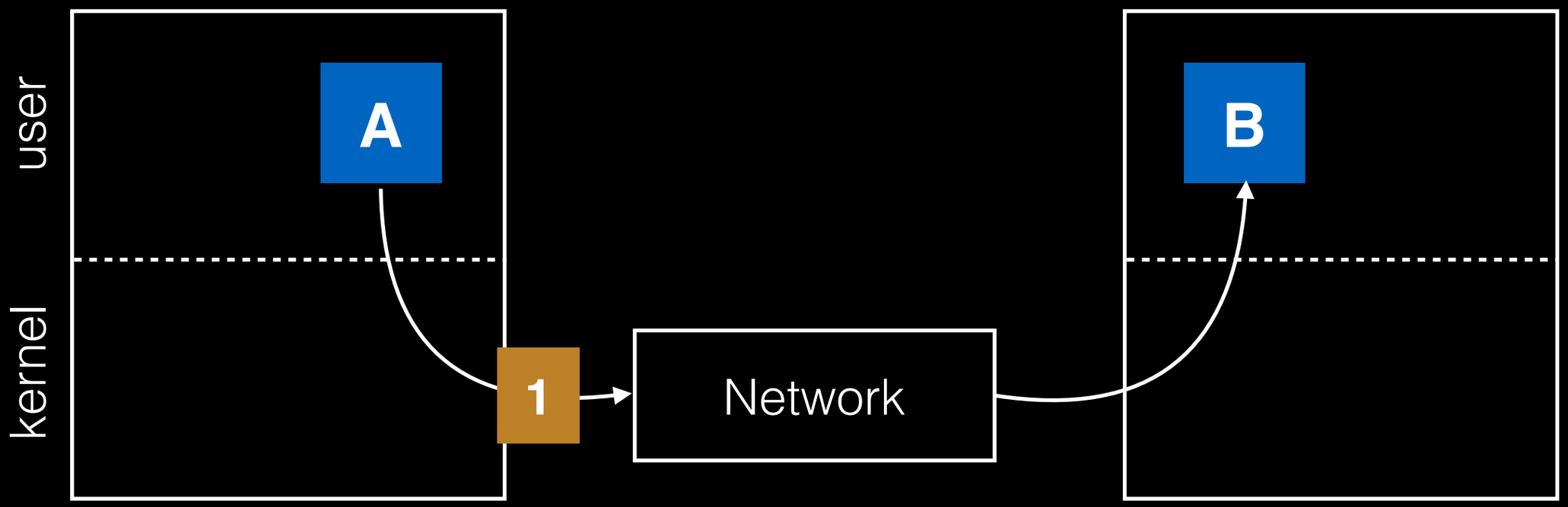
Timeout/retry.

Buffer messages to preserve ordering.

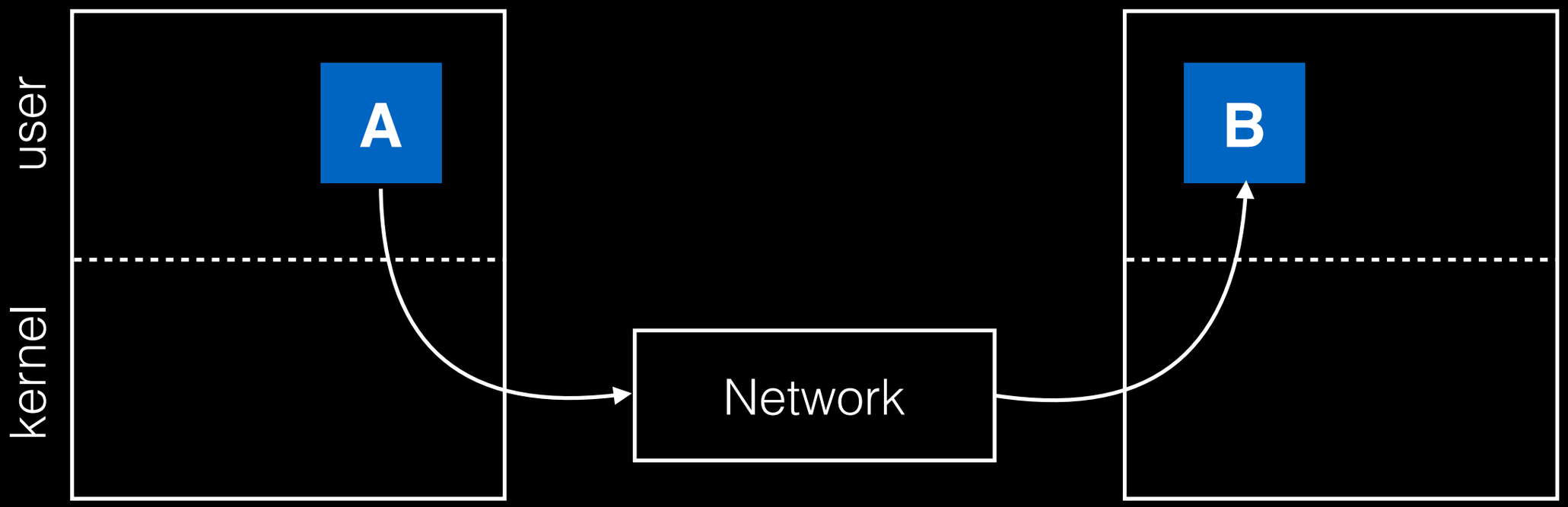
Buffering



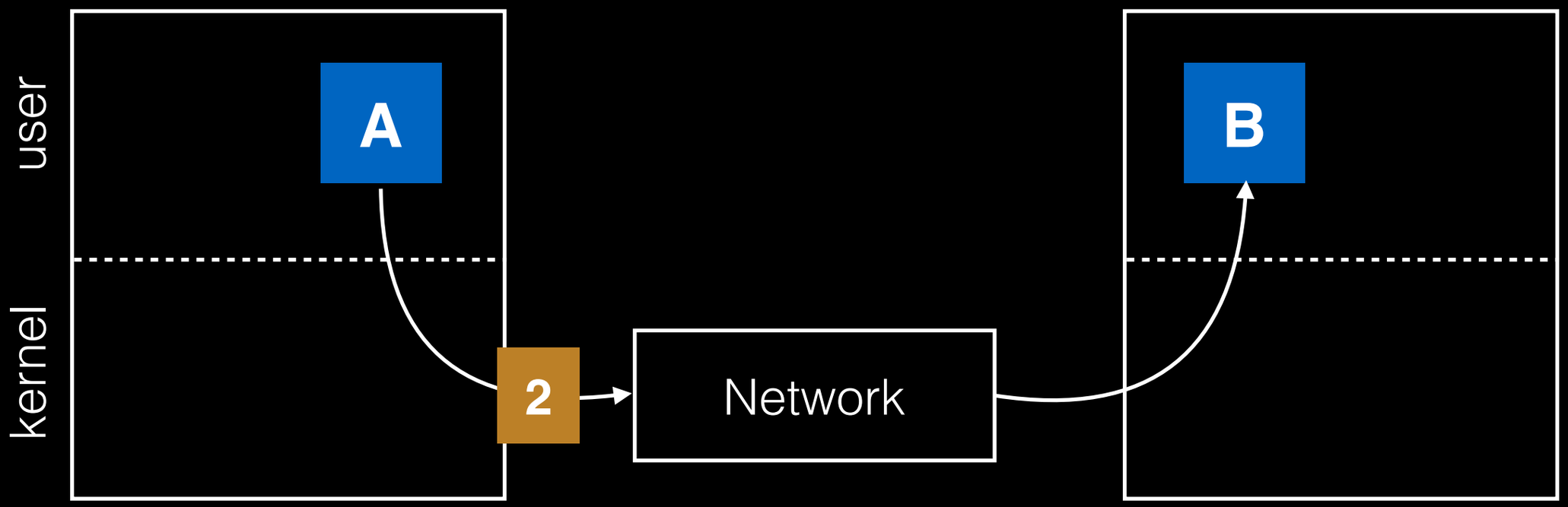
Buffering



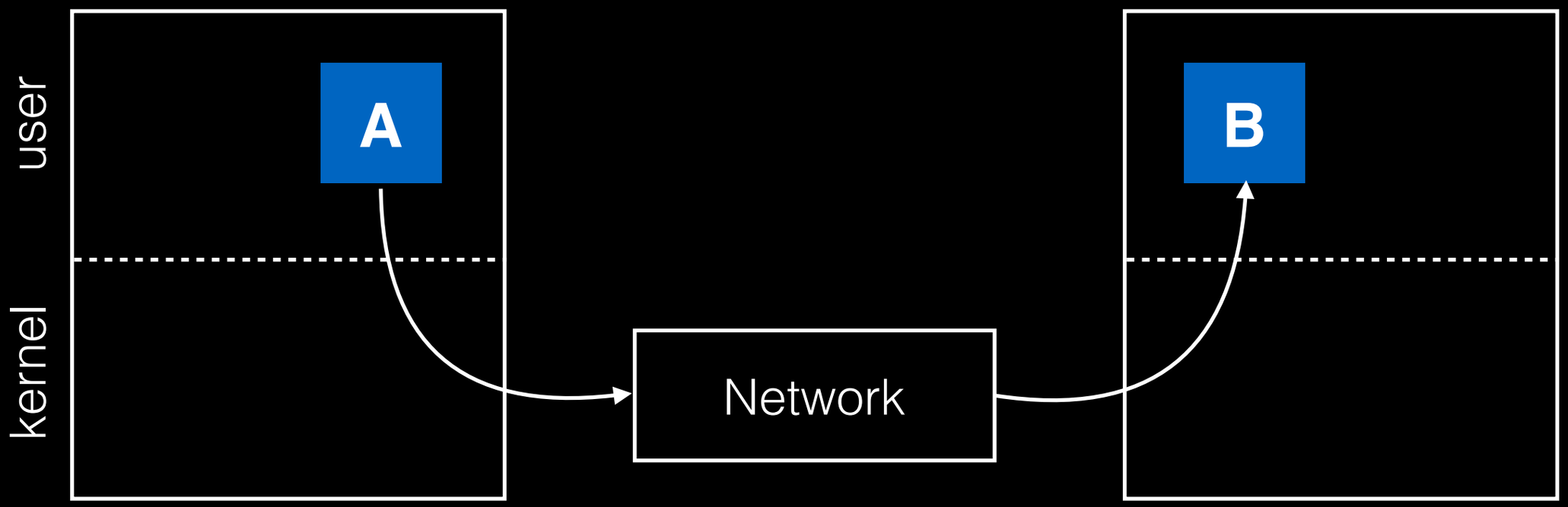
Buffering



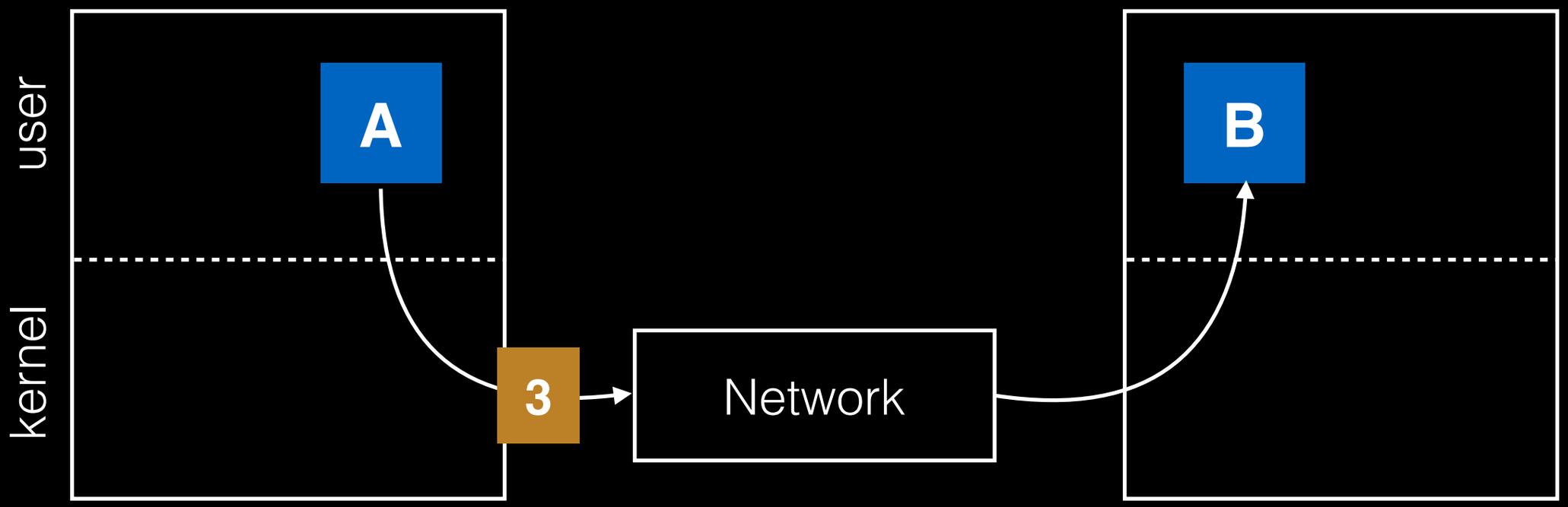
Buffering



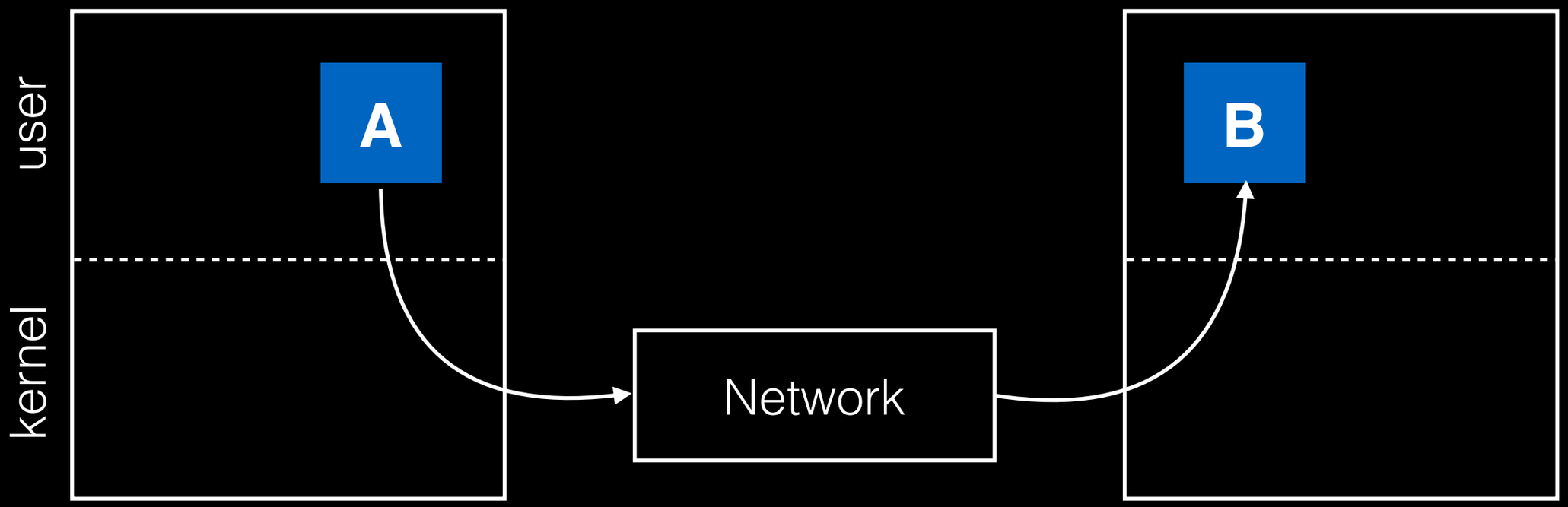
Buffering



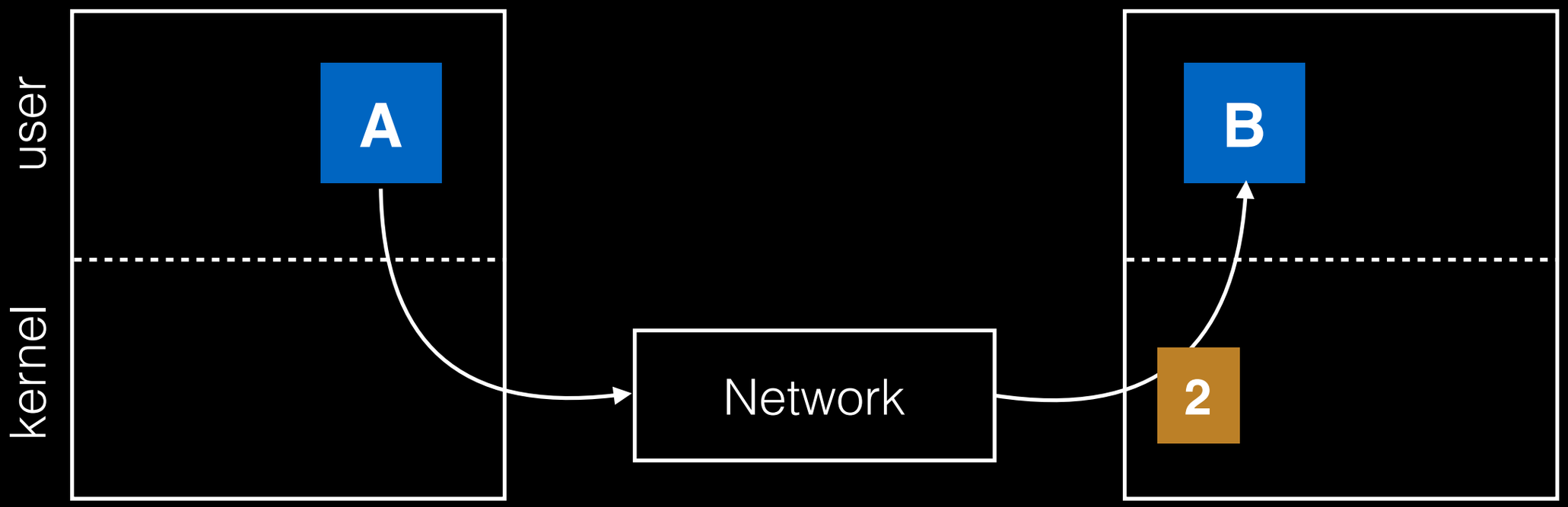
Buffering



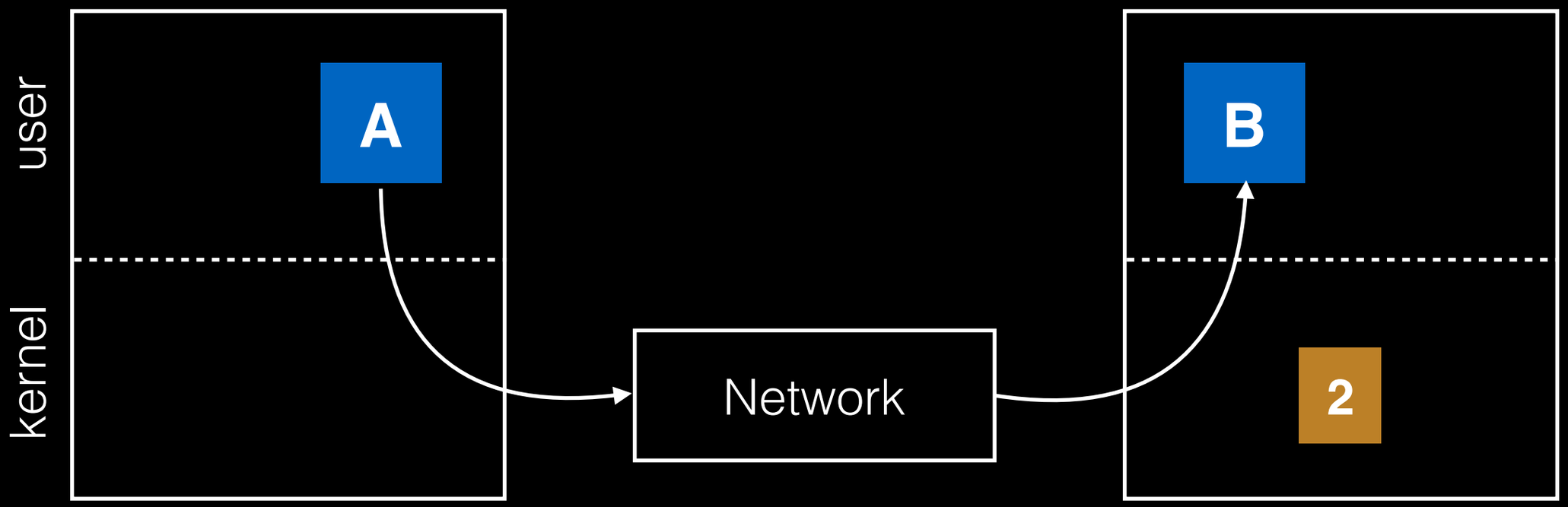
Buffering



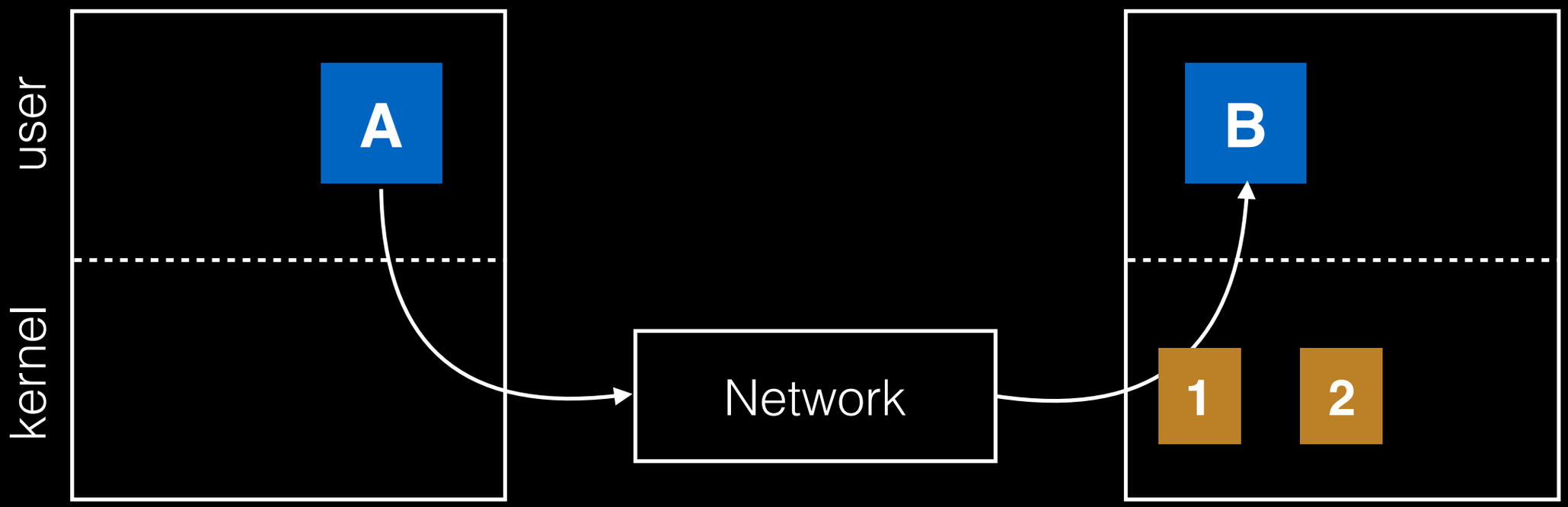
Buffering



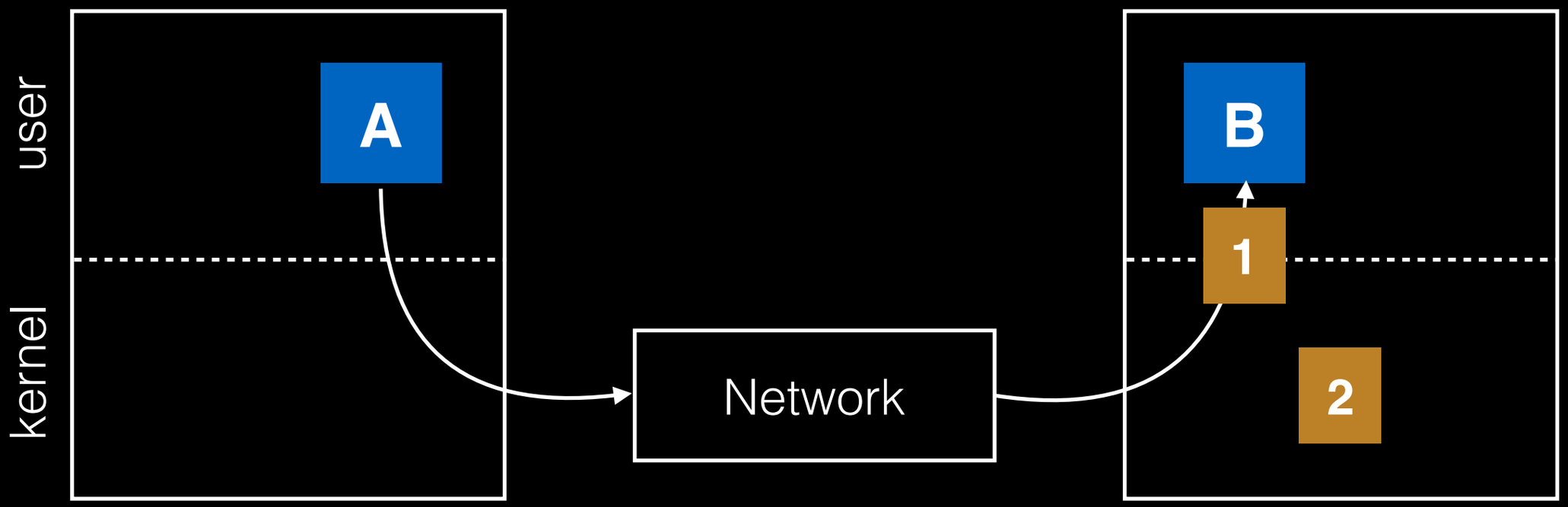
Buffering



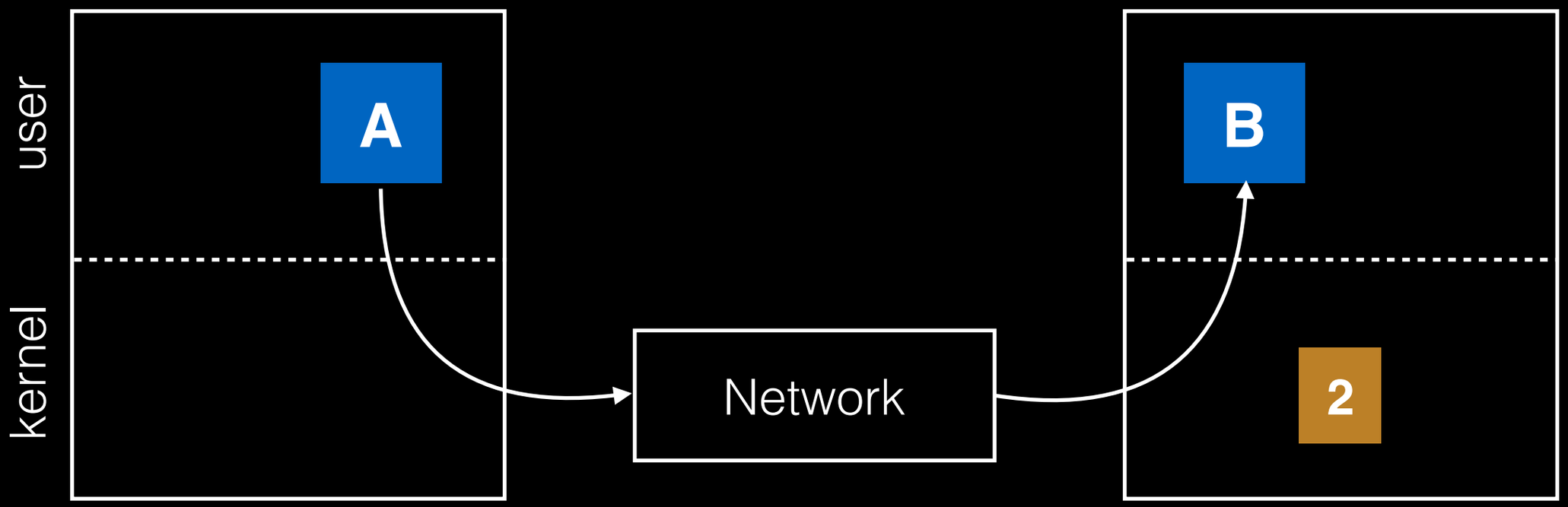
Buffering



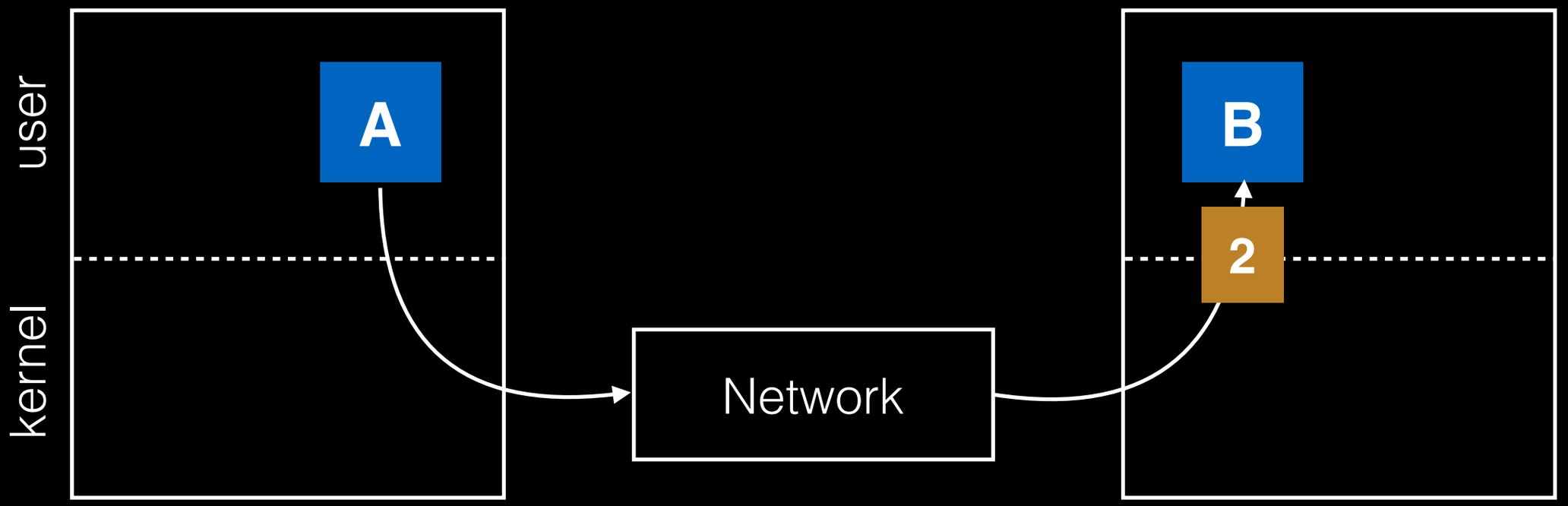
Buffering



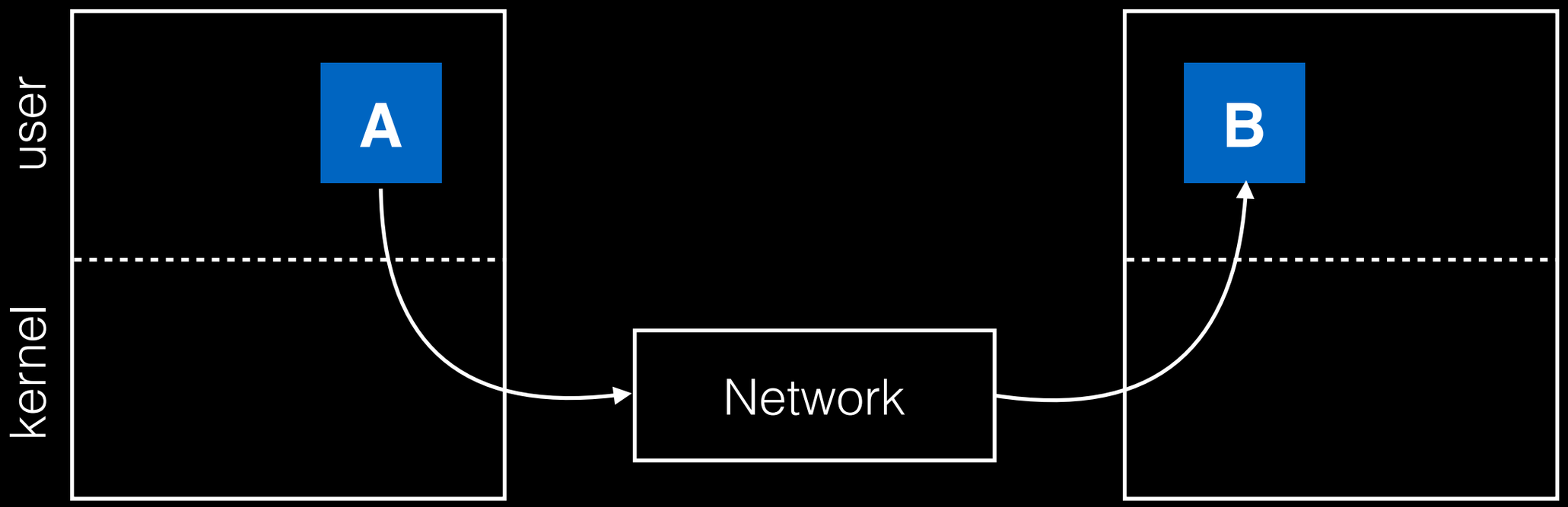
Buffering



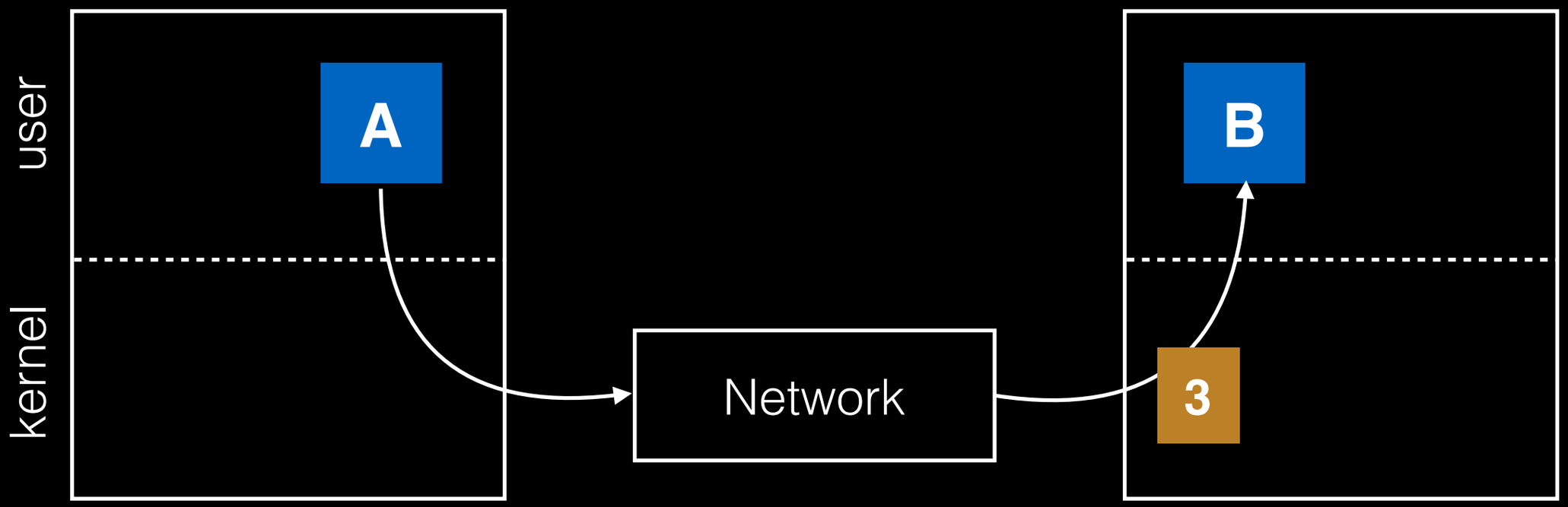
Buffering



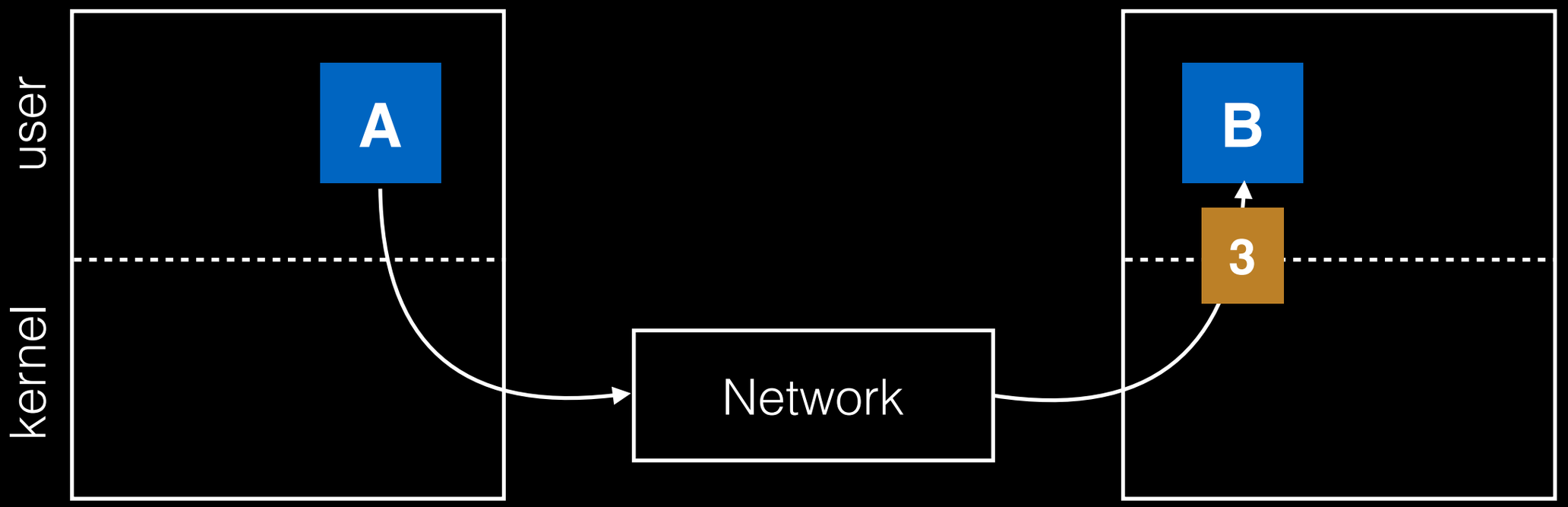
Buffering



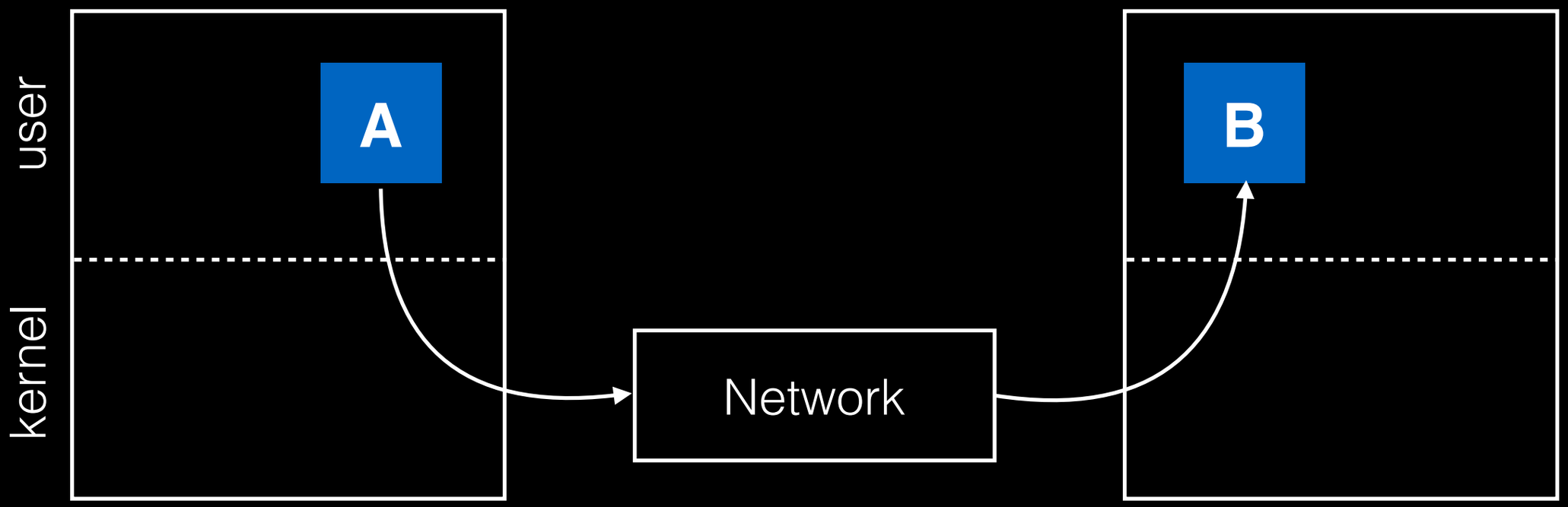
Buffering



Buffering



Buffering



Reliable Messages: TCP

Timeout/retry.

Buffer messages to preserve ordering.

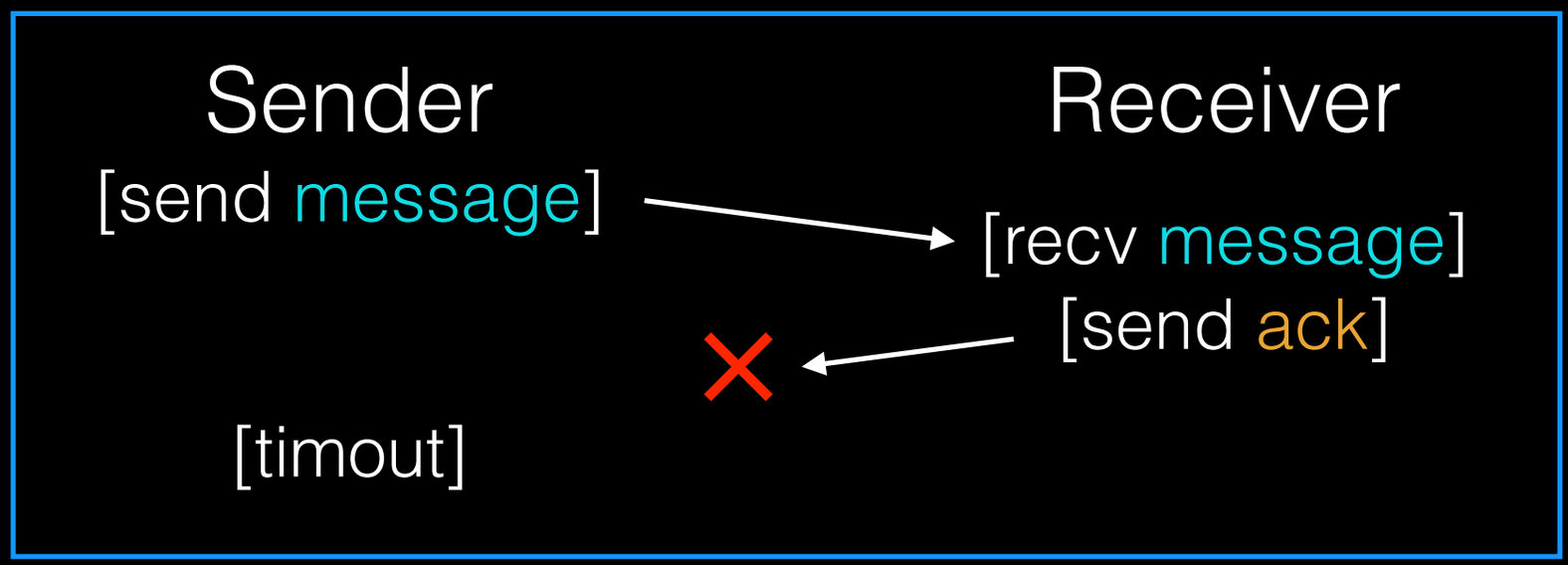
Suppress duplicates.

Case 1

How can sender tell difference?

Retries may cause duplicates.

Case 2



Suppressing Duplicates

TCP gives each message a seq num.

TCP remember all messages before N have been received.

Suppose message K is received. Suppress if:
- K < N
- Msg K is already buffered

Communication Abstractions

Raw messages

Reliable messages

PL: RPC call

OS: global FS

Remote Procedure Calls

Strategy: create wrappers so calling a function on another machine feels just like calling a local function.

RPC

Machine A

```
int main(…) {

}
```

Machine B

```
int foo(char *msg) {
        …
}
```

RPC

Machine A

```
int main(…) {
    int x = foo();
}
```

Machine B

```
int foo(char *msg) {
    …
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(…) {
    int x = foo();
}
```

Machine B

```
int foo(char *msg) {
    …
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```
int foo(char *msg) {
    …
}
```

Want main() on A to call foo() on B.

RPC

Machine A

```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```
int foo(char *msg) {
    …
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

Want main() on A to call foo() on B.

RPC

Machine A

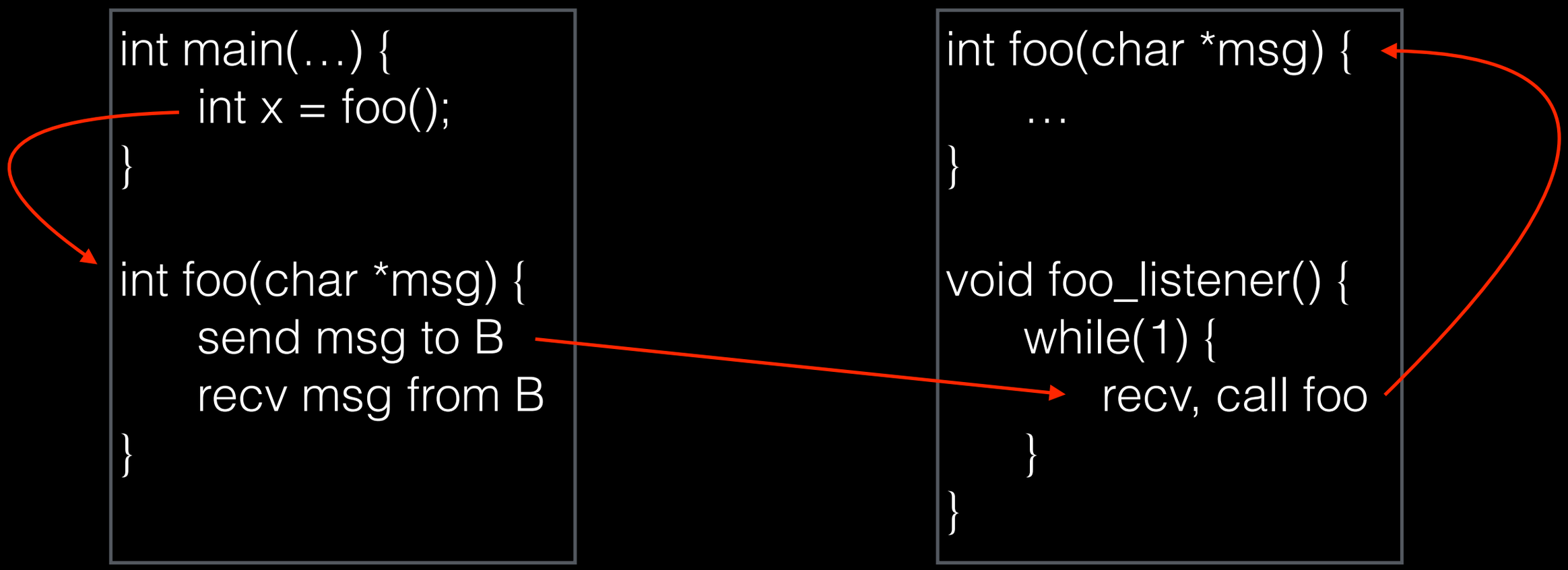
```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

Machine B

```
int foo(char *msg) {
    …
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```



Actual calls.

RPC

Machine A

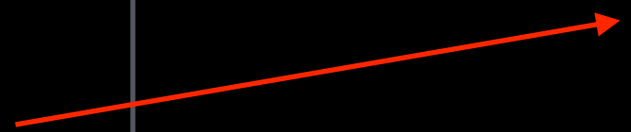
```
int main(…) {
      int x = foo();
}

int foo(char *msg) {
      send msg to B
      recv msg from B
}
```

Machine B

```
int foo(char *msg) {
      …
}

void foo_listener() {
      while(1) {
            recv, call foo
      }
}
```



What it feels like for programmer.

RPC

Machine A

```
int main(…) {
    int x = foo();
}

int foo(char *msg) {
    send msg to B
    recv msg from B
}
```

client wrapper

Machine B

```
int foo(char *msg) {
    …
}

void foo_listener() {
    while(1) {
        recv, call foo
    }
}
```

server wrapper

Wrappers.

Tricky Issues

Pointers.

Build over TCP or UDP?

Communication Abstractions

Raw messages

Reliable messages

PL: RPC call

OS: global FS

Network File System

Primary Goal

Local FS: processes on same machine access shared files.

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

Subgoals

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

Overview

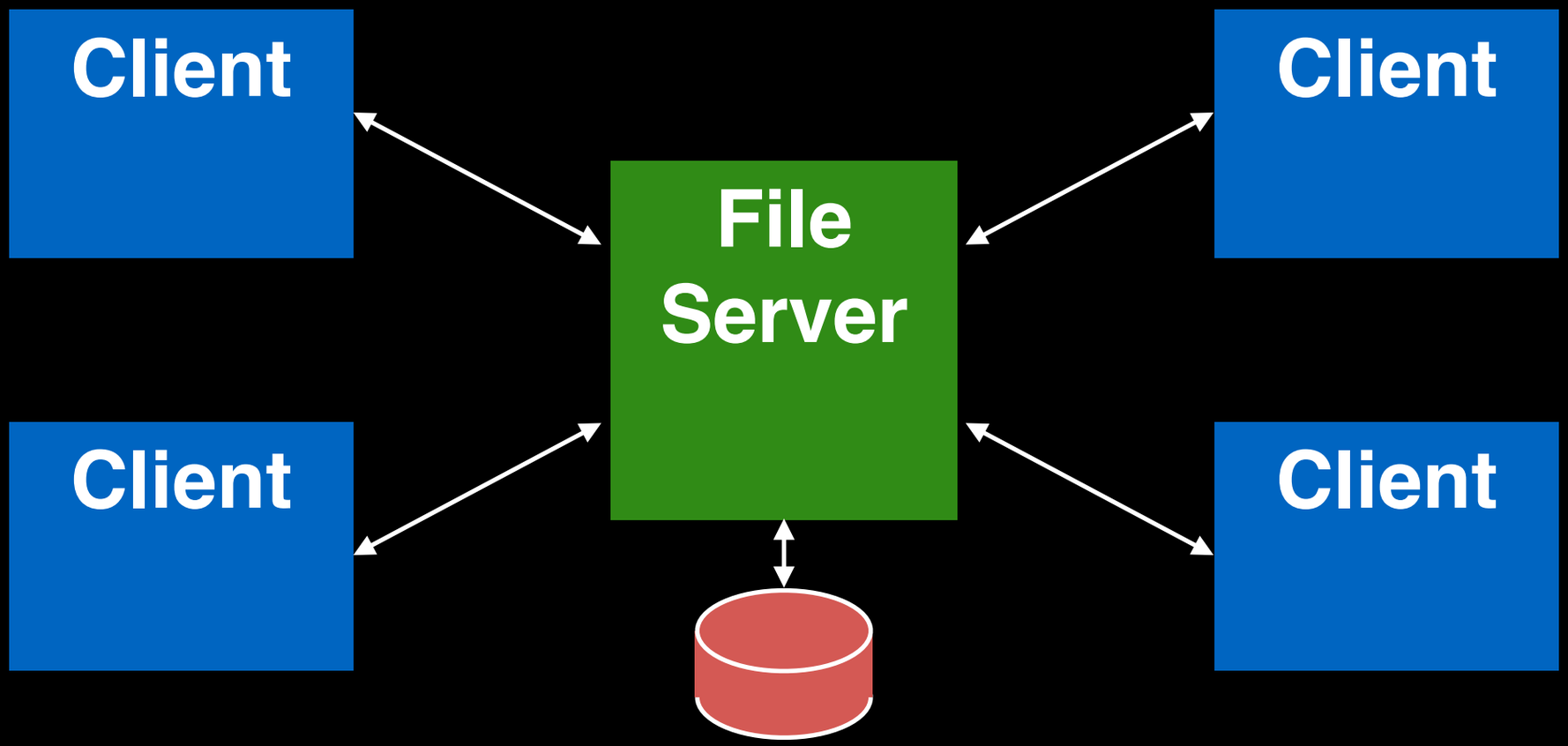
Architecture

Network API

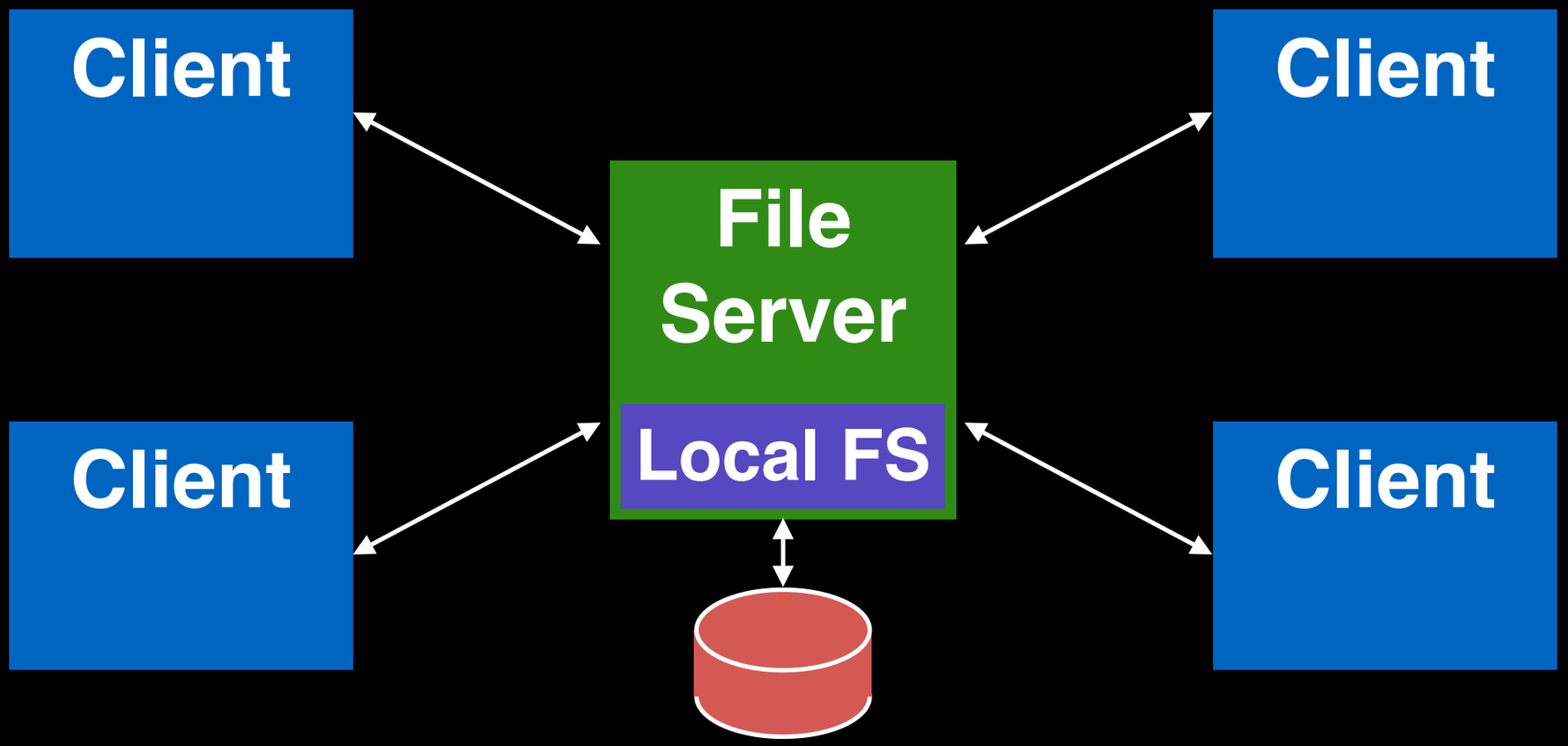
Write Buffering

Cache

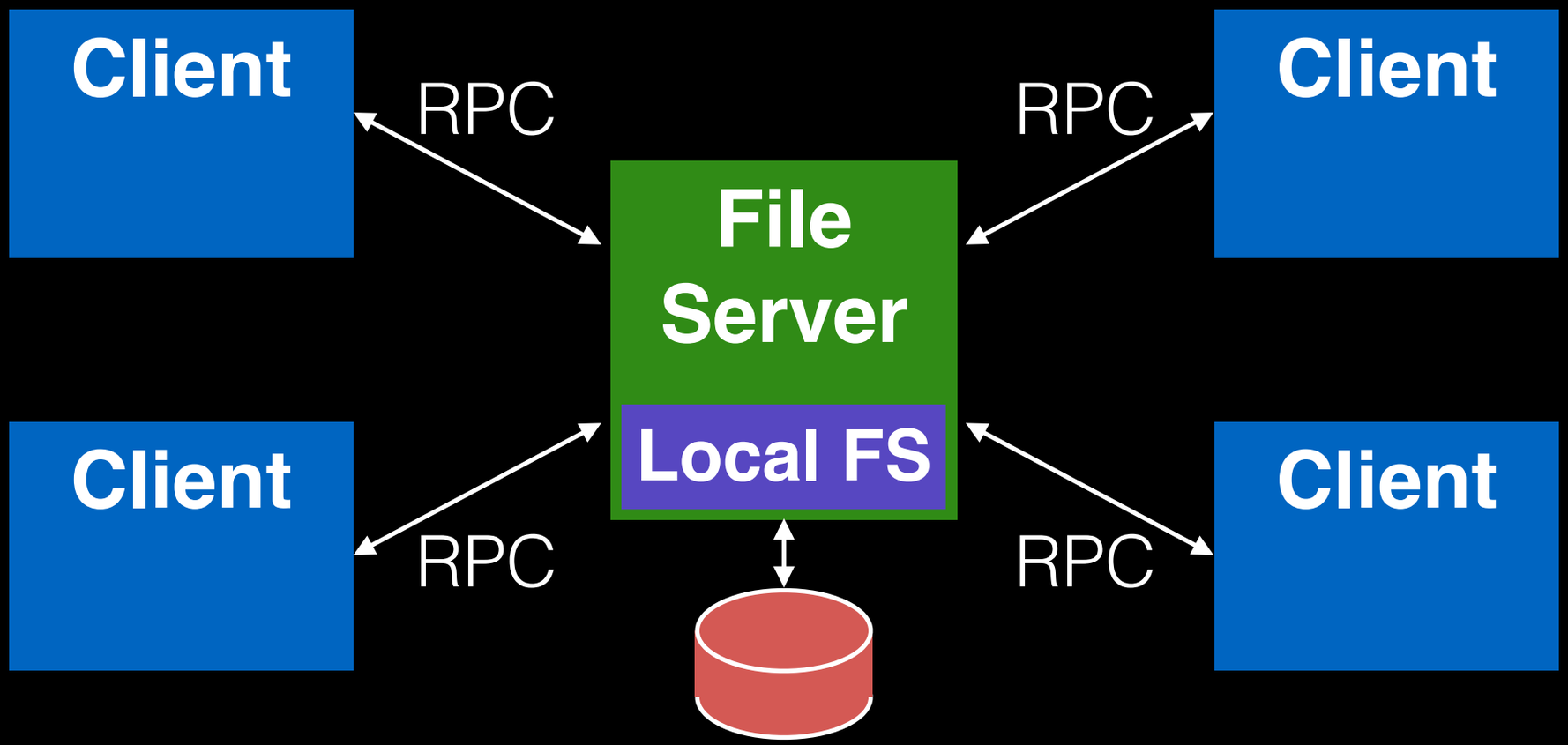
NFS Architecture



Building Block 1: Local FS



Building Block 2: RPC



Main Design Decisions

What functions to expose via RPC?

Think of NFS as more of a protocol than a particular file system.

Many companies have implemented NFS:

Oracle/Sun, NetApp, EMC, IBM, etc

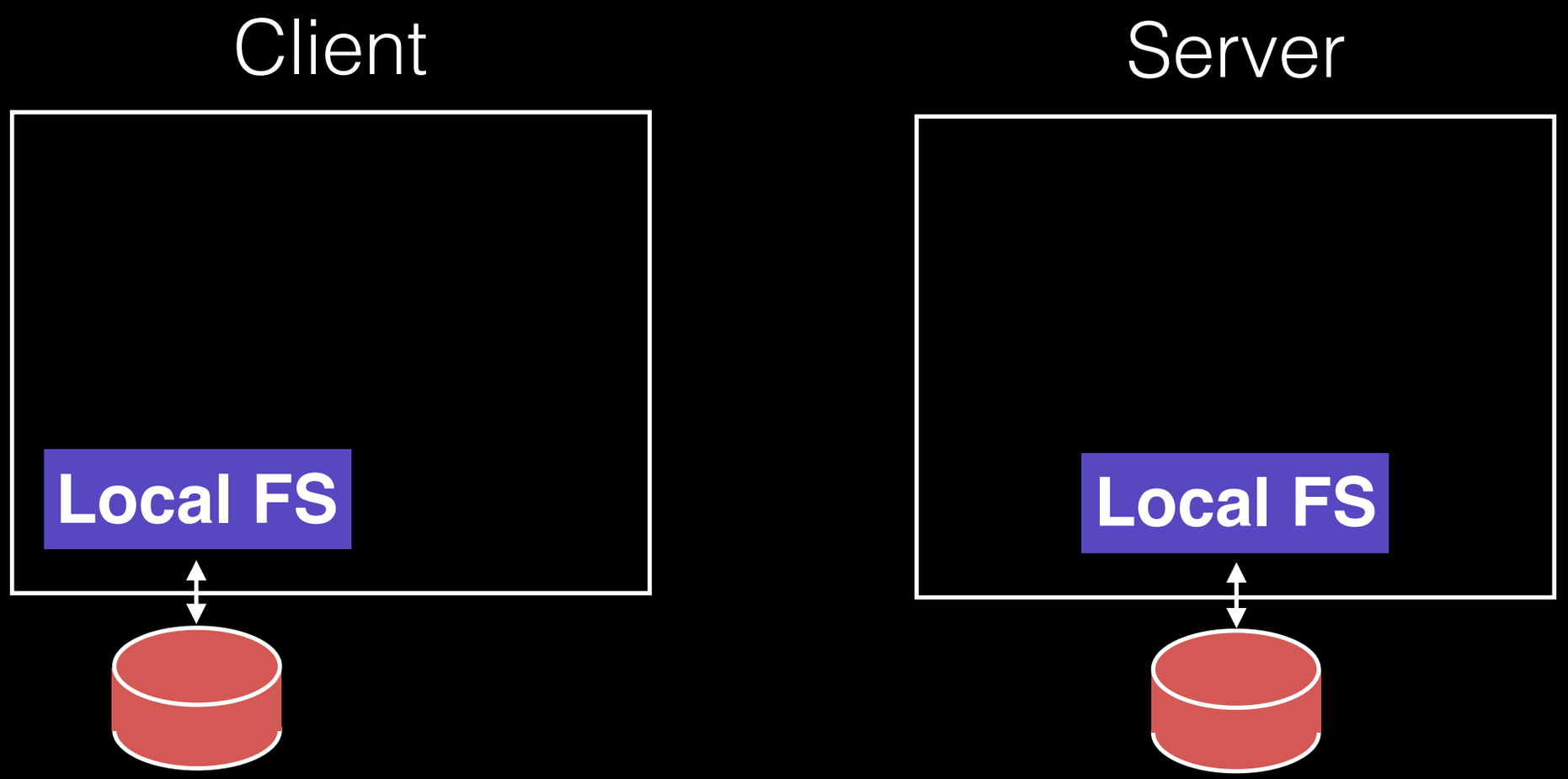
Today's Lecture

We're looking at NFSv2.

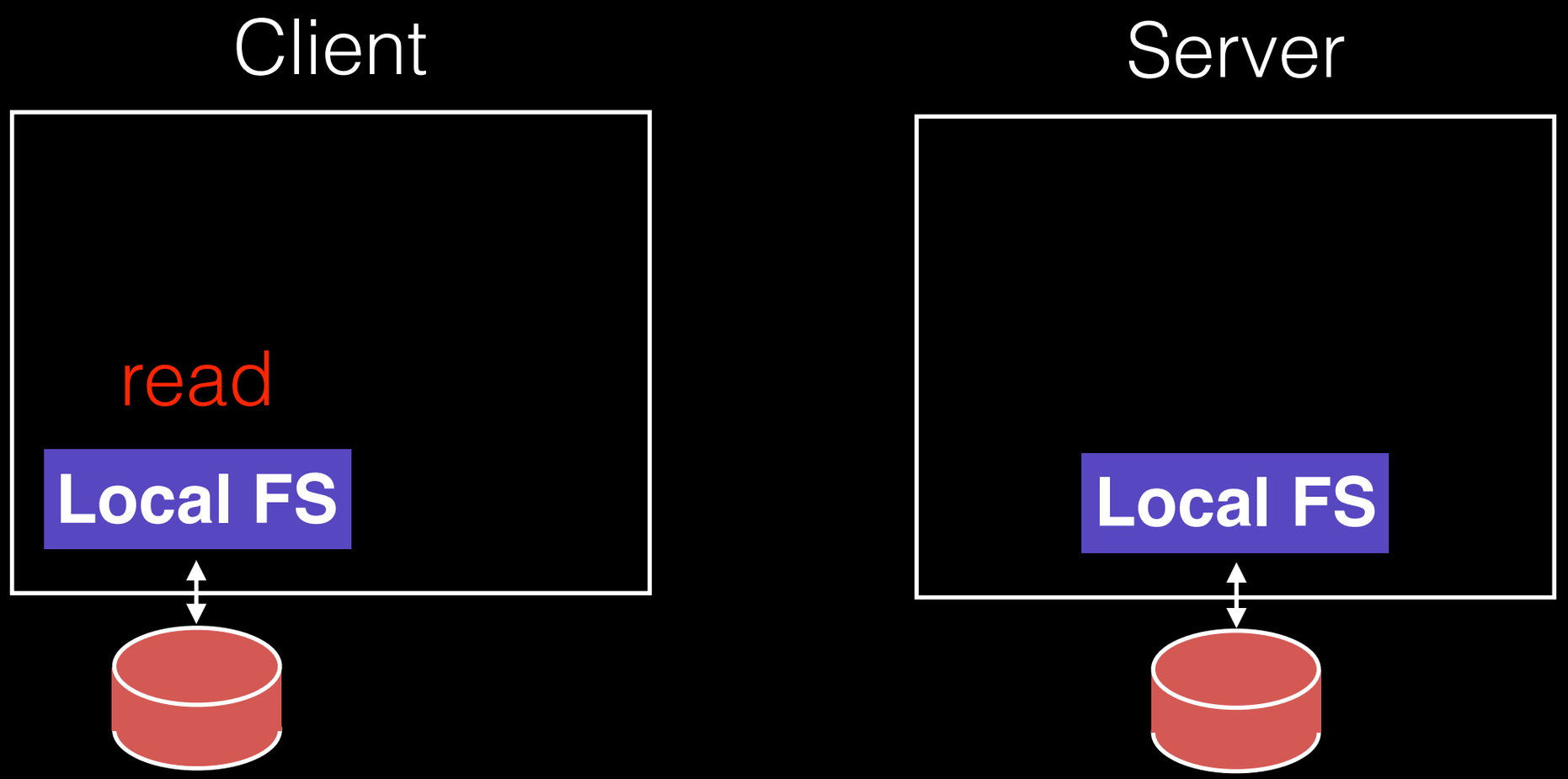
There is now an NFSv4 with many changes.

Why look at an older protocol?
To compare and contrast NFS with AFS.

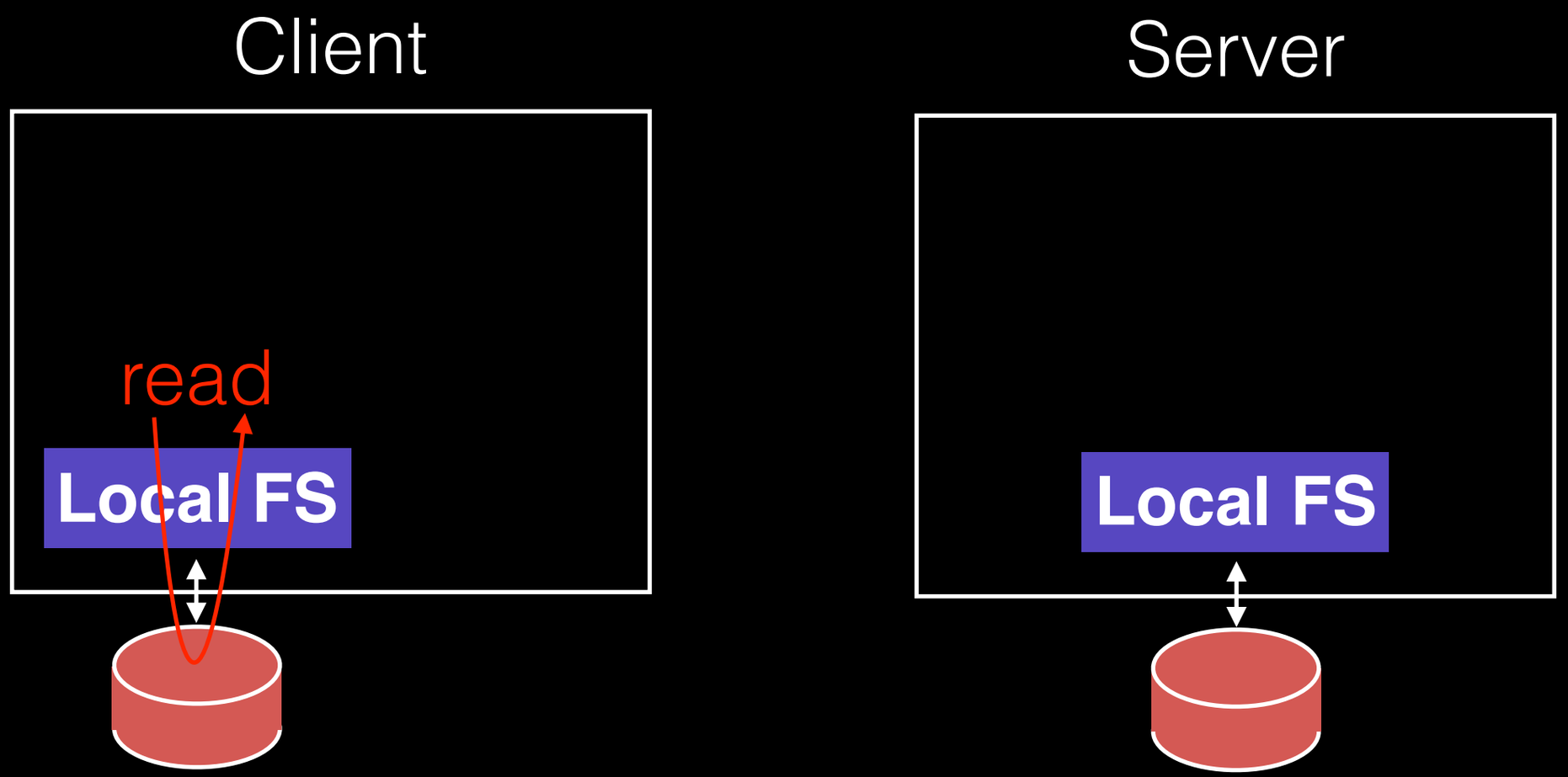
General Strategy: Export FS



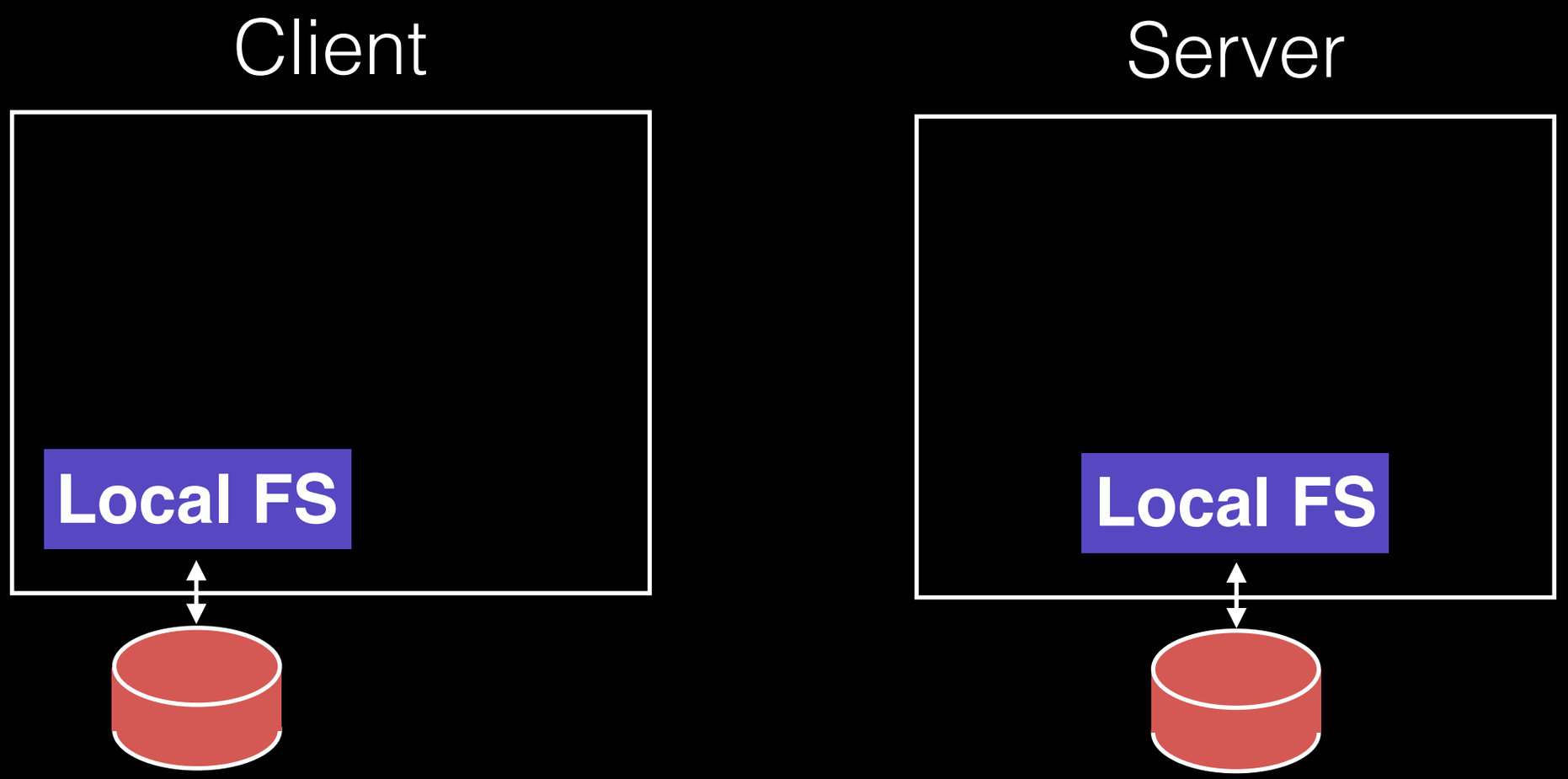
General Strategy: Export FS



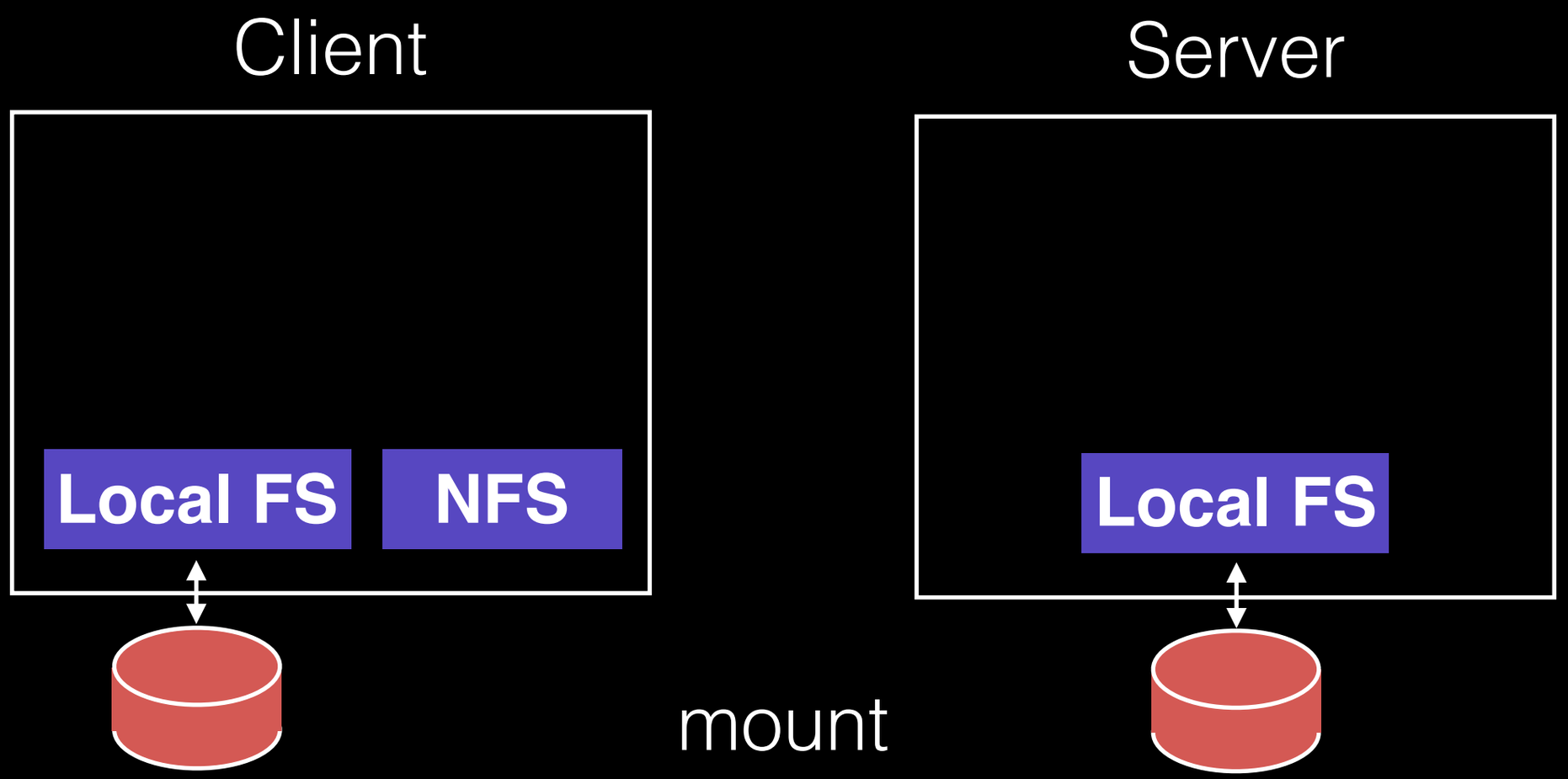
General Strategy: Export FS

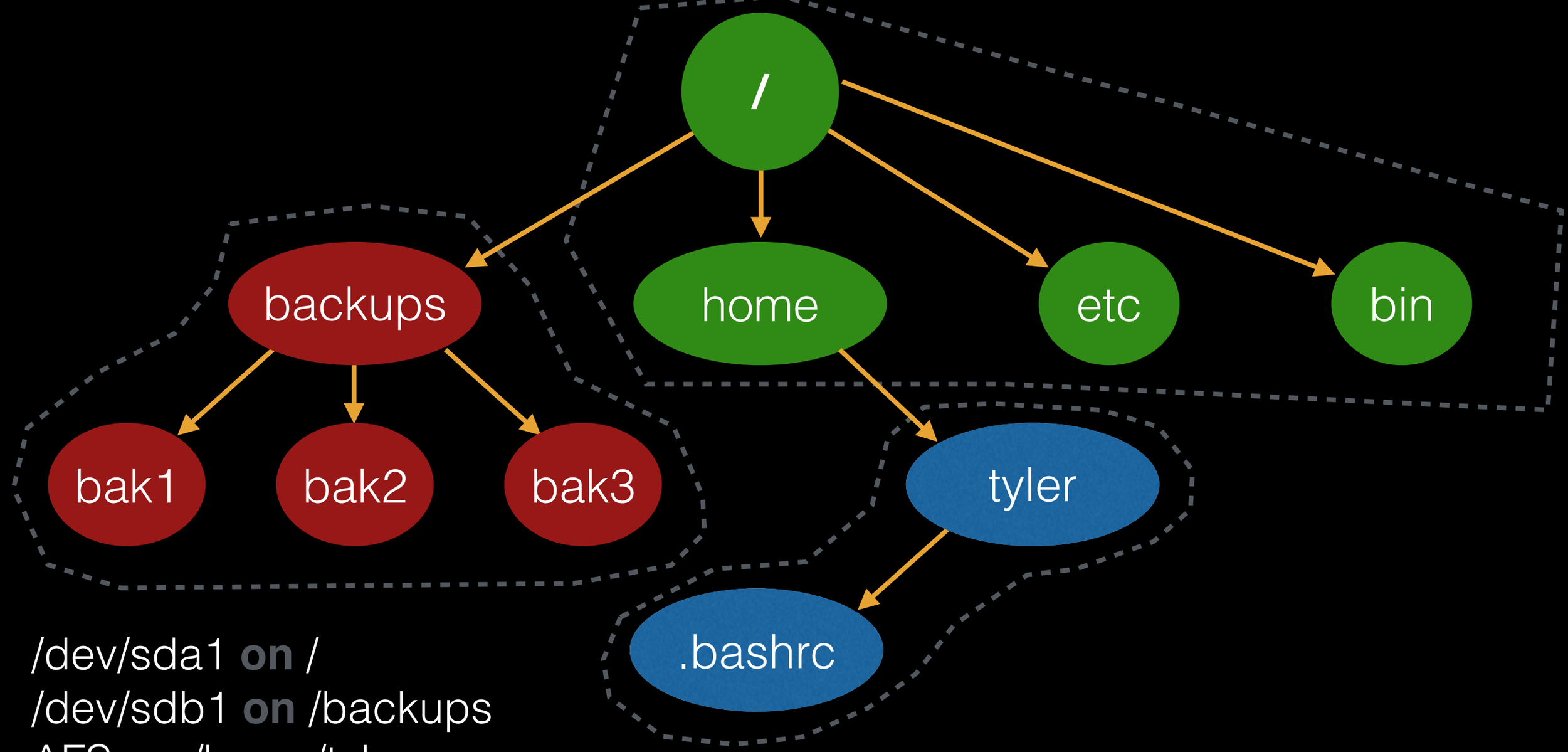


General Strategy: Export FS

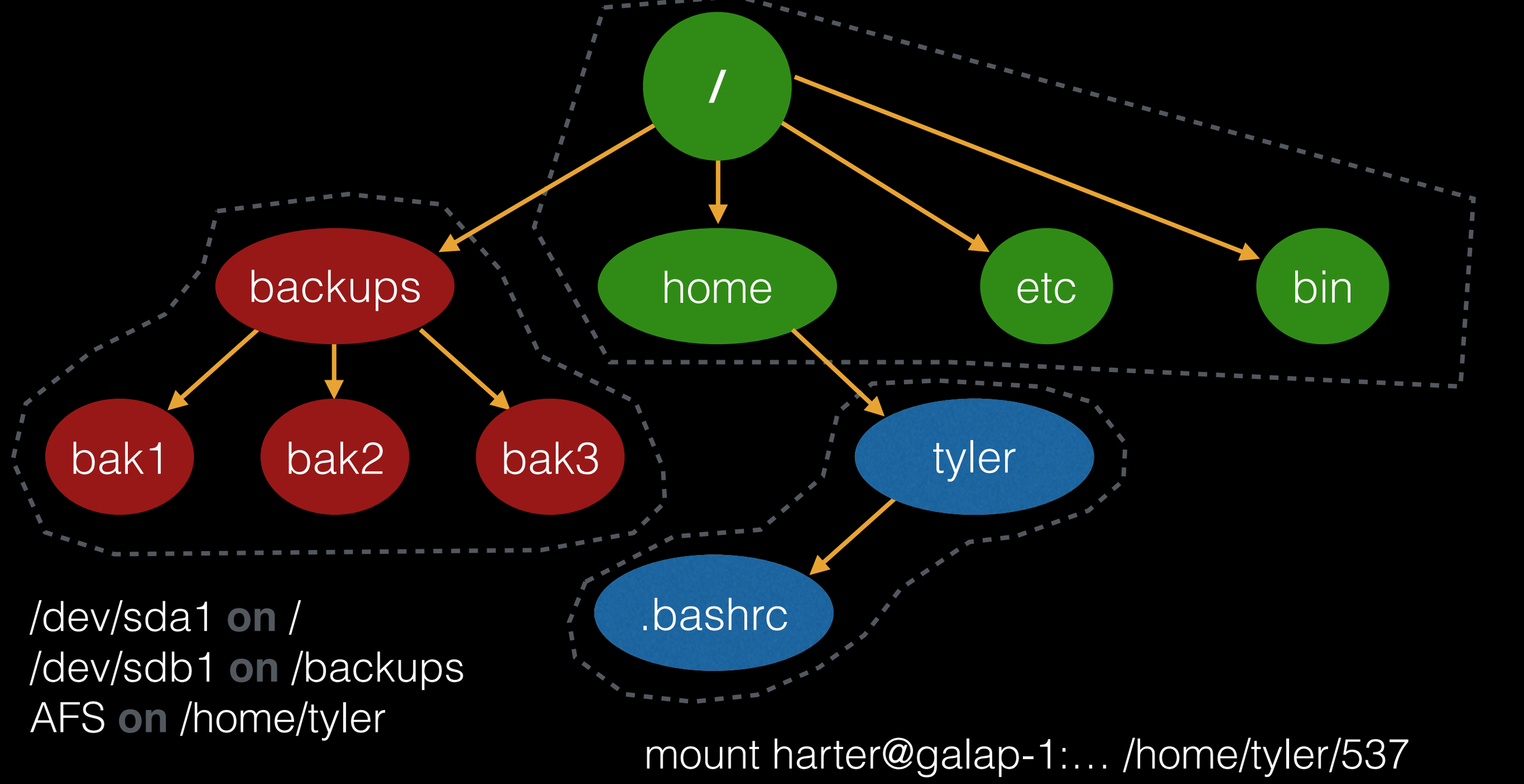


General Strategy: Export FS



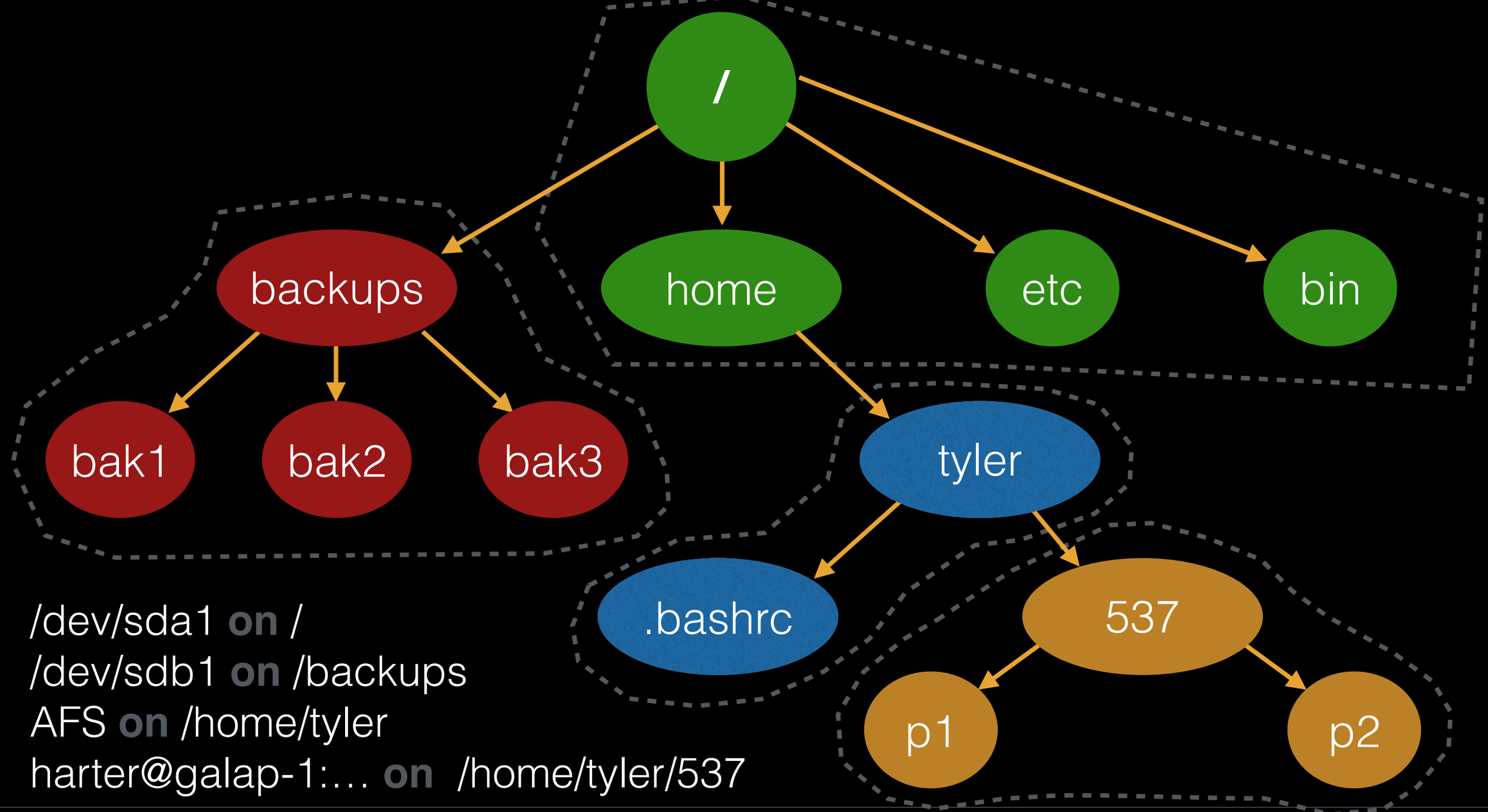


/dev/sda1 **on** /
/dev/sdb1 **on** /backups
AFS **on** /home/tyler



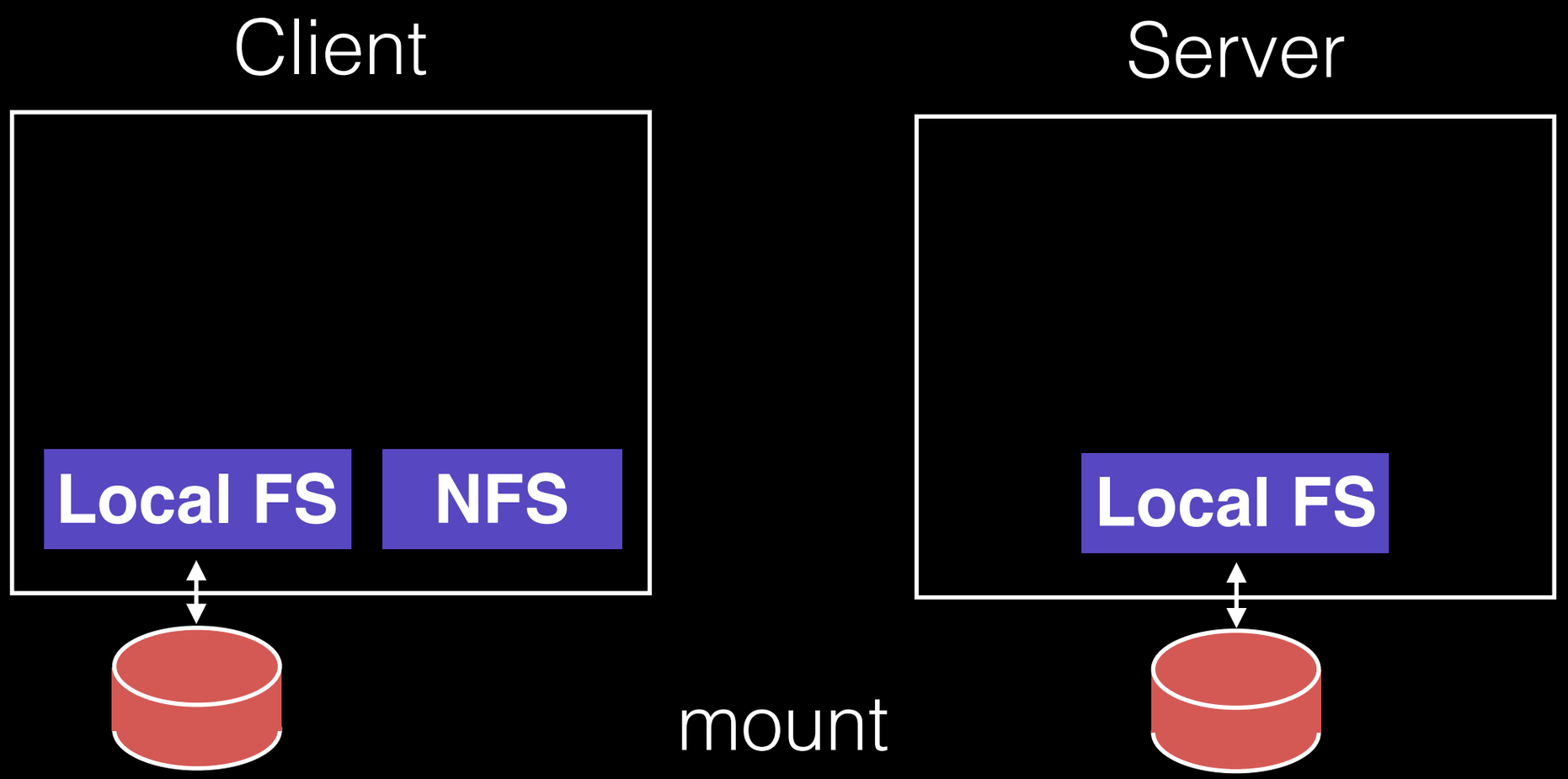
/dev/sda1 **on** /
/dev/sdb1 **on** /backups
AFS **on** /home/tyler

mount harter@galap-1:… /home/tyler/537

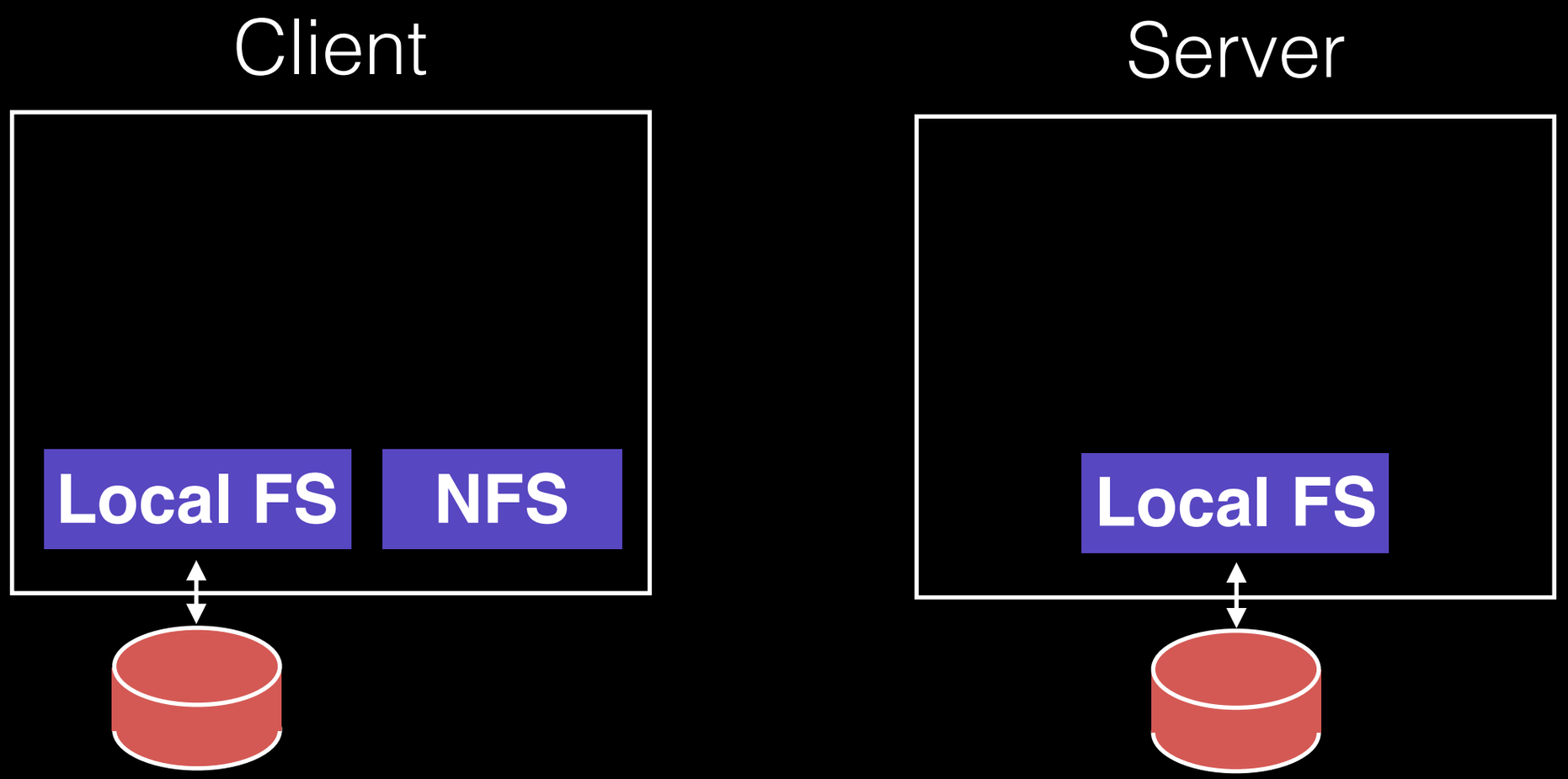


/dev/sda1 **on** /
/dev/sdb1 **on** /backups
AFS **on** /home/tyler
harter@galap-1:… **on** /home/tyler/537

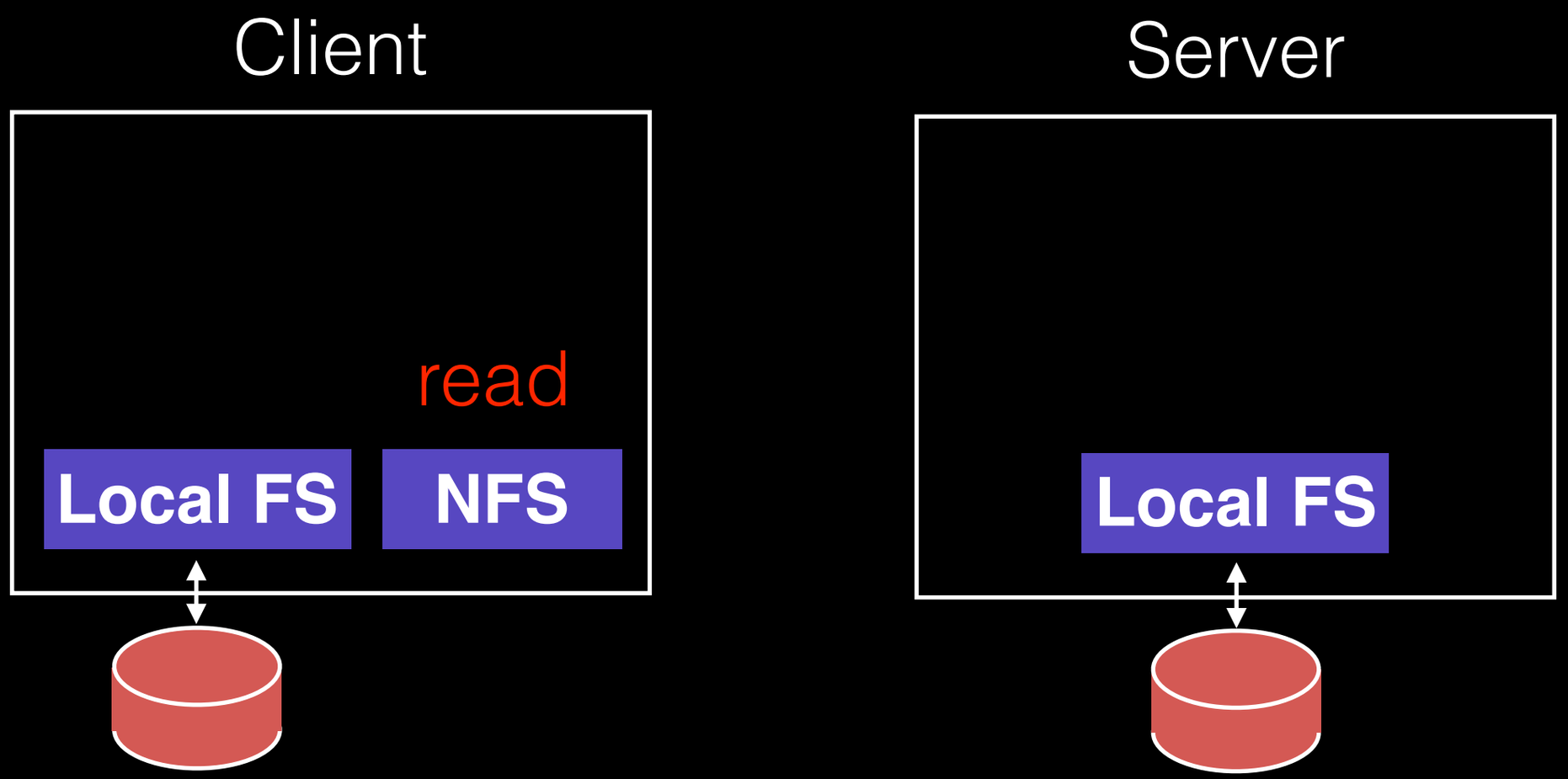
General Strategy: Export FS



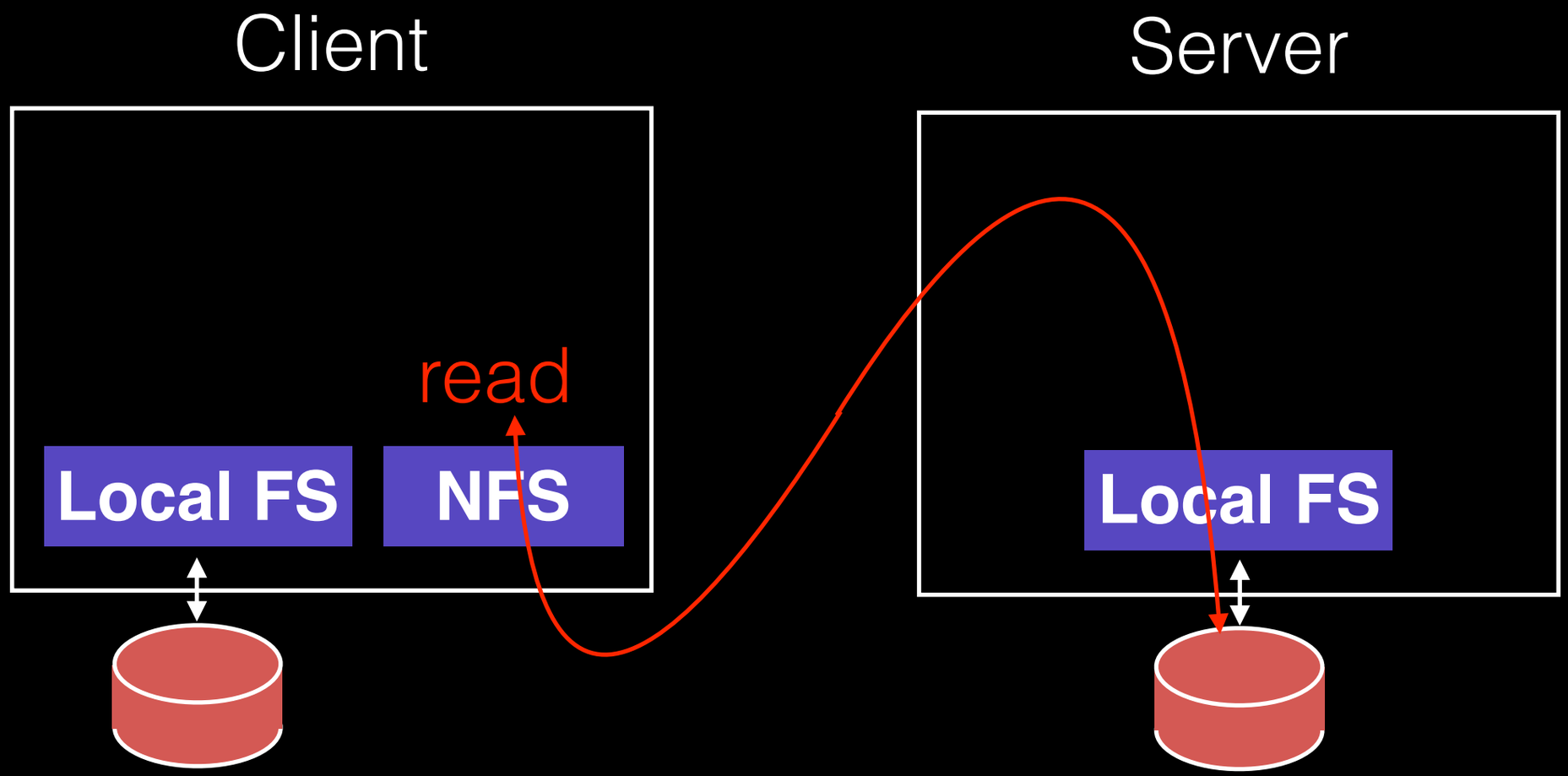
General Strategy: Export FS



General Strategy: Export FS



General Strategy: Export FS



Overview

Architecture

Network API

Write Buffering

Cache

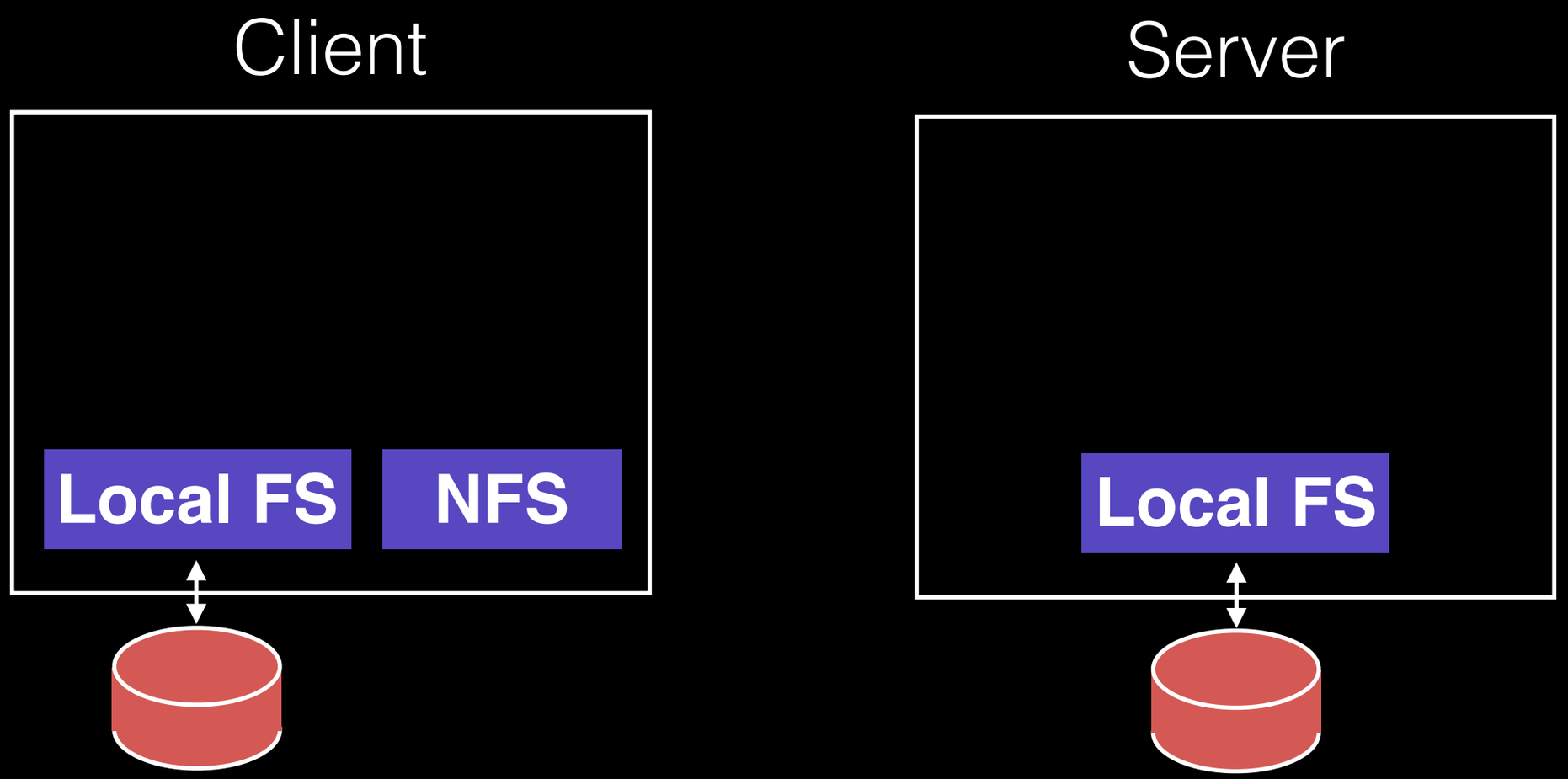
Strategy 1

Wrap regular UNIX system calls using RPC.

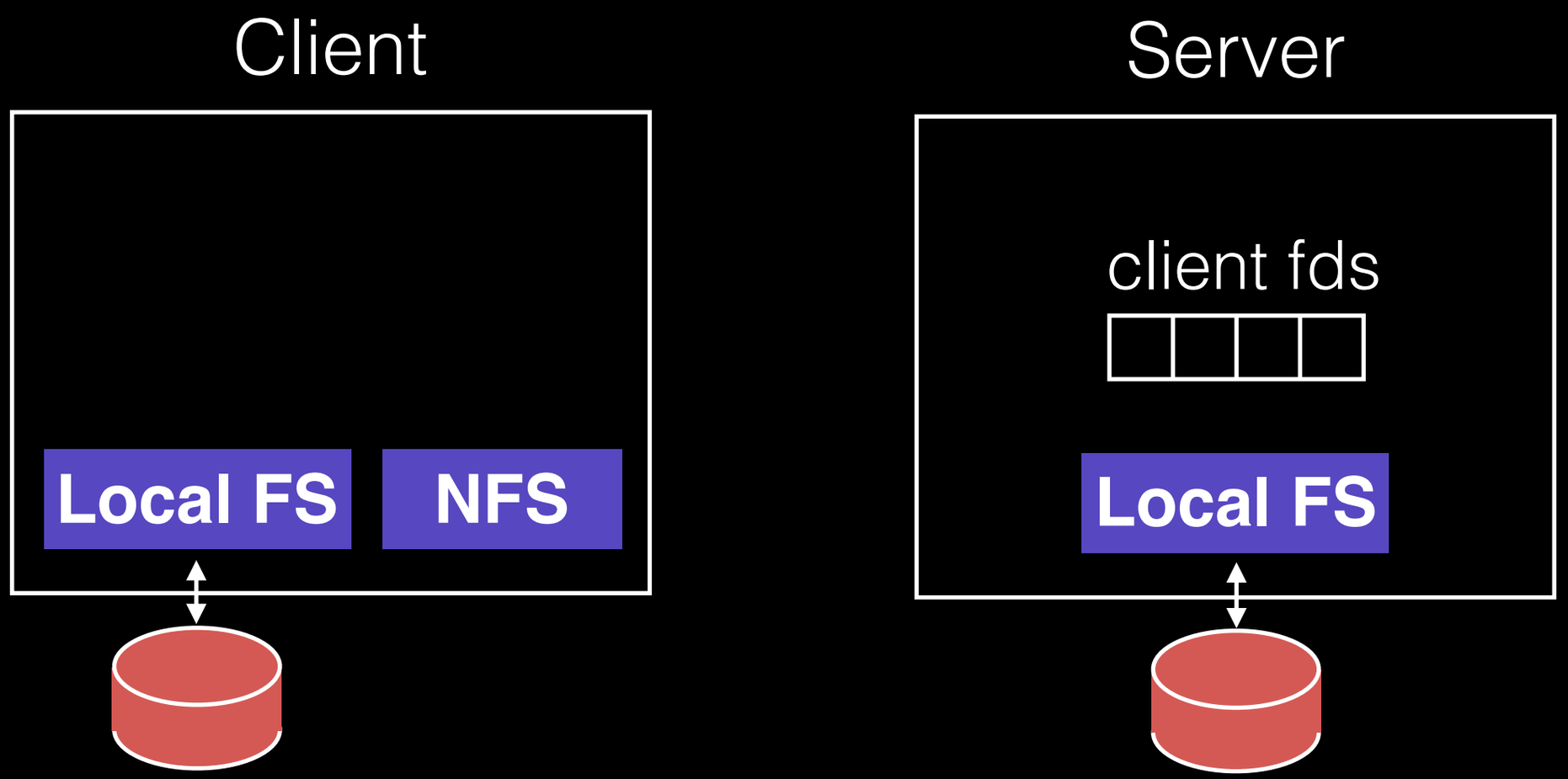
open() on client calls open() on server.
open() on server returns fd back to client.

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

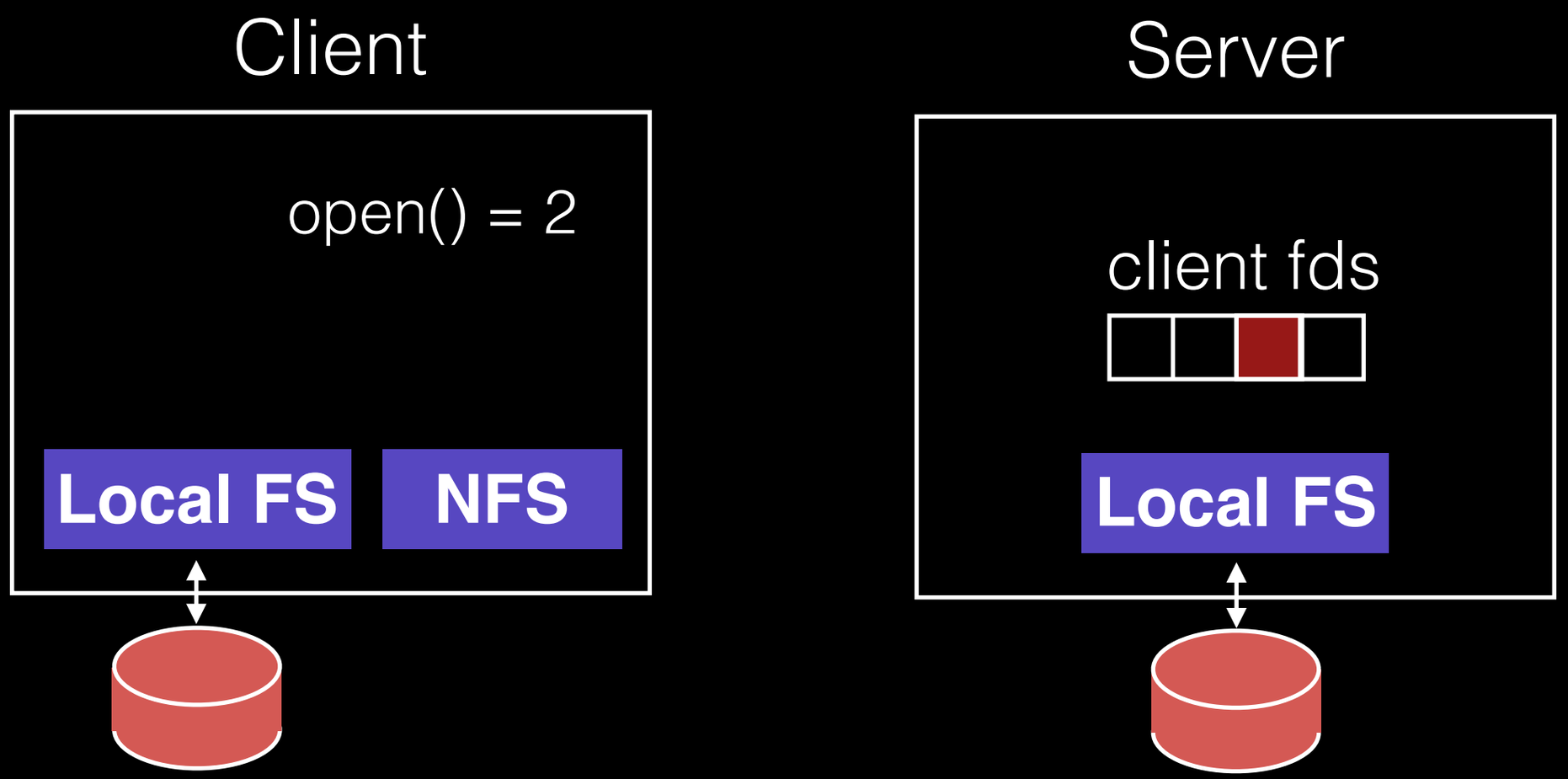
File Descriptors



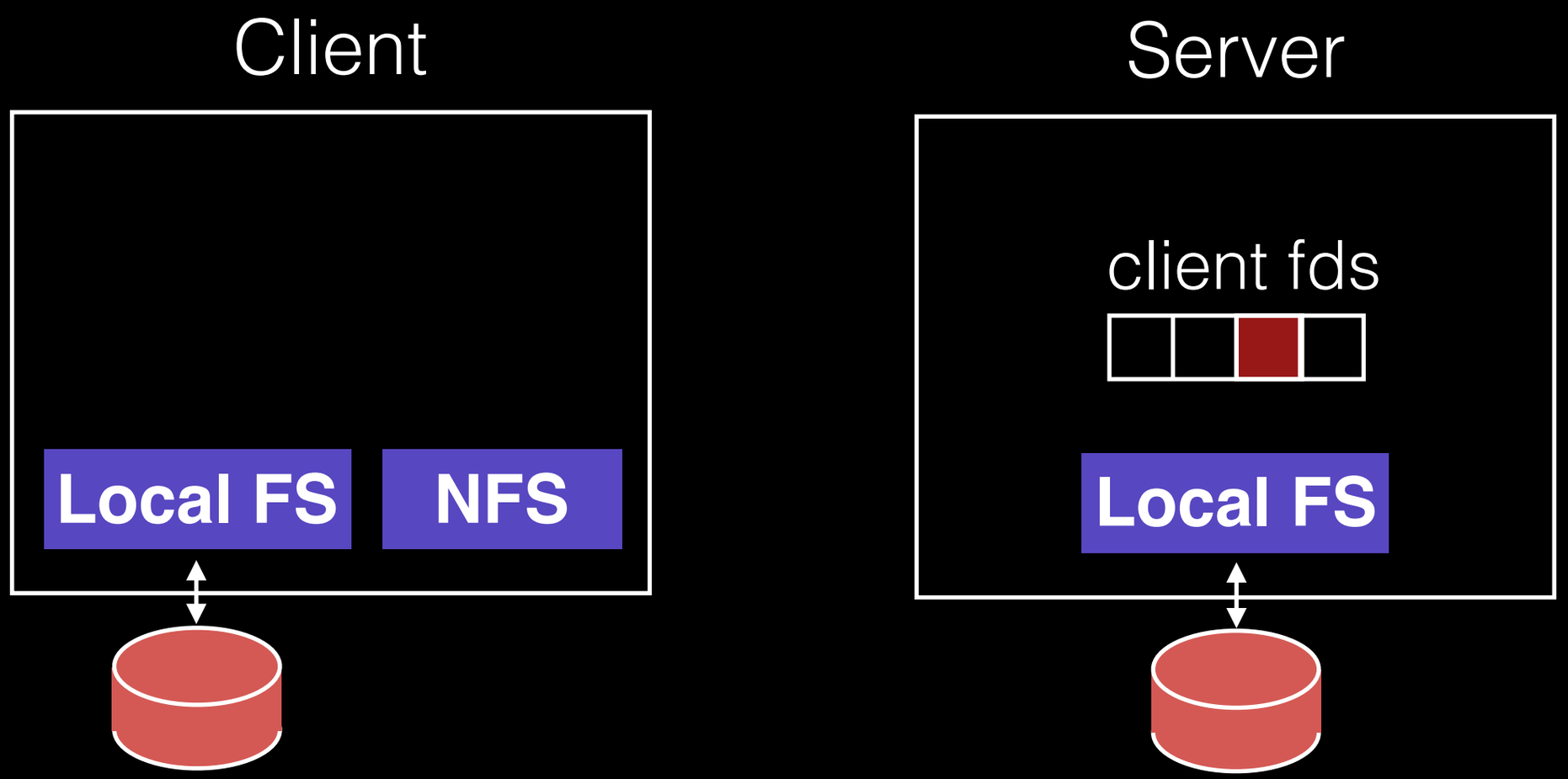
File Descriptors



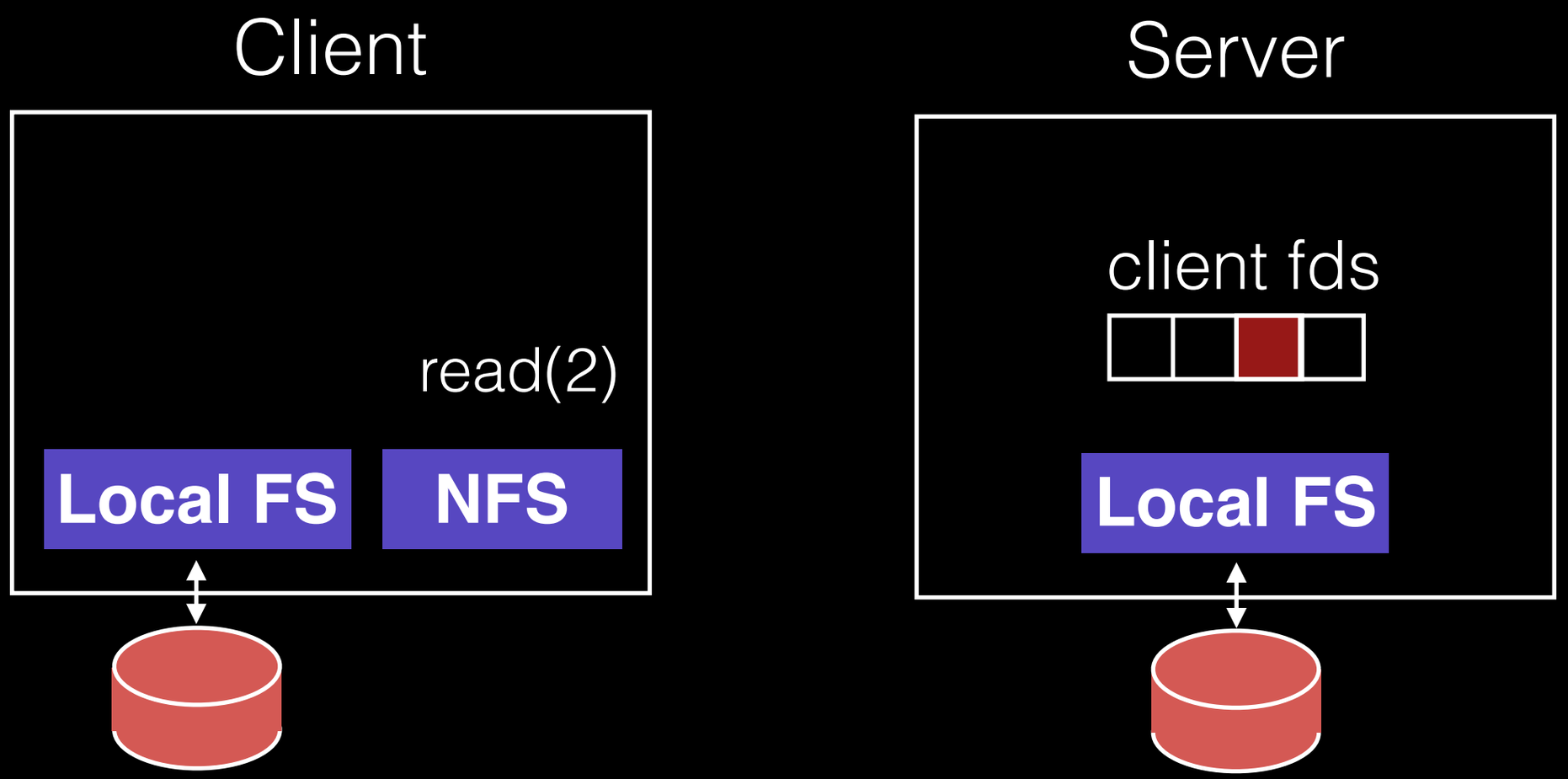
File Descriptors



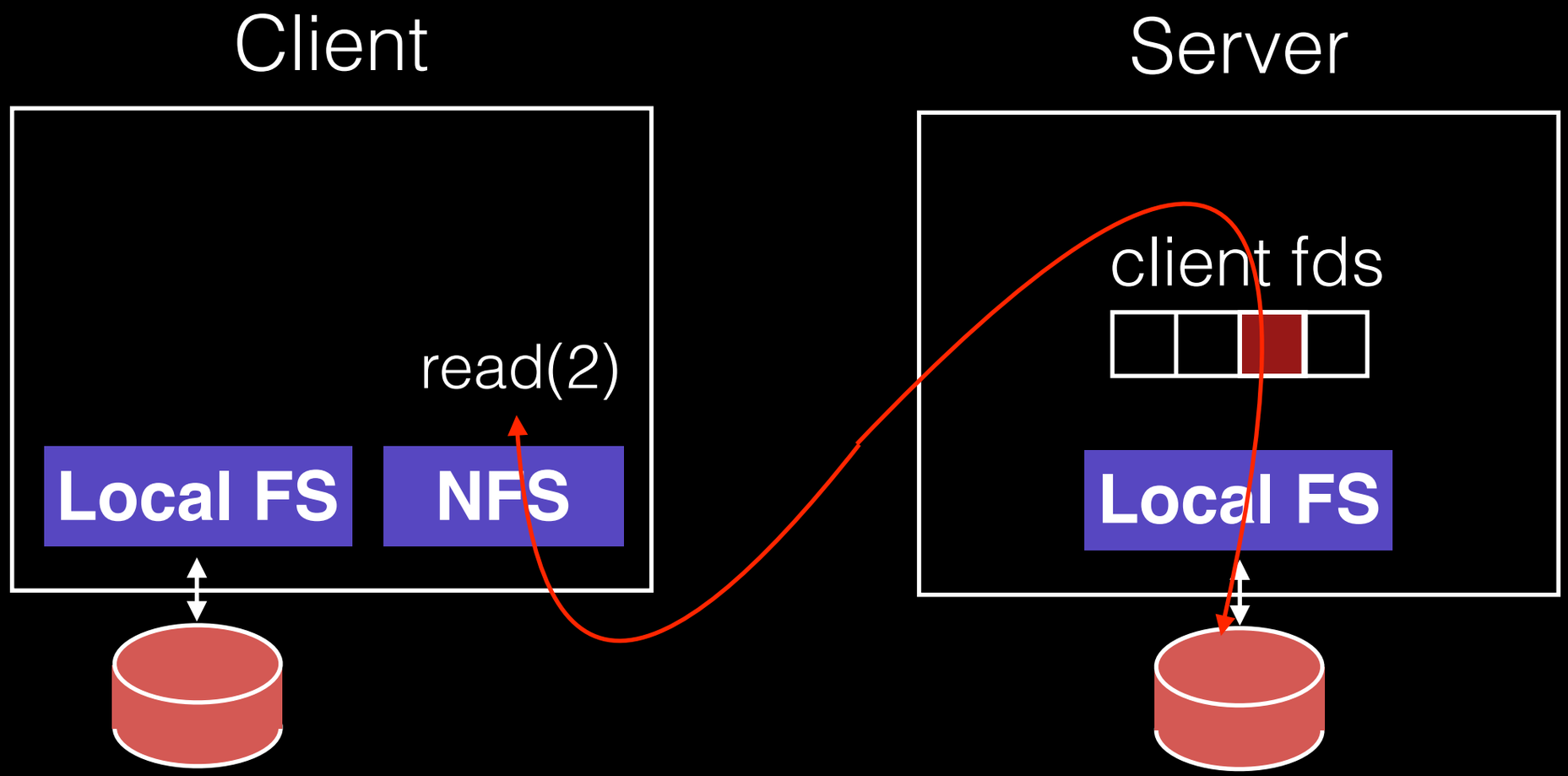
File Descriptors



File Descriptors



File Descriptors



Strategy 1 Problems

What about crashes?

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

crash!

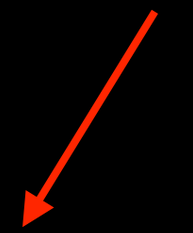
Imagine server crashes and reboots during reads…

Strategy 1 Problems

What about crashes?

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

nice if this just looks like a slow read



Imagine server crashes and reboots during reads…

Subgoals

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

Potential Solutions

1. Run some crash recovery protocol upon reboot.
   - complex

2. Persist fds on server disk.
   - slow
   - what if client crashes instead?

Subgoals

Fast+simple crash recovery ←

- both clients and file server may crash

Transparent access

- can't tell it's over the network
- normal UNIX semantics

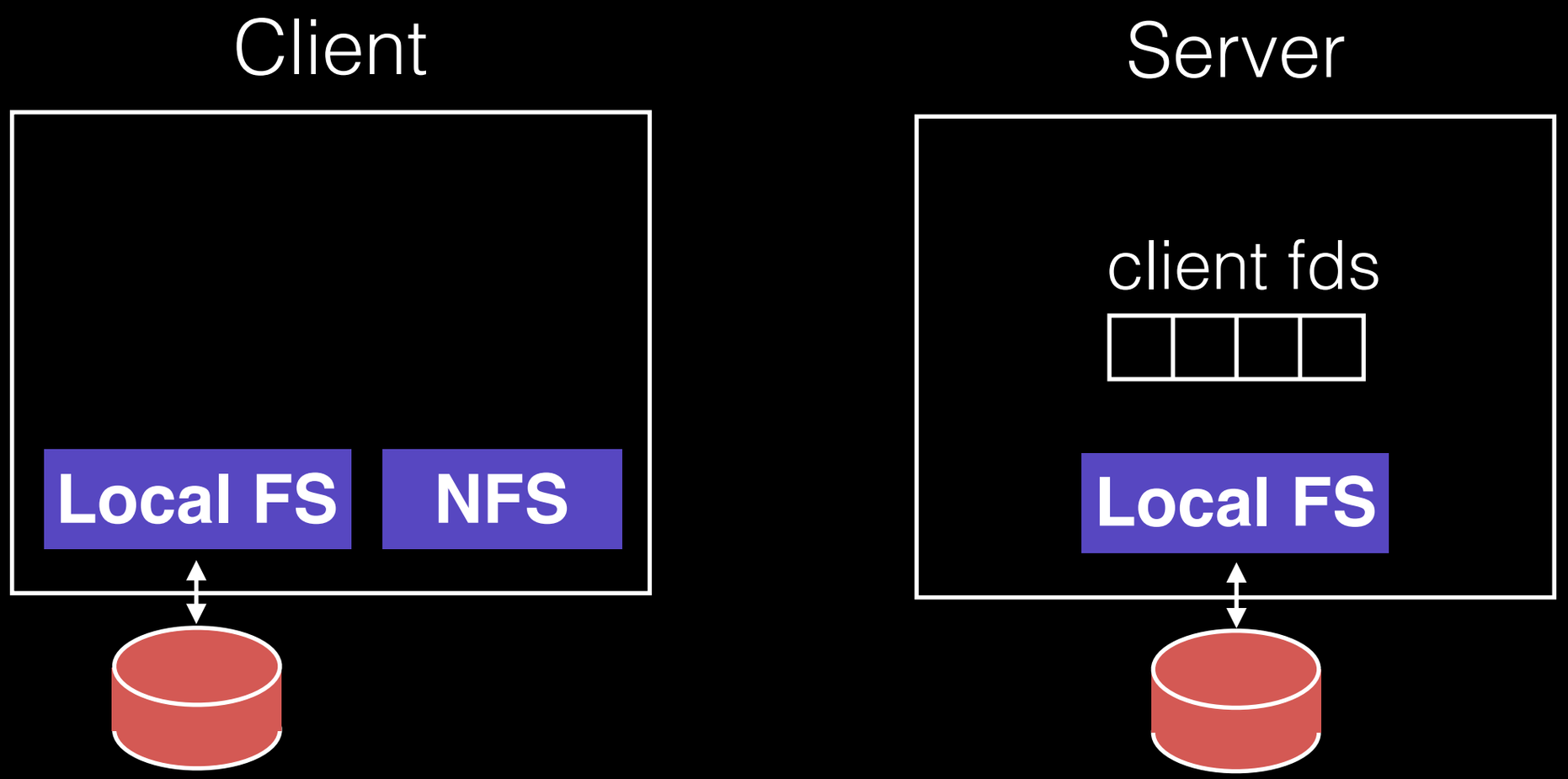
Reasonable performance ←

Strategy 2: put all info in requests

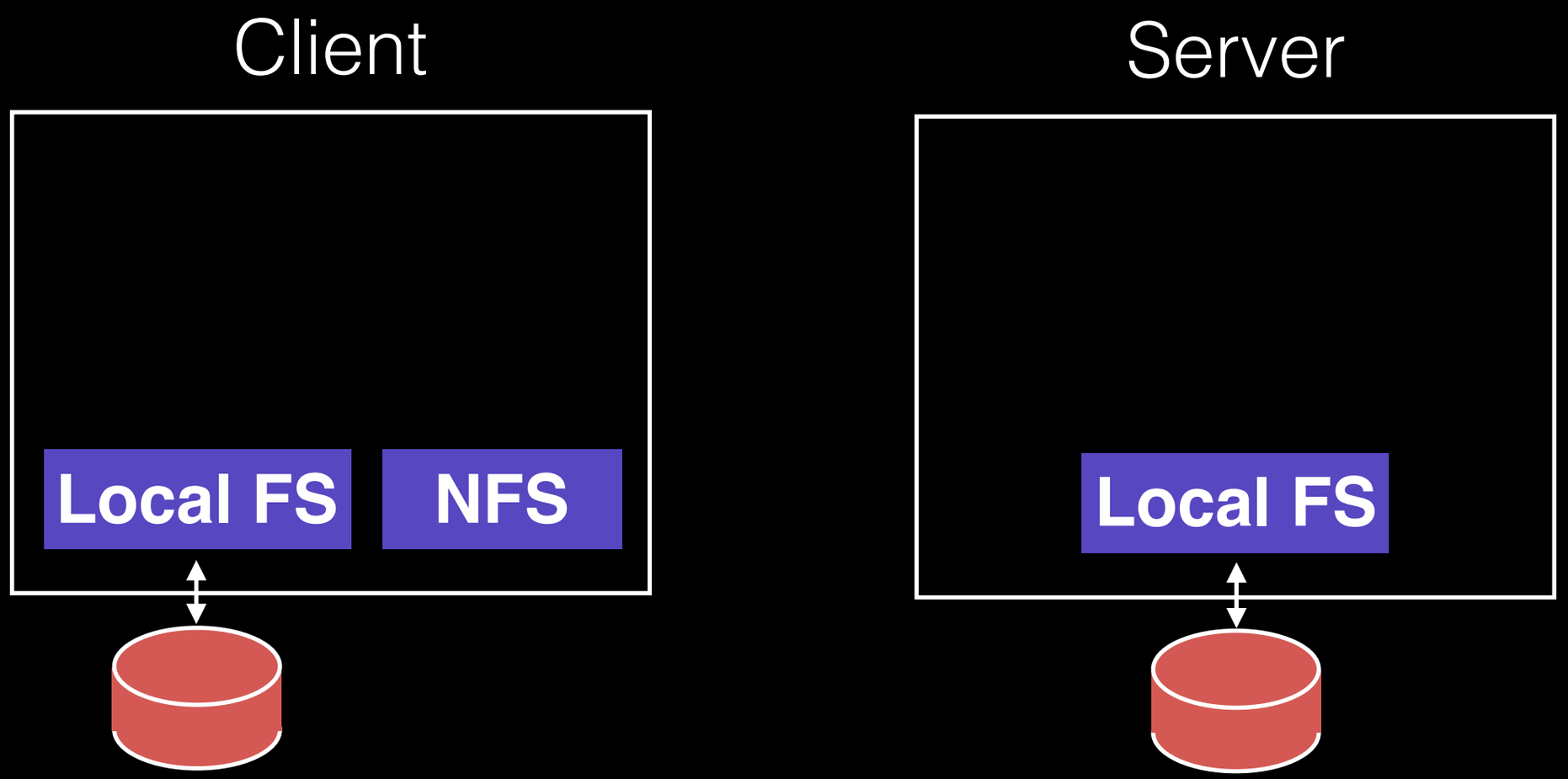
Use “stateless” protocol!

- server maintains no state about clients
- server still keeps other state, of course

Eliminate File Descriptors



Eliminate File Descriptors



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 need 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 need 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?

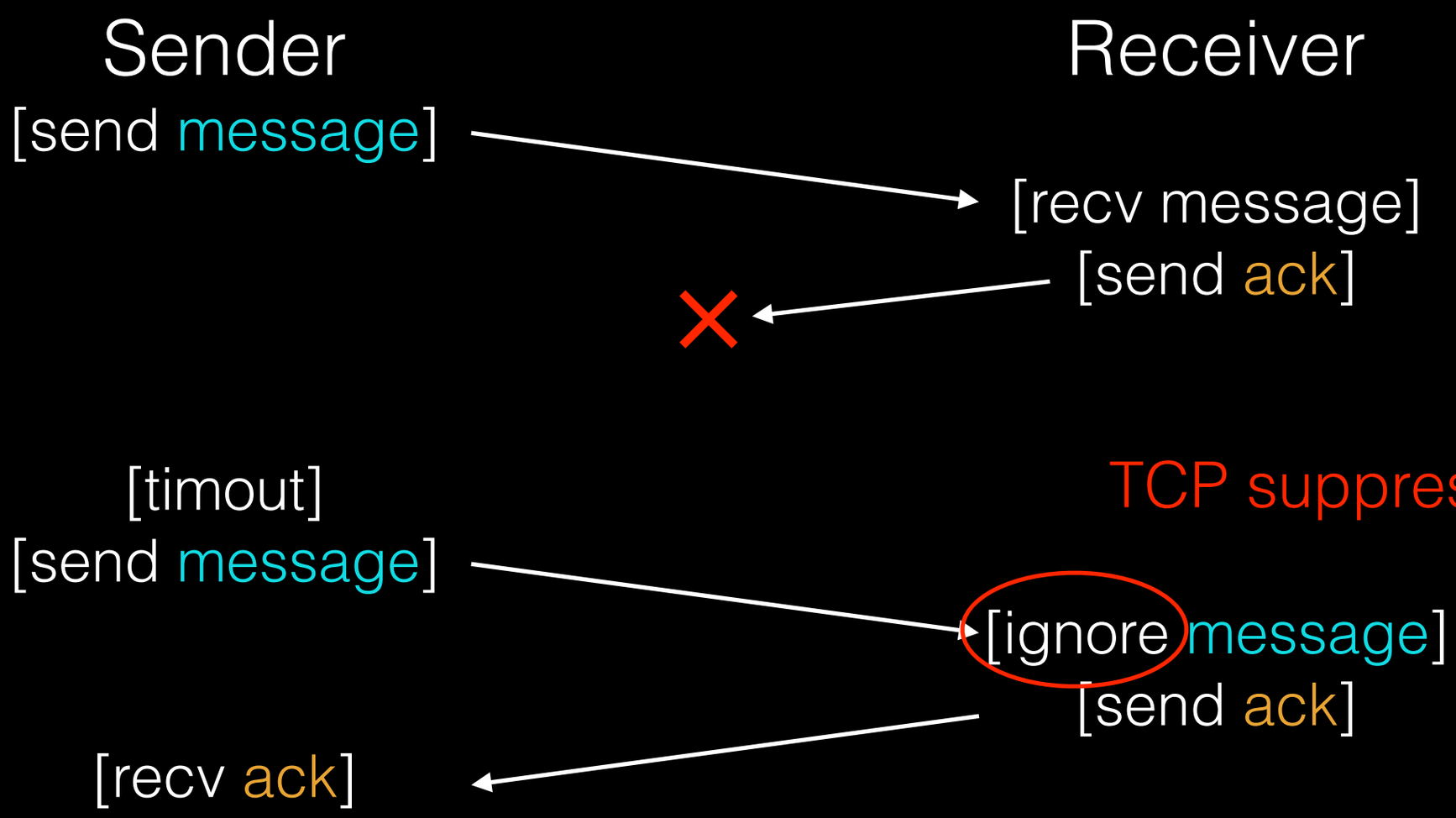
Aside: Append

```
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!

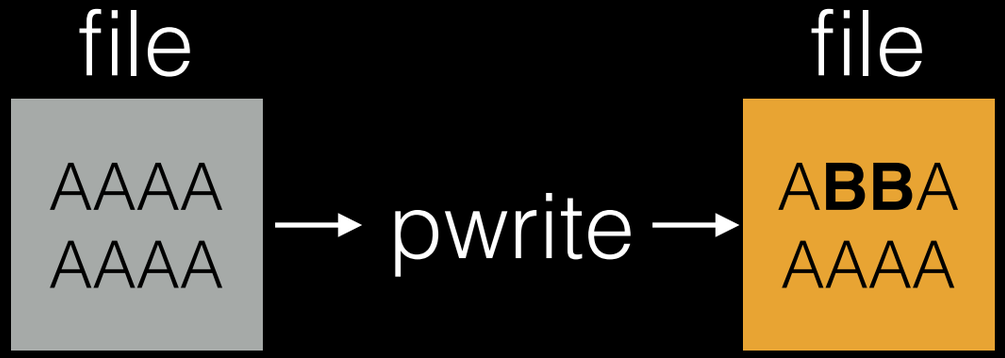
Replica Suppression is Stateful

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

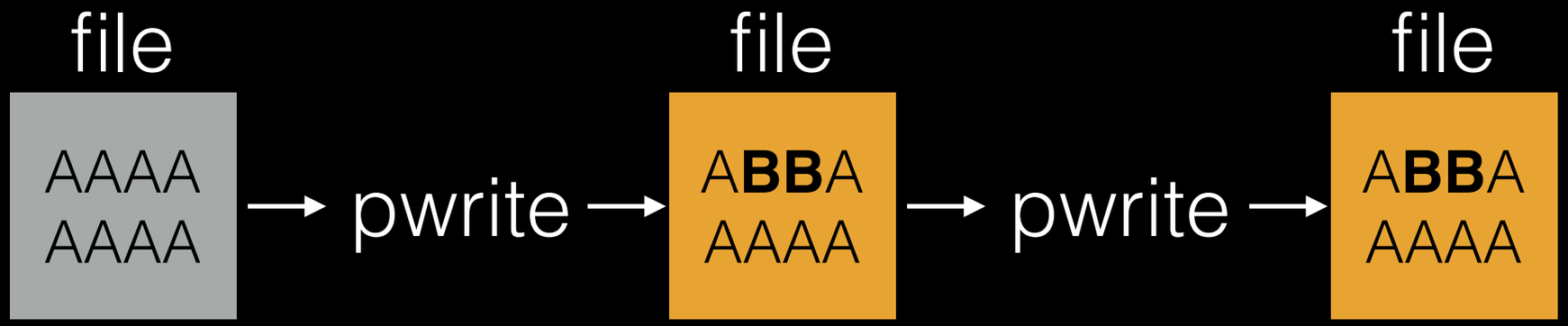
Solution: design API so that there is no harm is executing a call more than once.

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

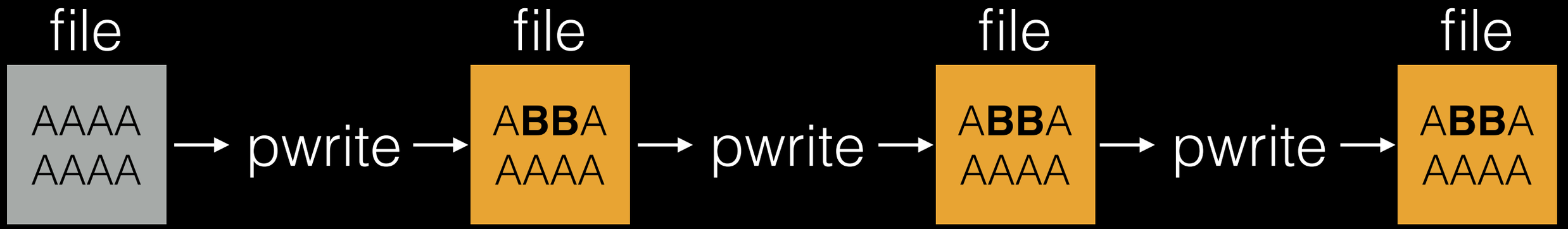
pwrite is idempotent



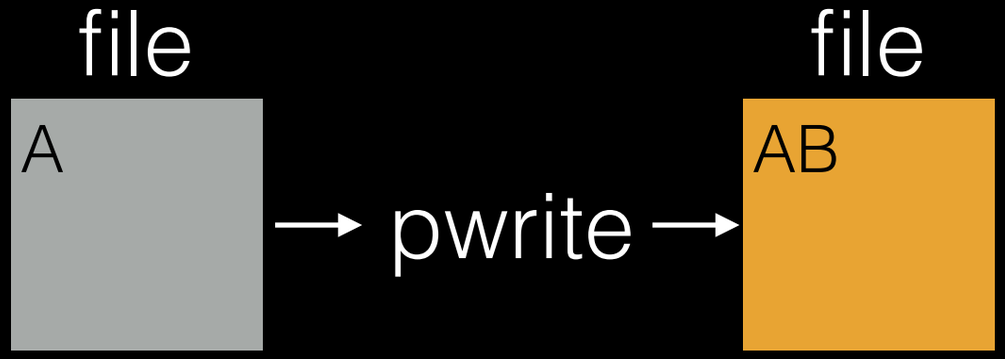
pwrite is idempotent



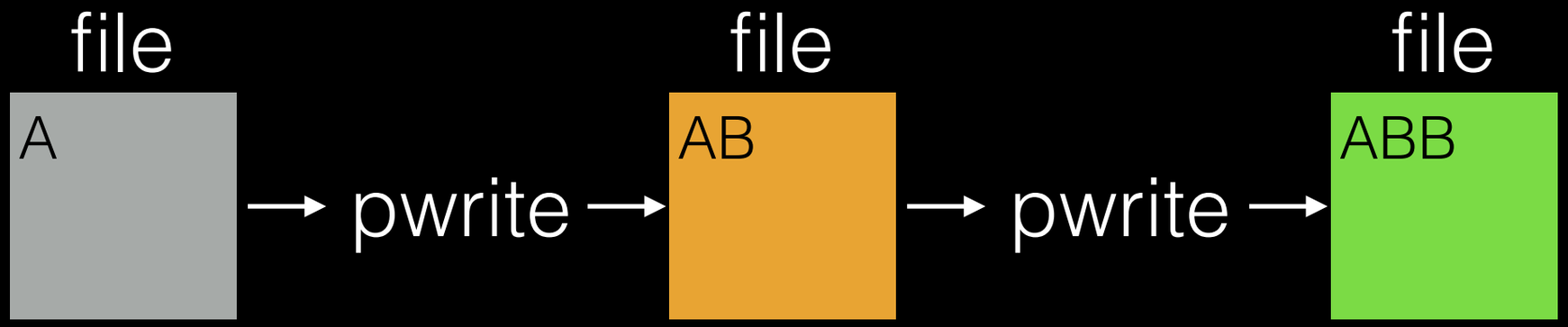
pwrite is idempotent



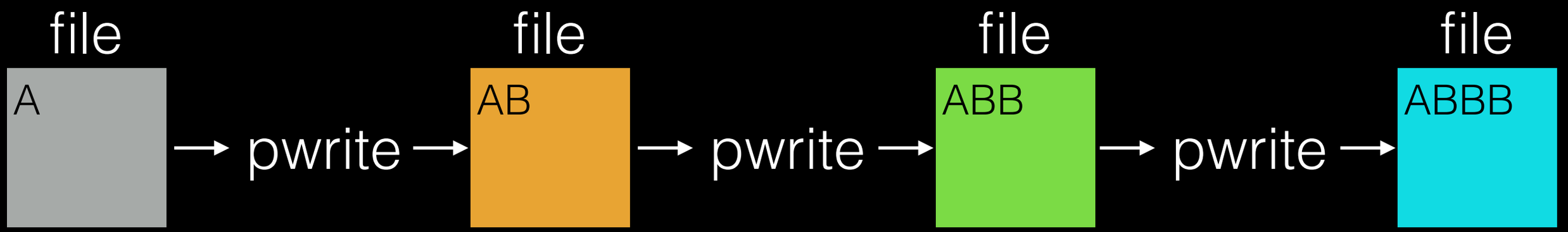
append is NOT idempotent



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 #>

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 #>

Subgoals

Fast+simple crash recovery
 - both clients and file server may crash

Transparent access
 - can't tell it's over the network
 - normal UNIX semantics

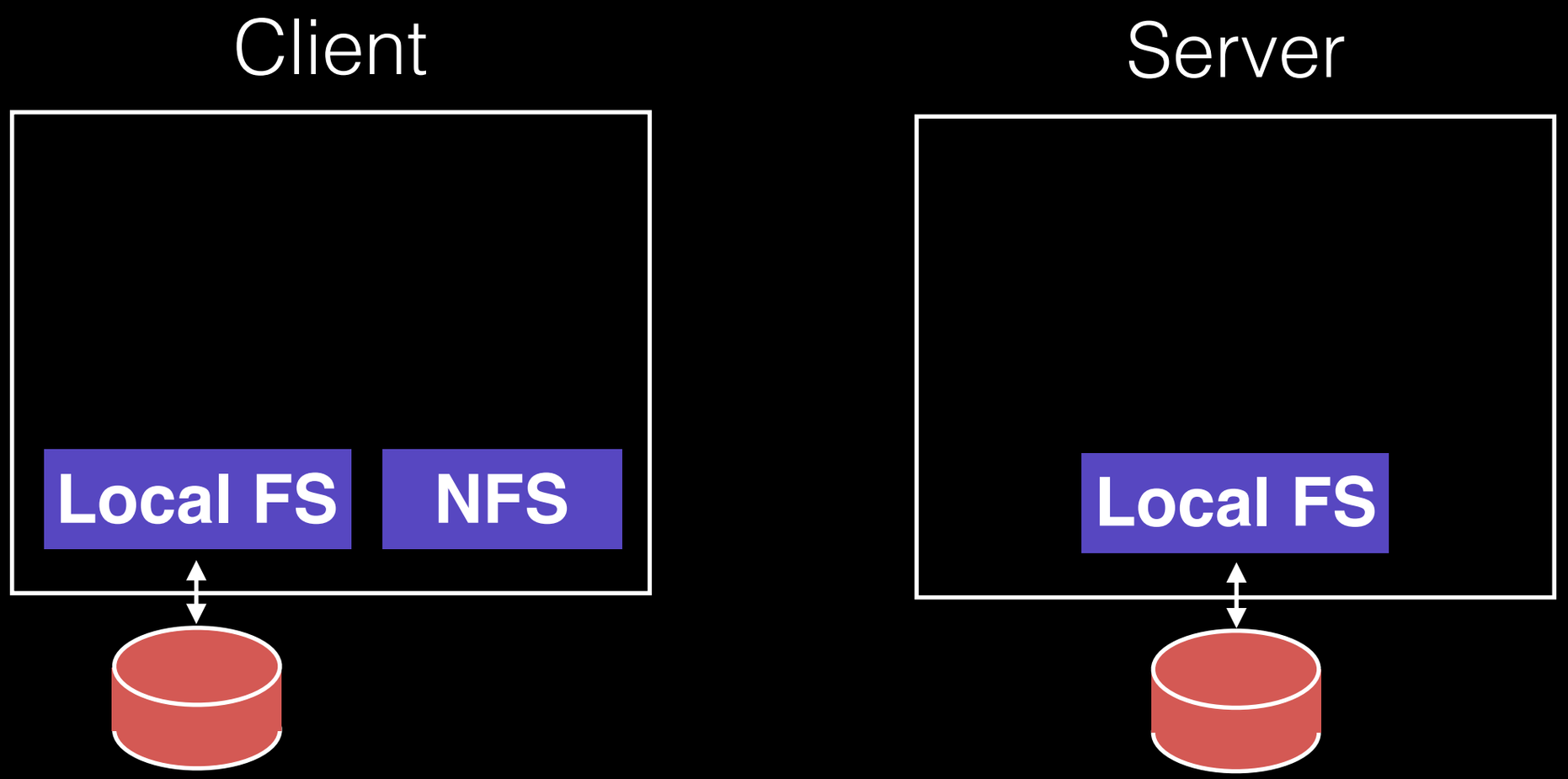
Reasonable performance

Strategy 5: client logic

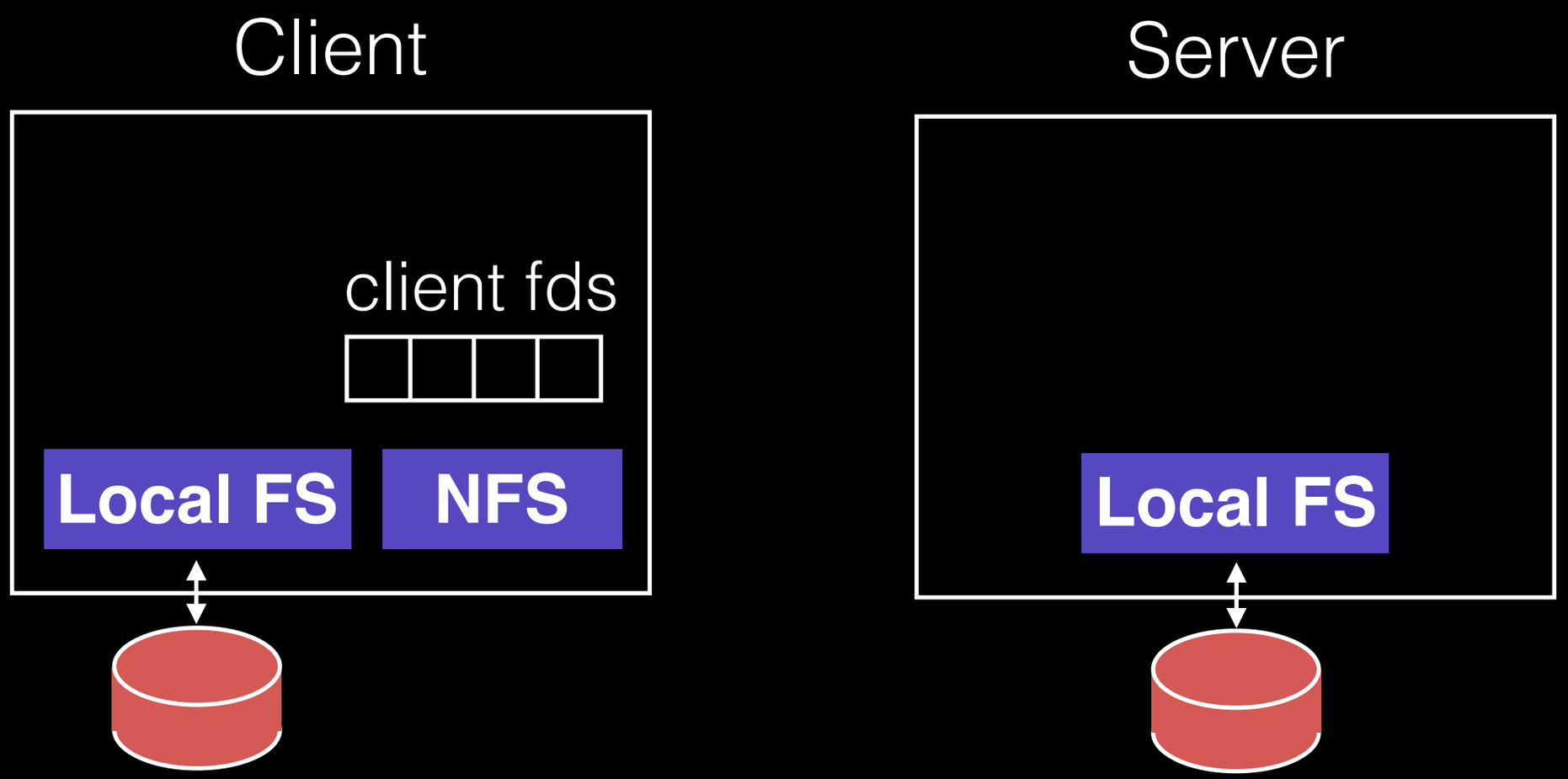
Build normal UNIX API on client side on top of the idempotent, RPC-based API we have described.

Client `open()` creates a local fd object. It contains:
- file handle
- offset

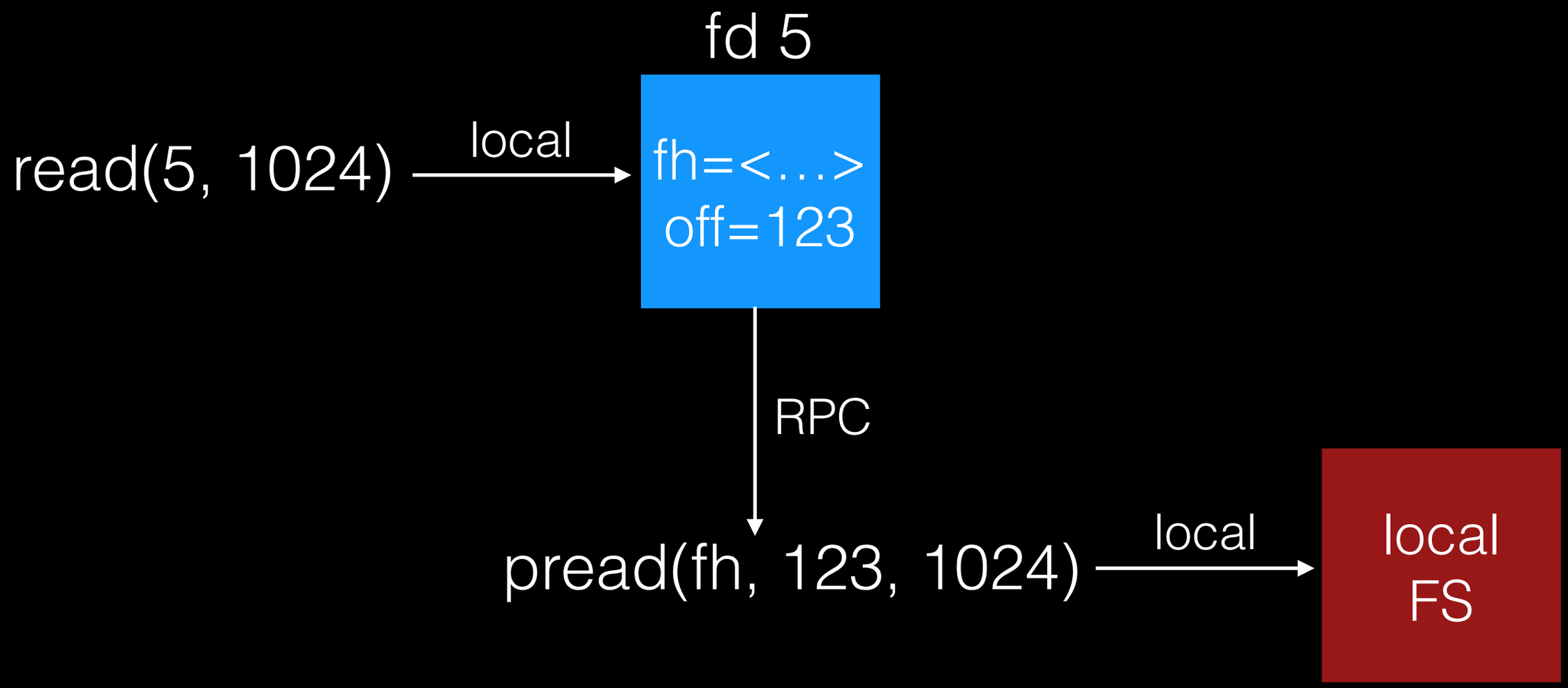
File Descriptors



File Descriptors



File Descriptors



Overview

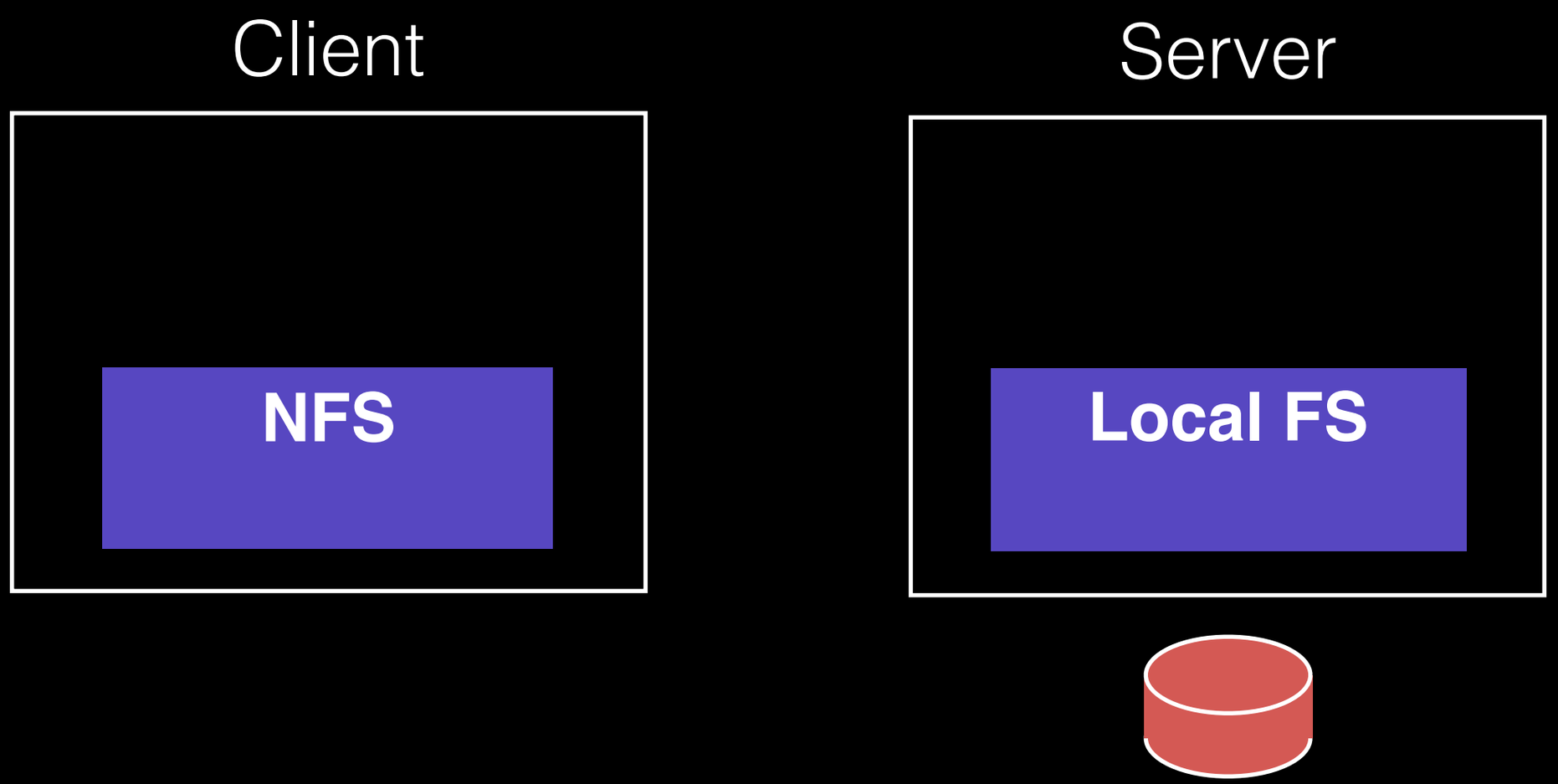
Architecture

Network API

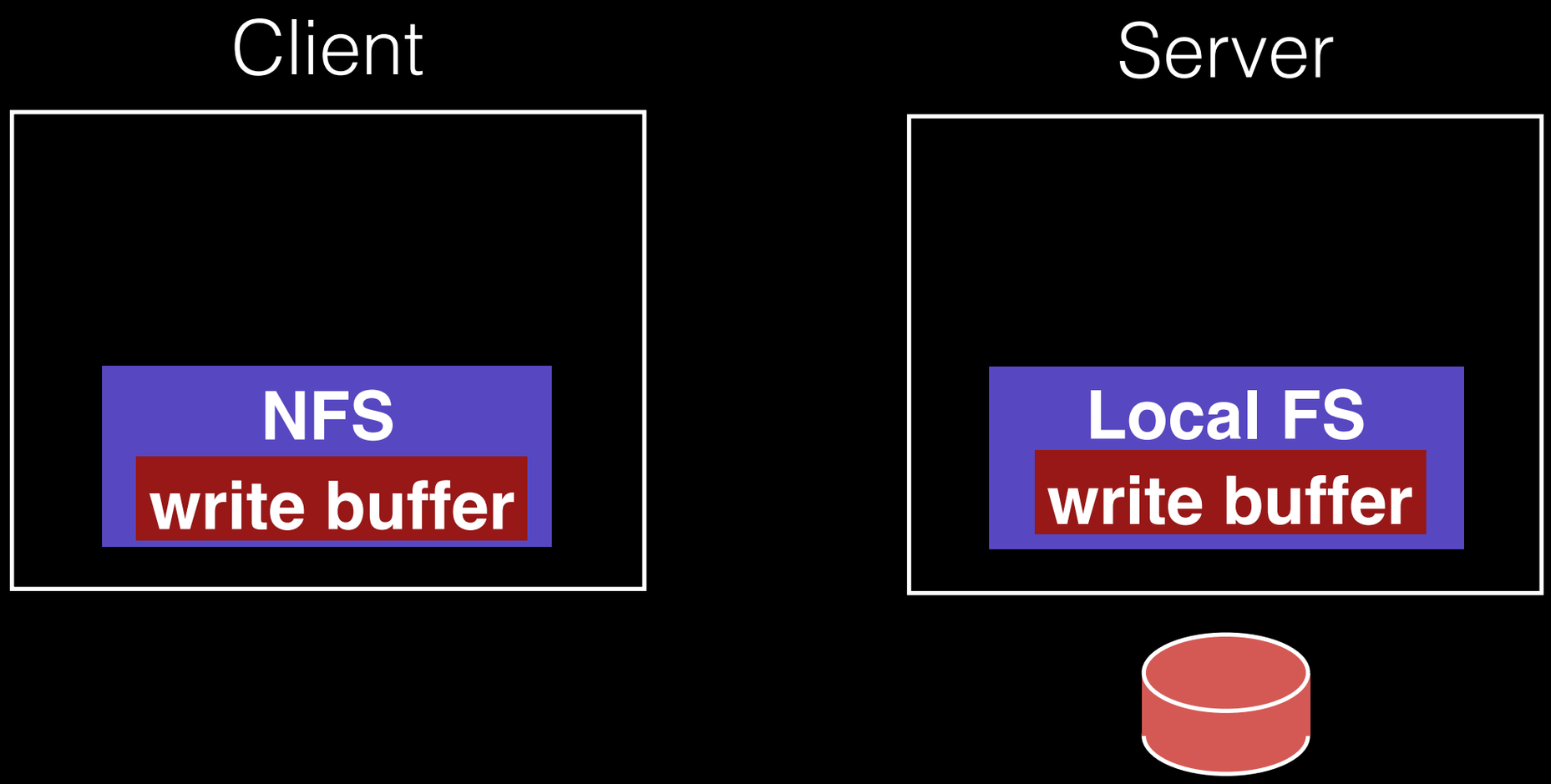
Write Buffering

Cache

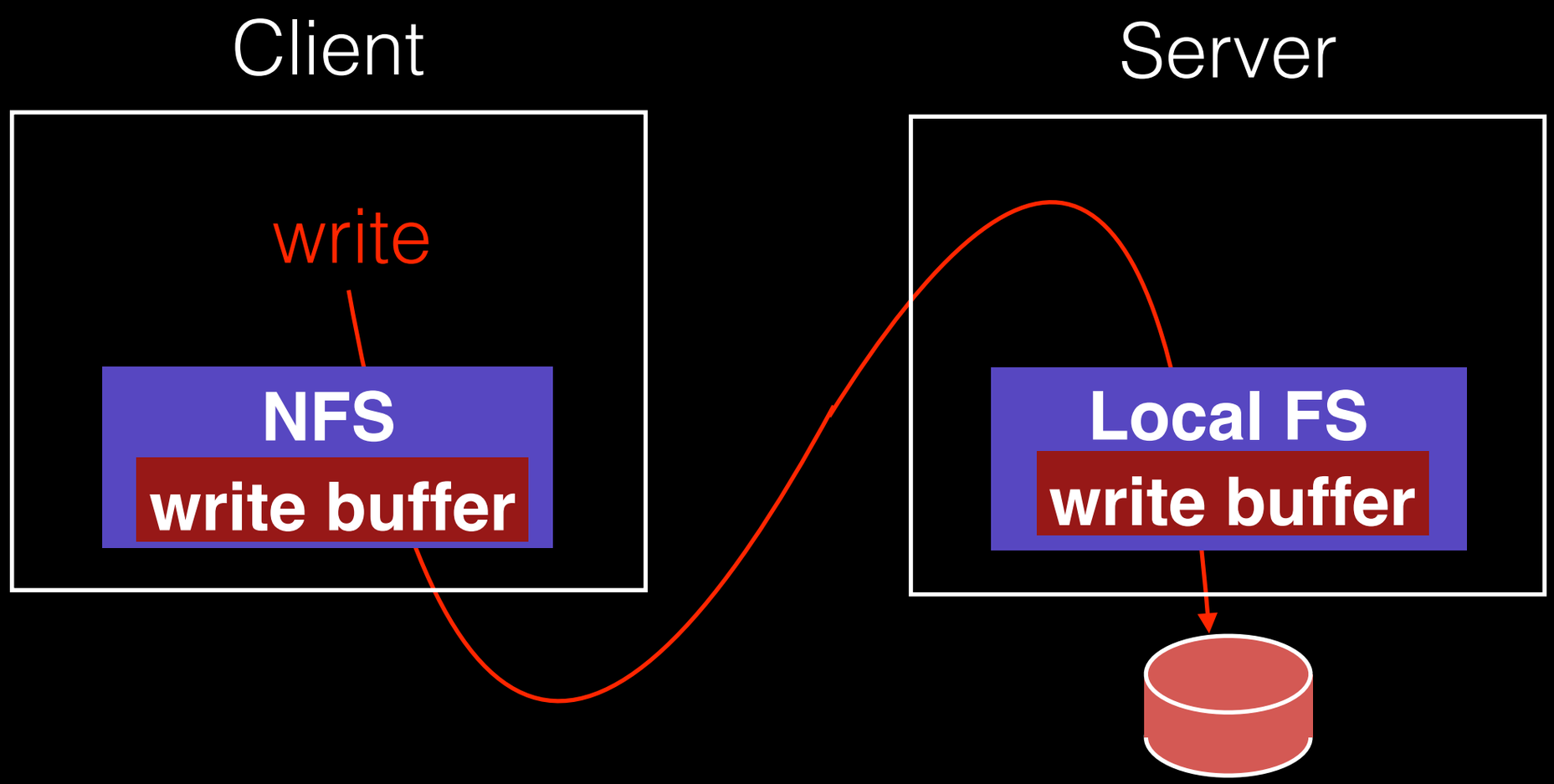
Write Buffers



Write Buffers

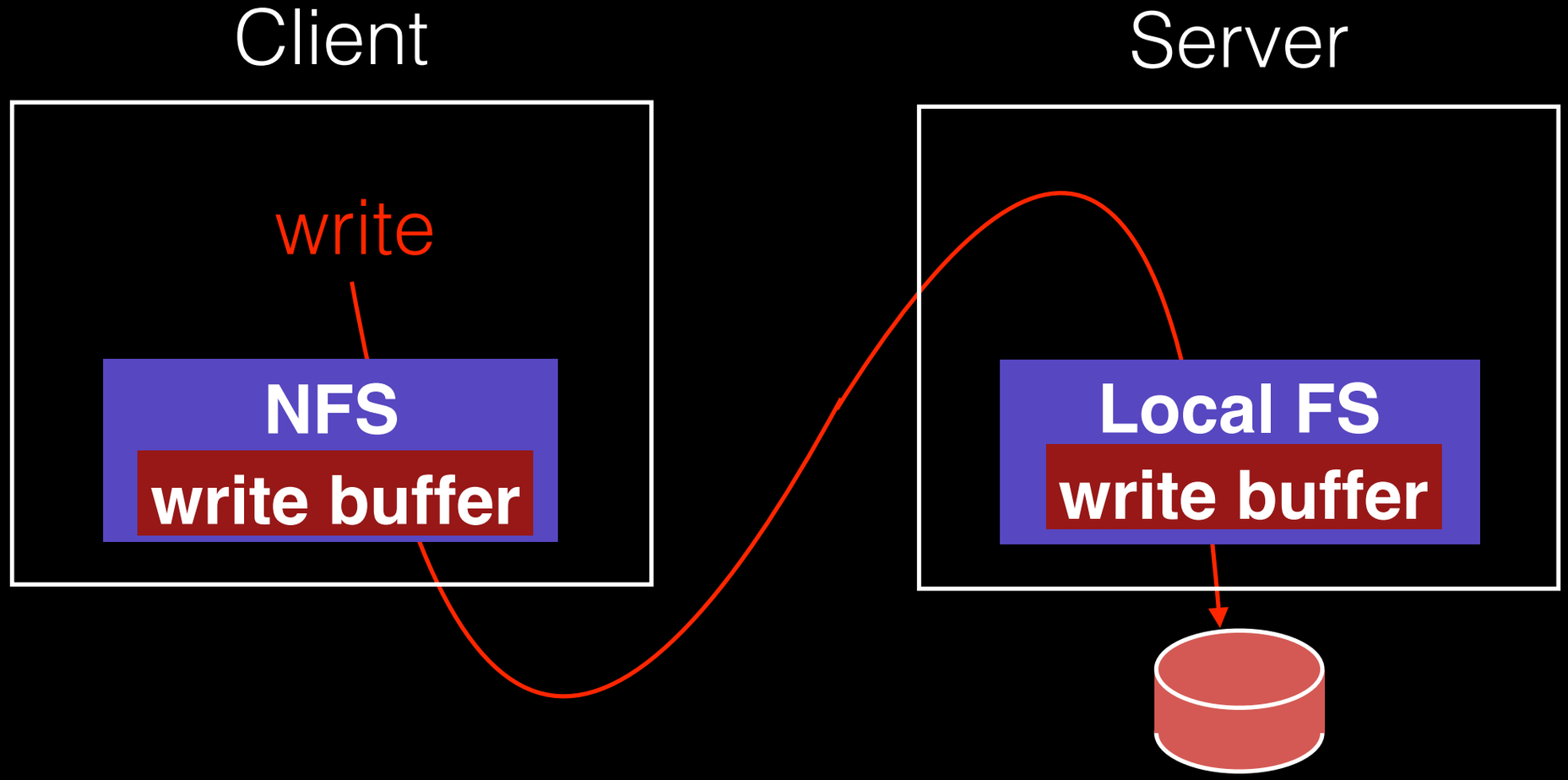


Write Buffers



Write Buffers

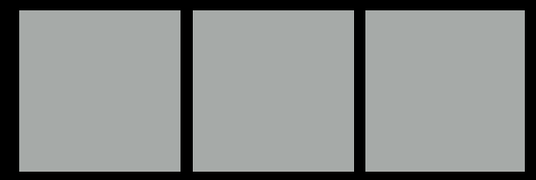
what if server crashes?



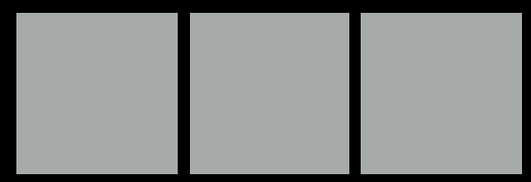
Server Write Buffer Lost

client:

server mem:



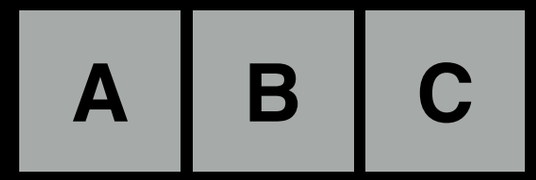
server disk:

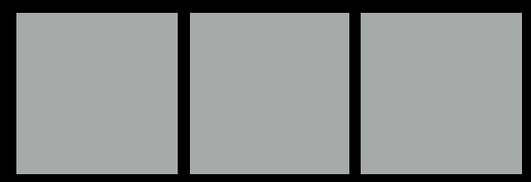


Server Write Buffer Lost

```
client:

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

server mem: 

server 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 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 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 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**

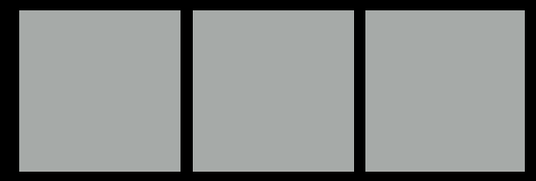
server disk: **X** **B** **C**

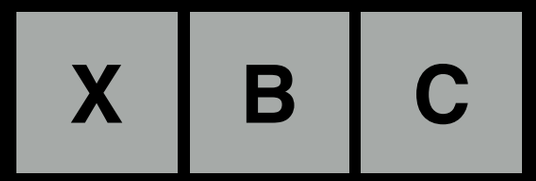
Server Write Buffer Lost

```
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: 

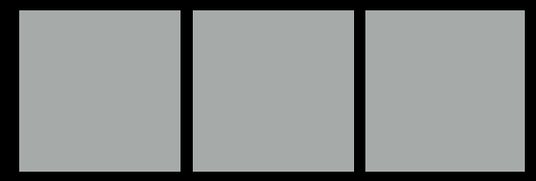
crash!

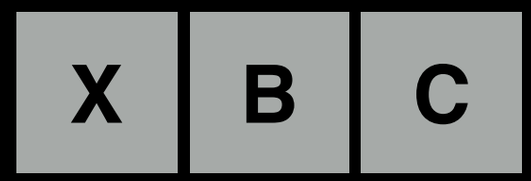
Server Write Buffer Lost

```
client:

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

 write X to 0
 write Y to 1
```

server mem: 

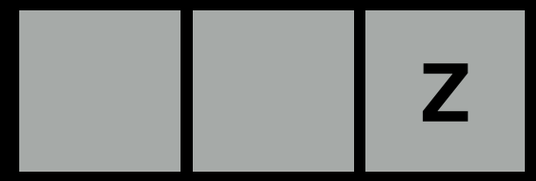
server disk: 

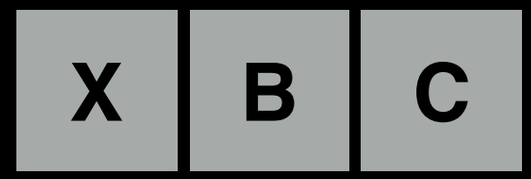
Server Write Buffer Lost

```
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: 

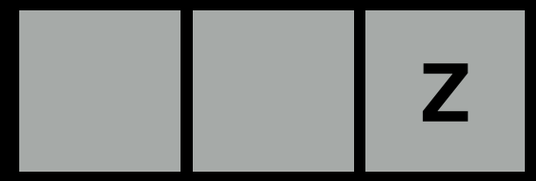
server disk: 

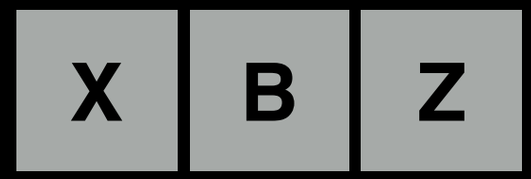
Server Write Buffer Lost

```
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: 

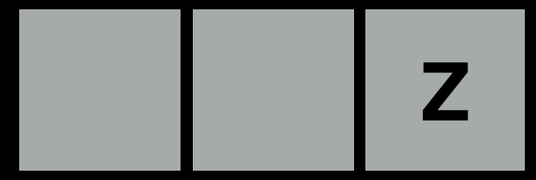
server disk: 

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
```

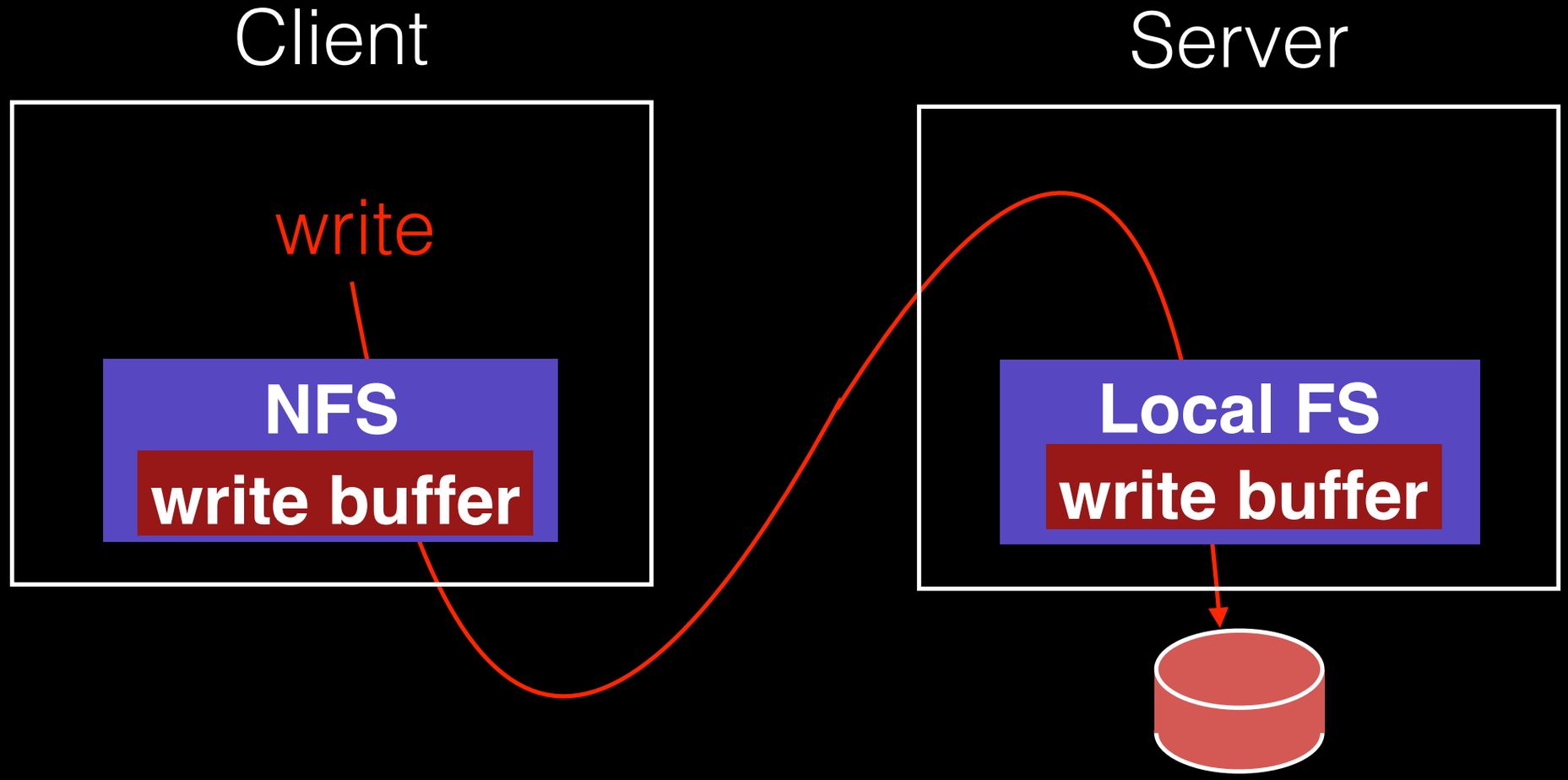
server mem: 

server disk:

no write failed, but
disk state is weird

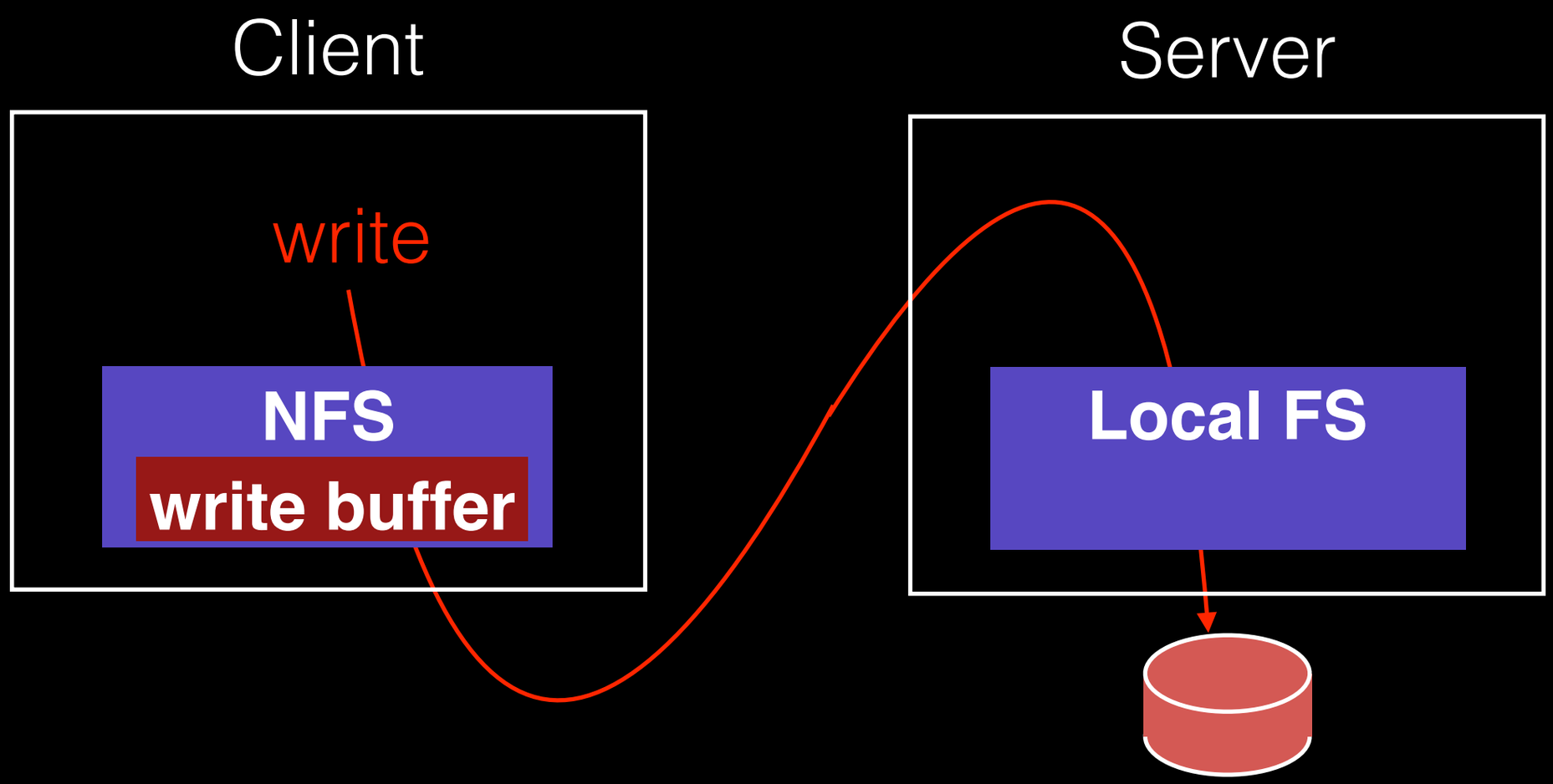
Write Buffers

what if server crashes?



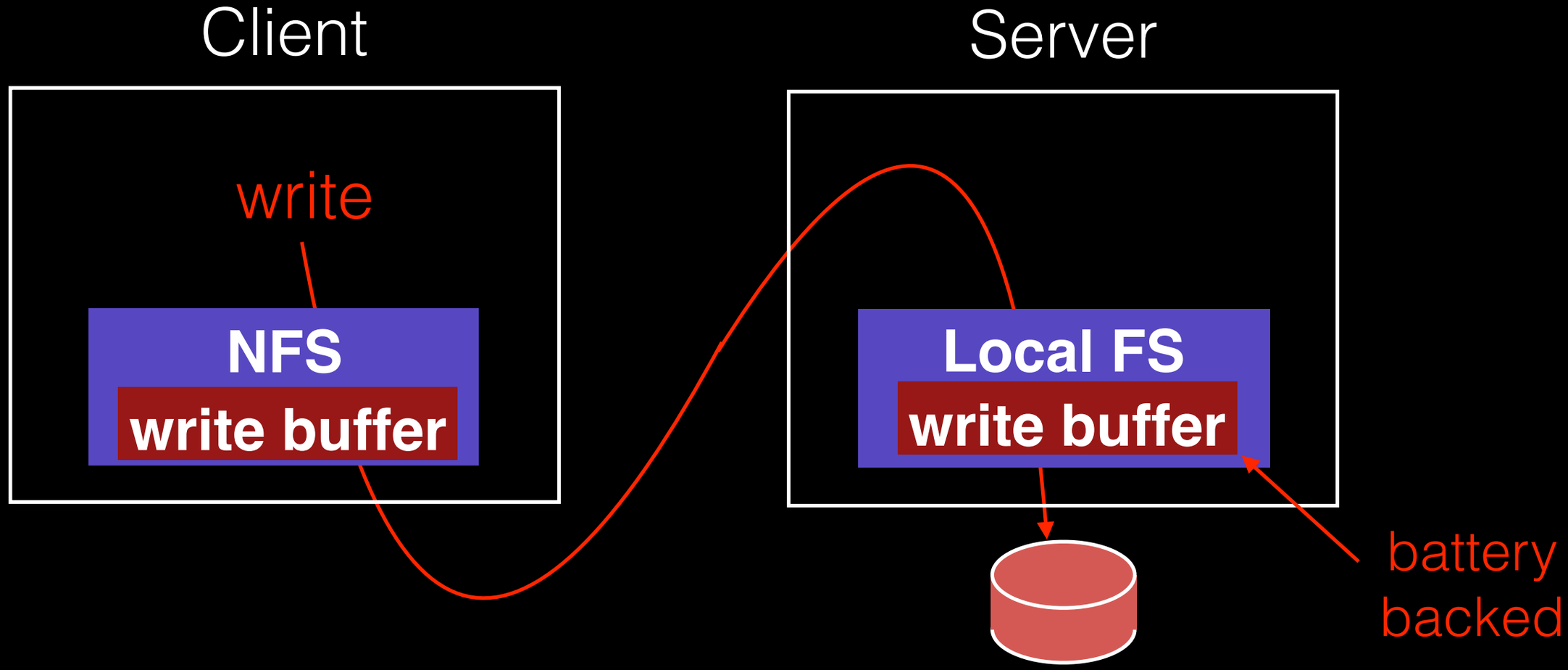
Write Buffers

1. don’t use server write buffer



Write Buffers

2. use persistent write buffer



Overview

Architecture

Network API

Write Buffering

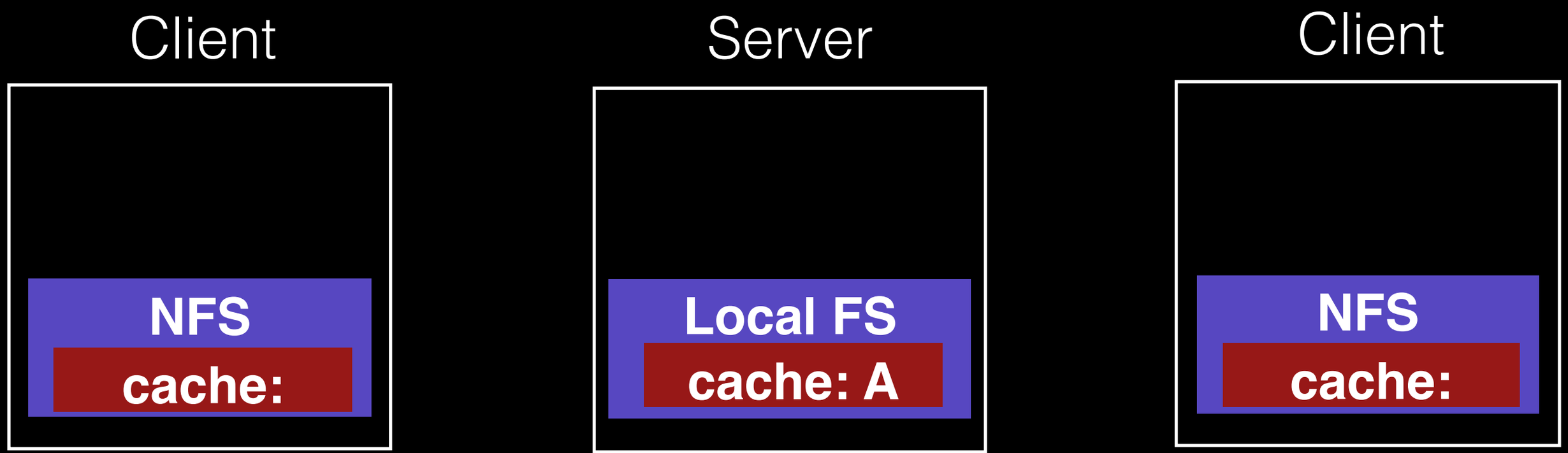
Cache

Cache

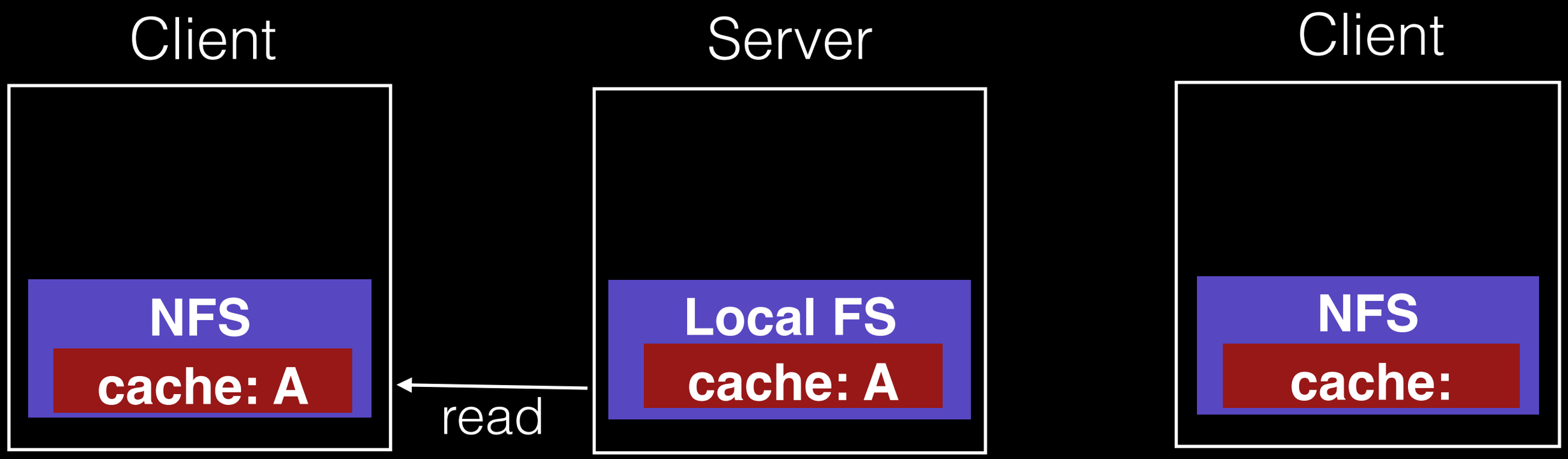
We can cache data in three places:
- server memory
- client disk
- client memory

How to make sure all versions are in sync?

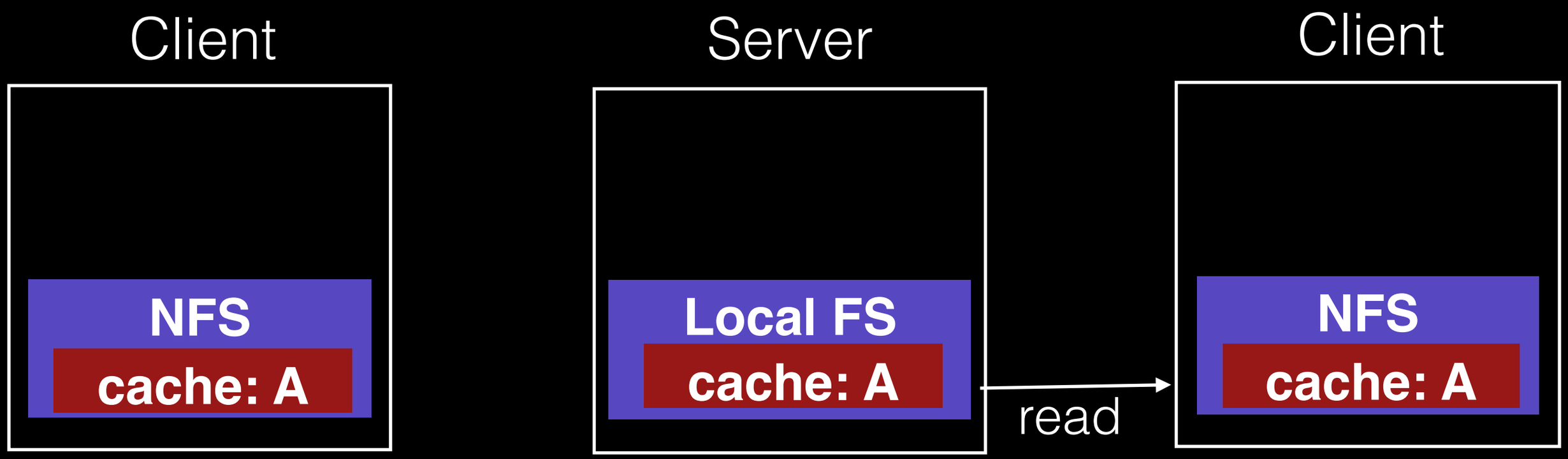
Cache



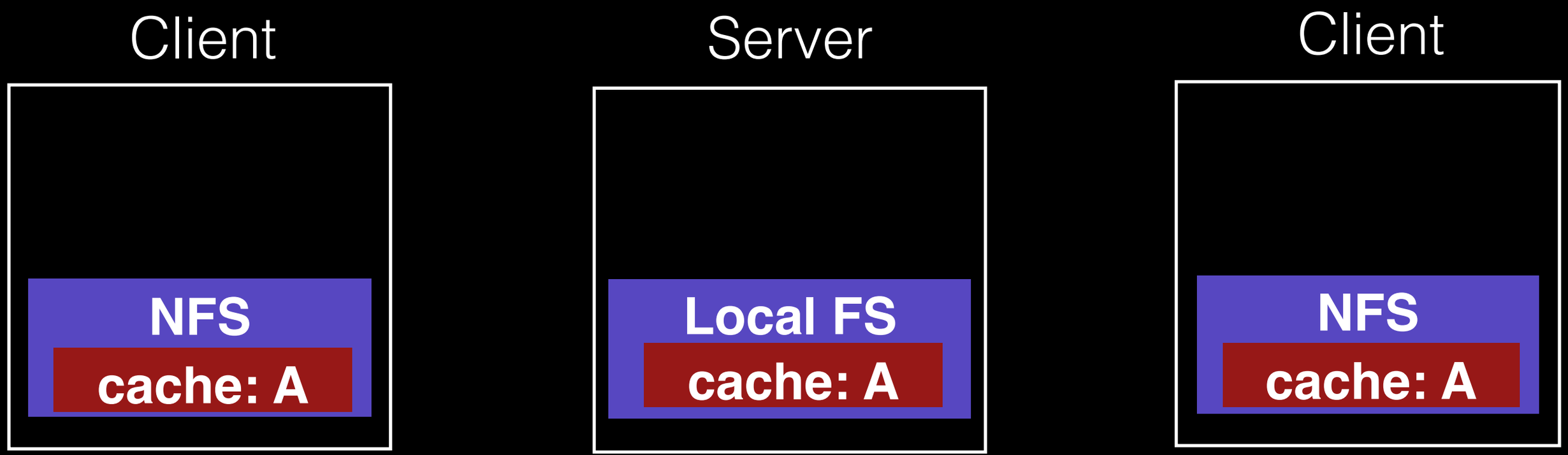
Cache


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

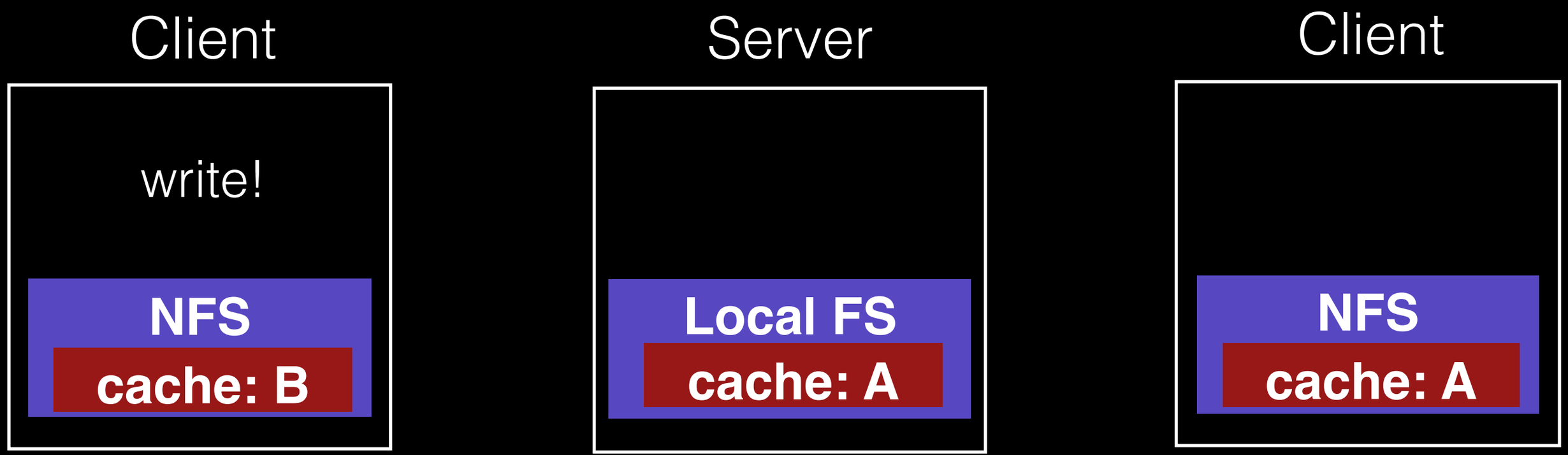
Cache


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

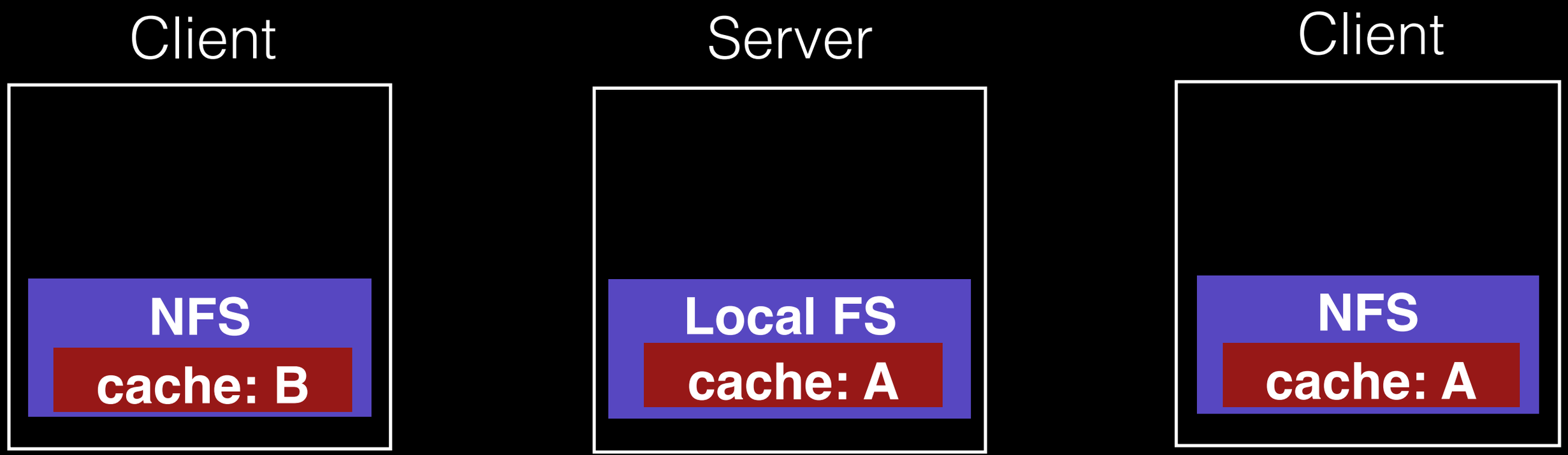
Cache



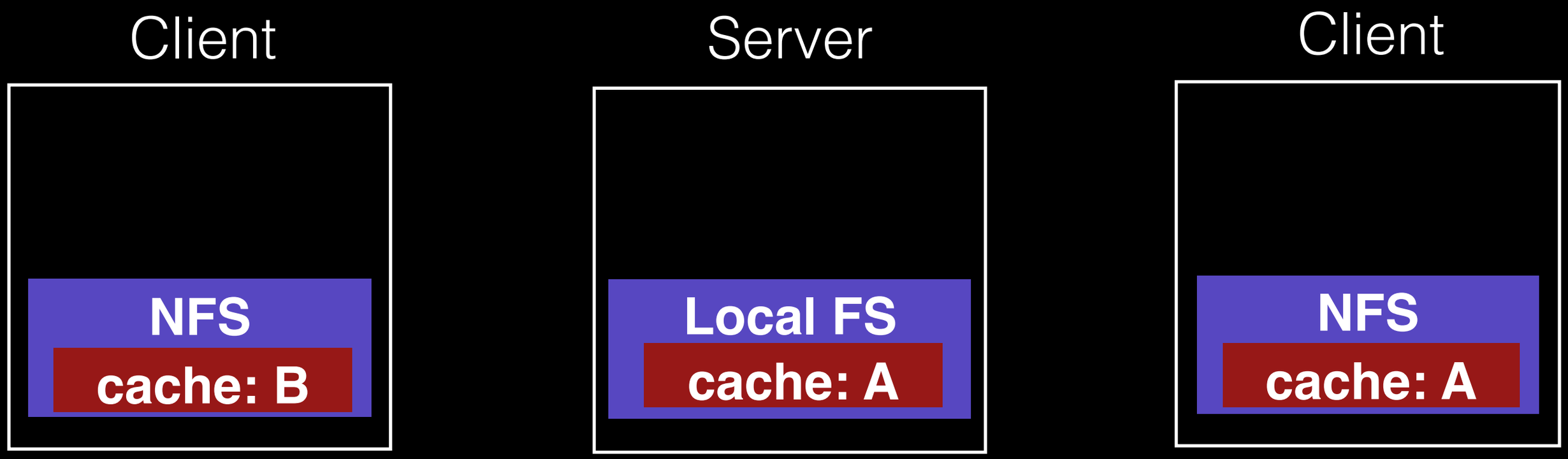
Cache


Client
write!
NFS
cache: B
Server
Local FS
cache: A
Client
NFS
cache: A

Cache

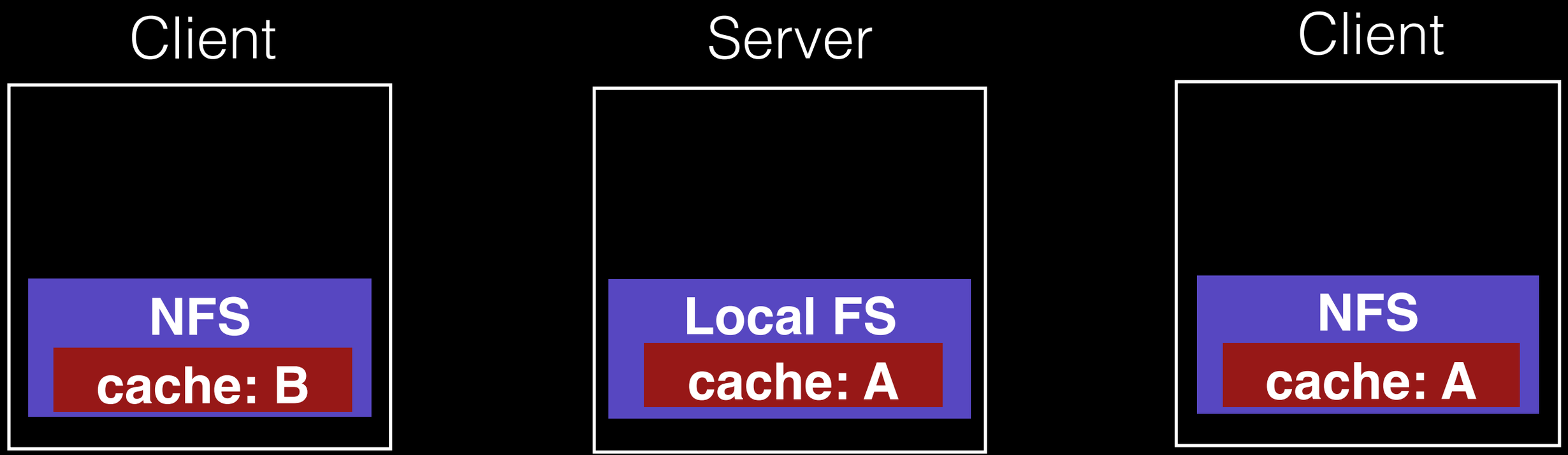


Cache

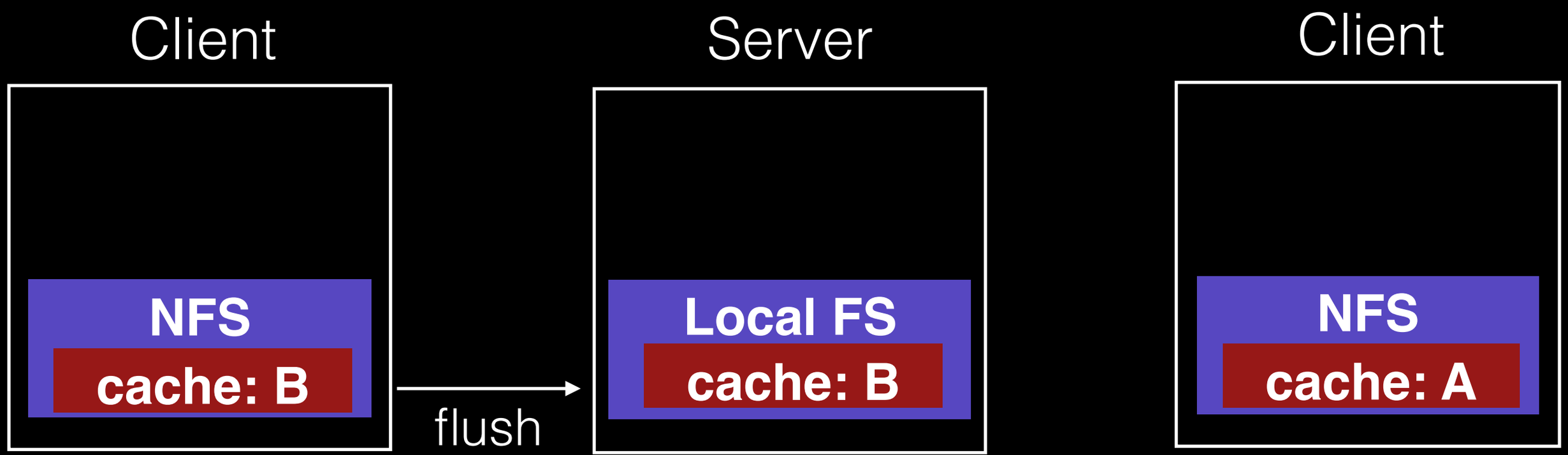


“Update Visibility” problem: server doesn’t have latest.

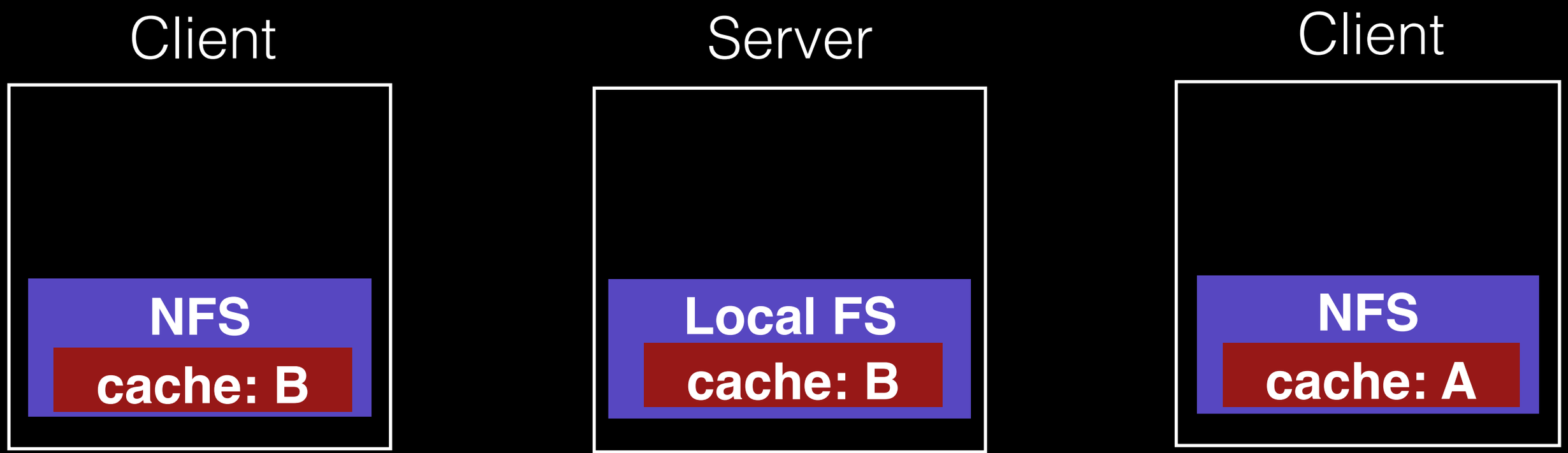
Cache



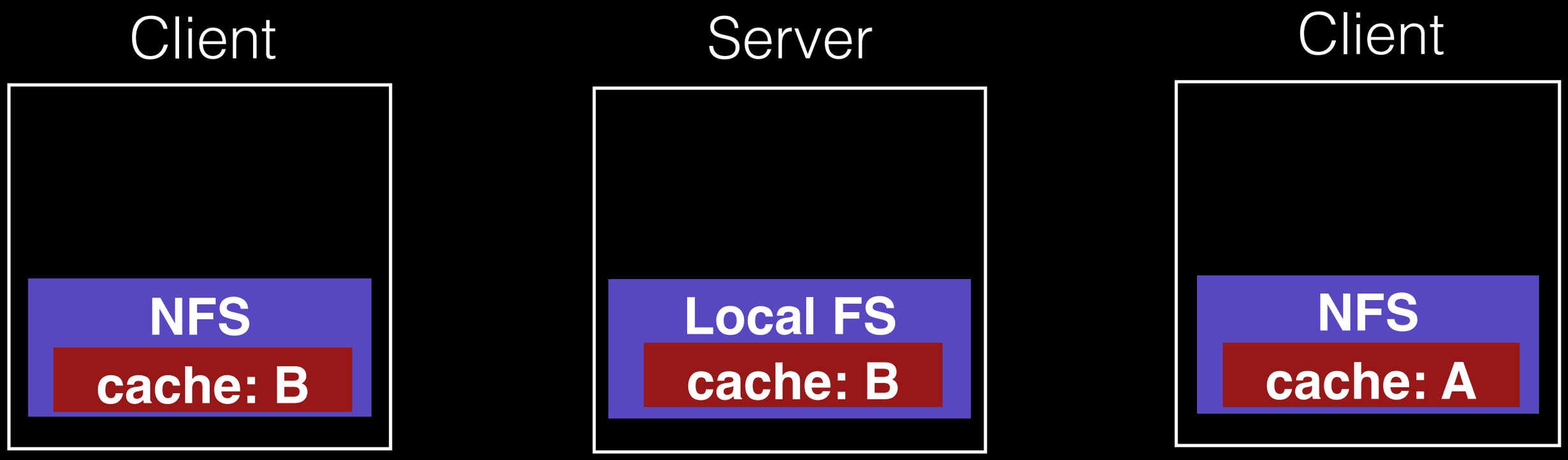
Cache


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

Cache

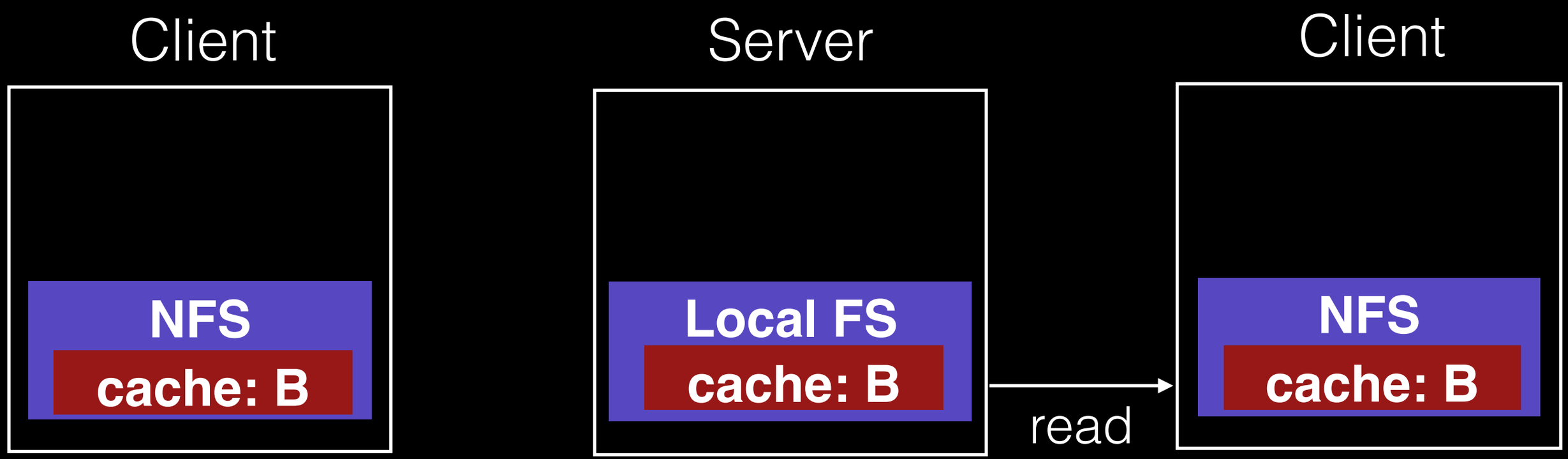


Cache



“Stale Cache” problem: client doesn’t have latest.

Cache


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

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.

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.

NFS solution: clients recheck if cache is current before using it.

Stale Cache Solution

Cache metadata records when data was fetched.

Before it is used, client does a STAT request to server.
- get's last modified timestamp
- compare to cache
- refetch 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?

Also 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 and write buffering is harder in distributed systems, especially with crashes.

Announcements

Wednesday lecture
- cancelled

Office hours
- today at noon to 1pm, in lab

Happy Thanksgiving!