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*

Definition:
More than 1 machine working together to solve a problem

Examples:

- client/server: web server and web client
- cluster: page rank computation

Other courses:

- **CS 640**: Networking
- **CS 739**: Distributed Systems

Why Go Distributed?

More computing power

More storage capacity

Fault tolerance

Data sharing

New Challenges

System failure: need to worry about partial failure

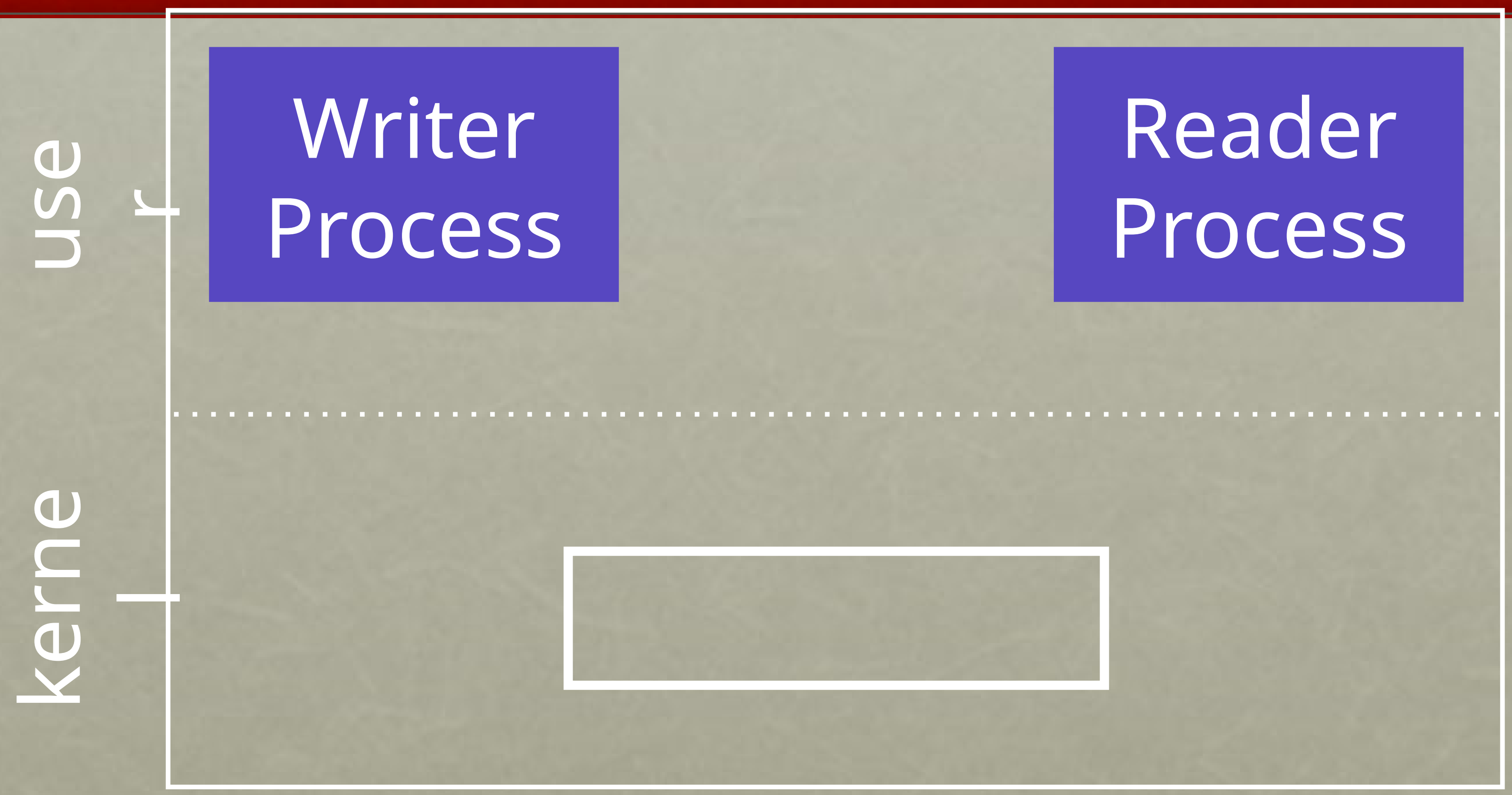
Communication failure: links unreliable

- bit errors
- packet loss
- node/link failure

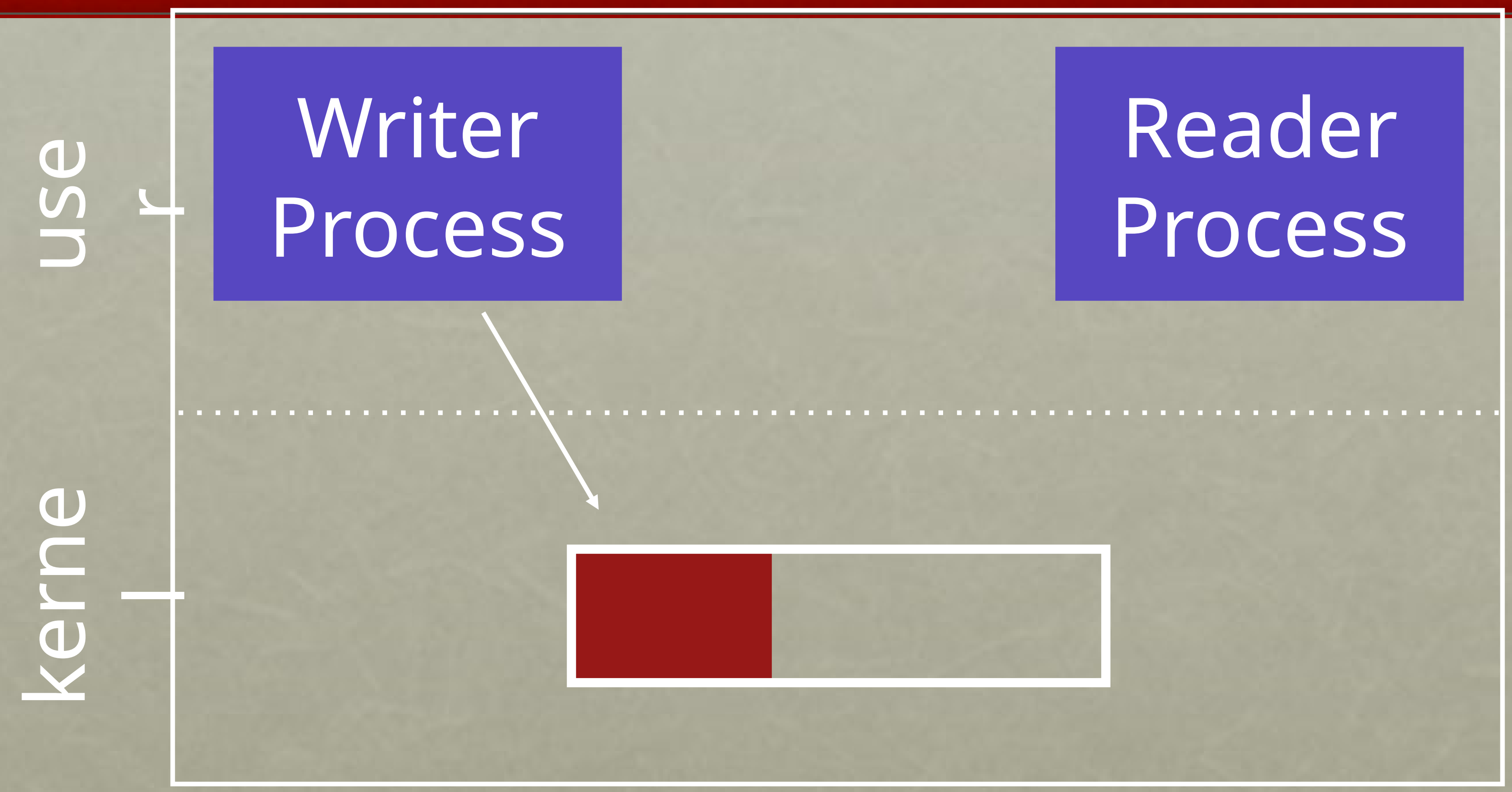
Motivation example:

Why are network sockets less reliable than pipes?

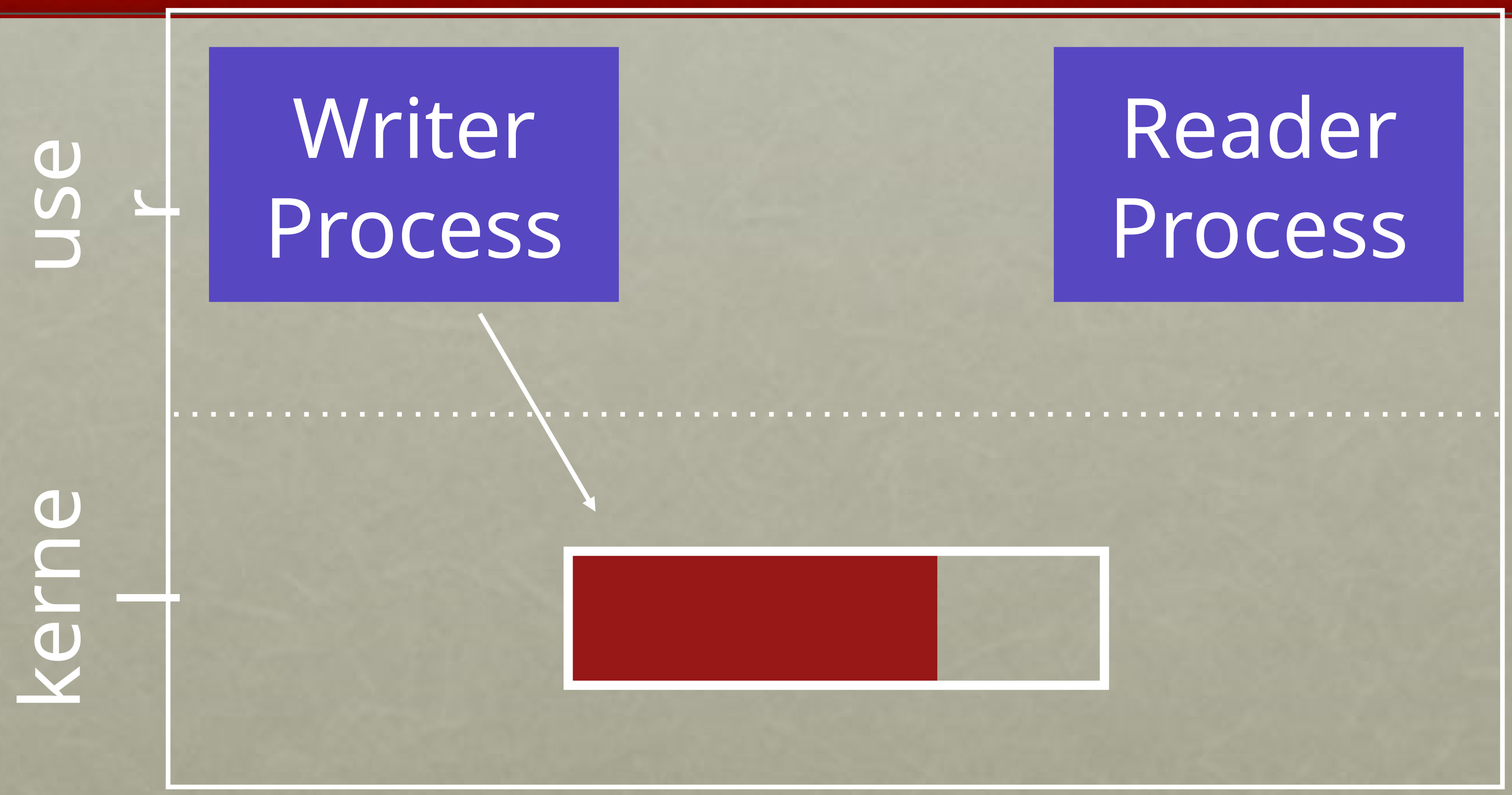
Pipe



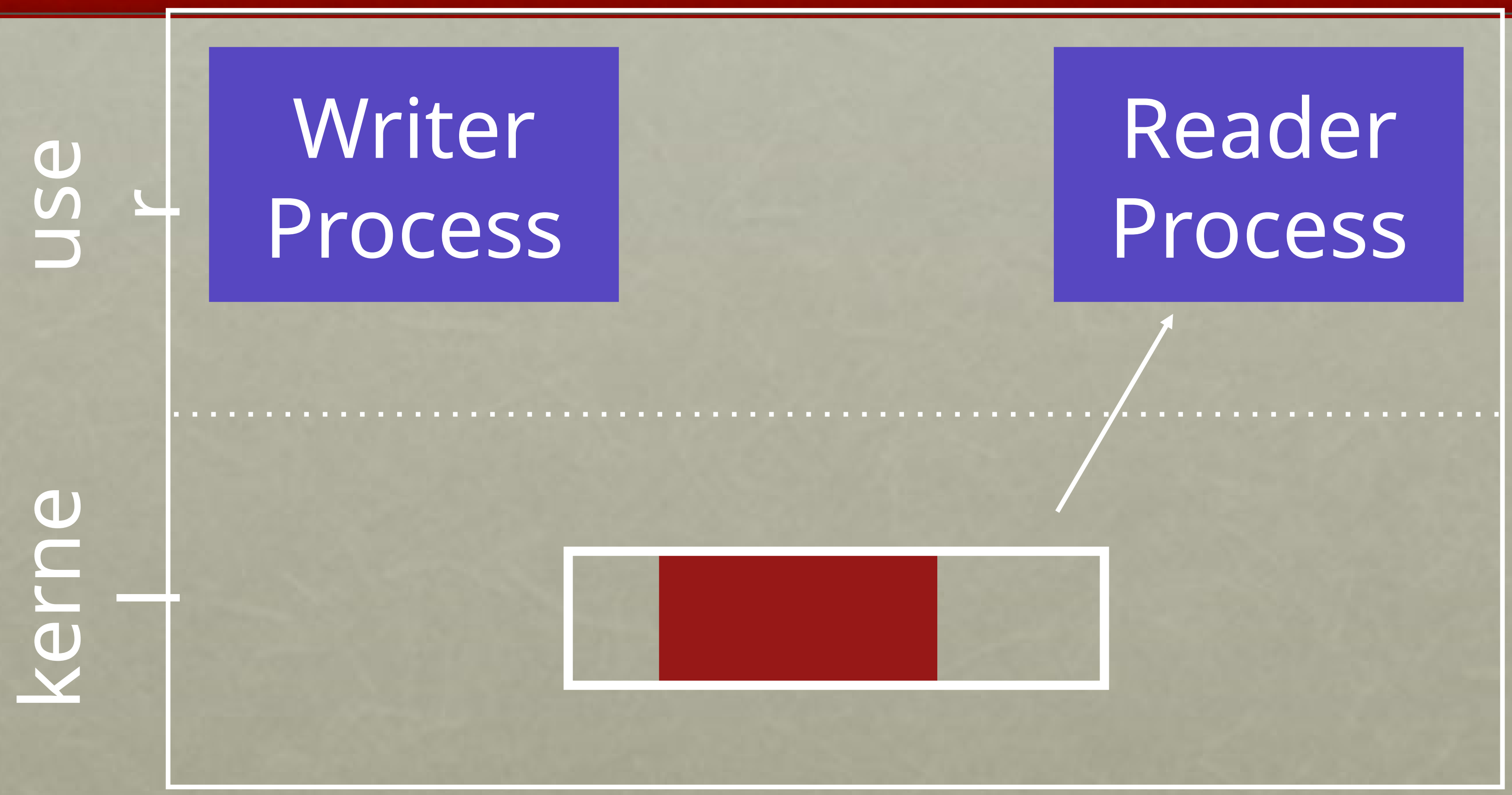
Pipe



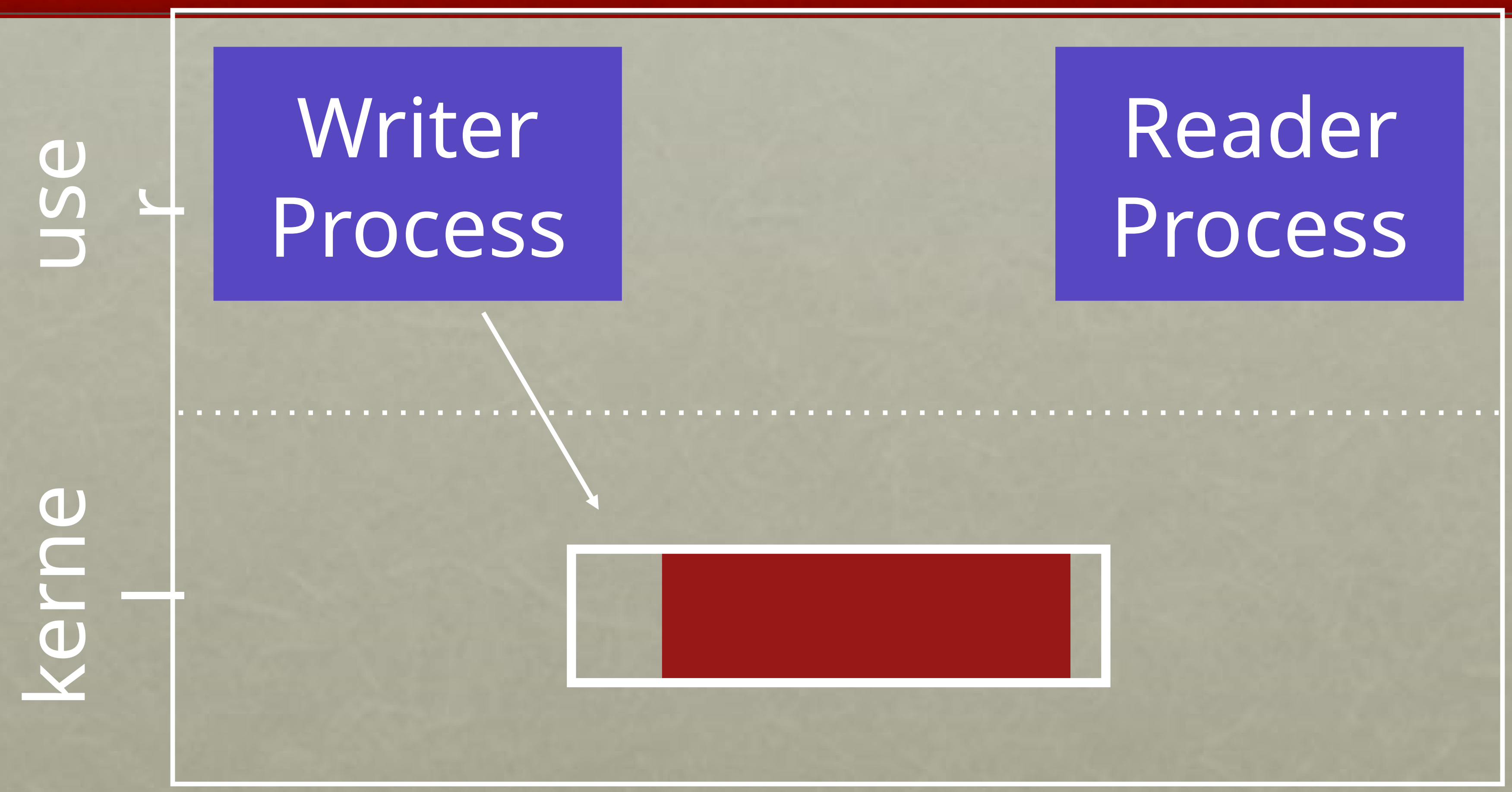
Pipe



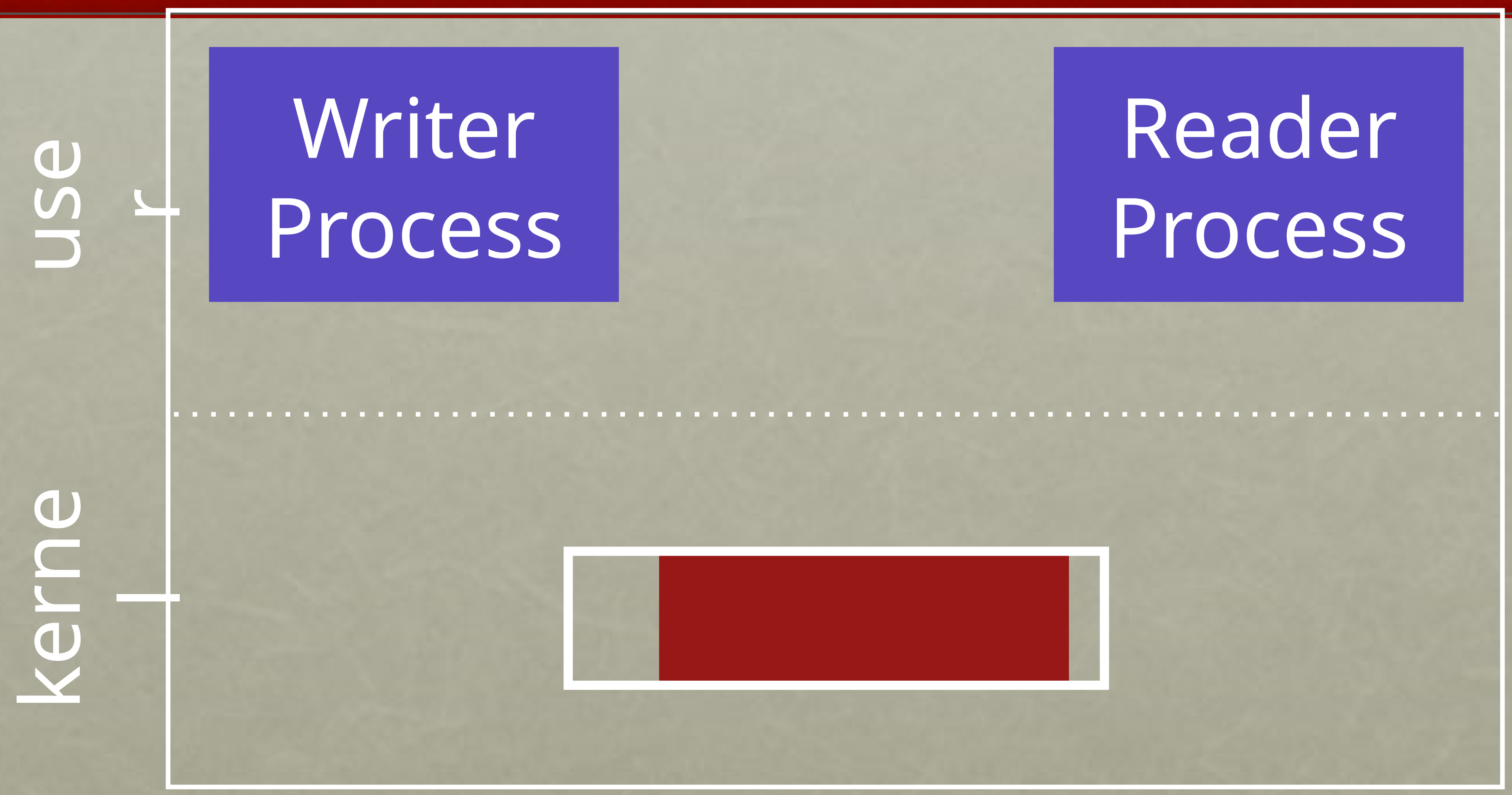
Pipe



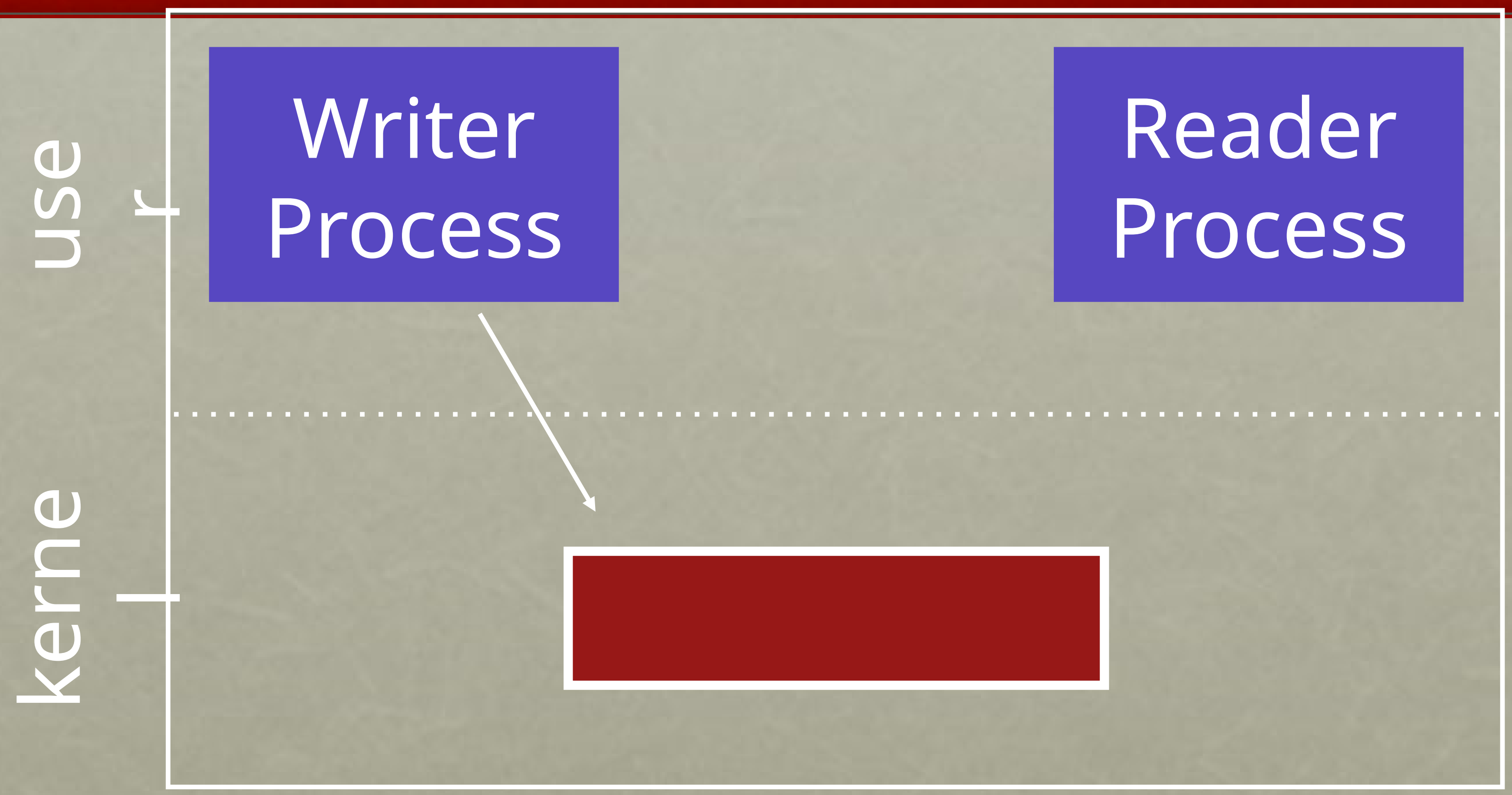
Pipe



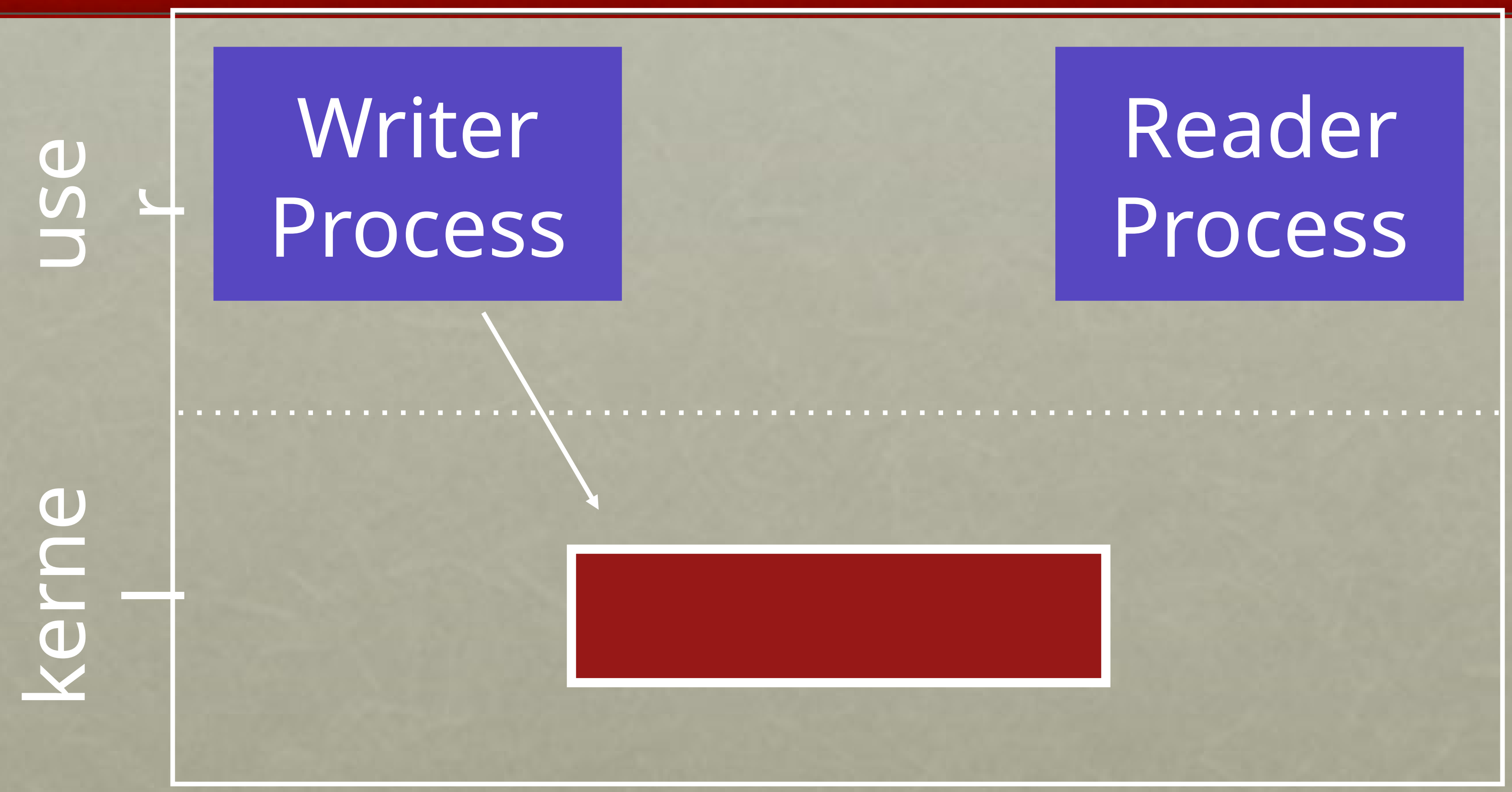
Pipe



Pipe

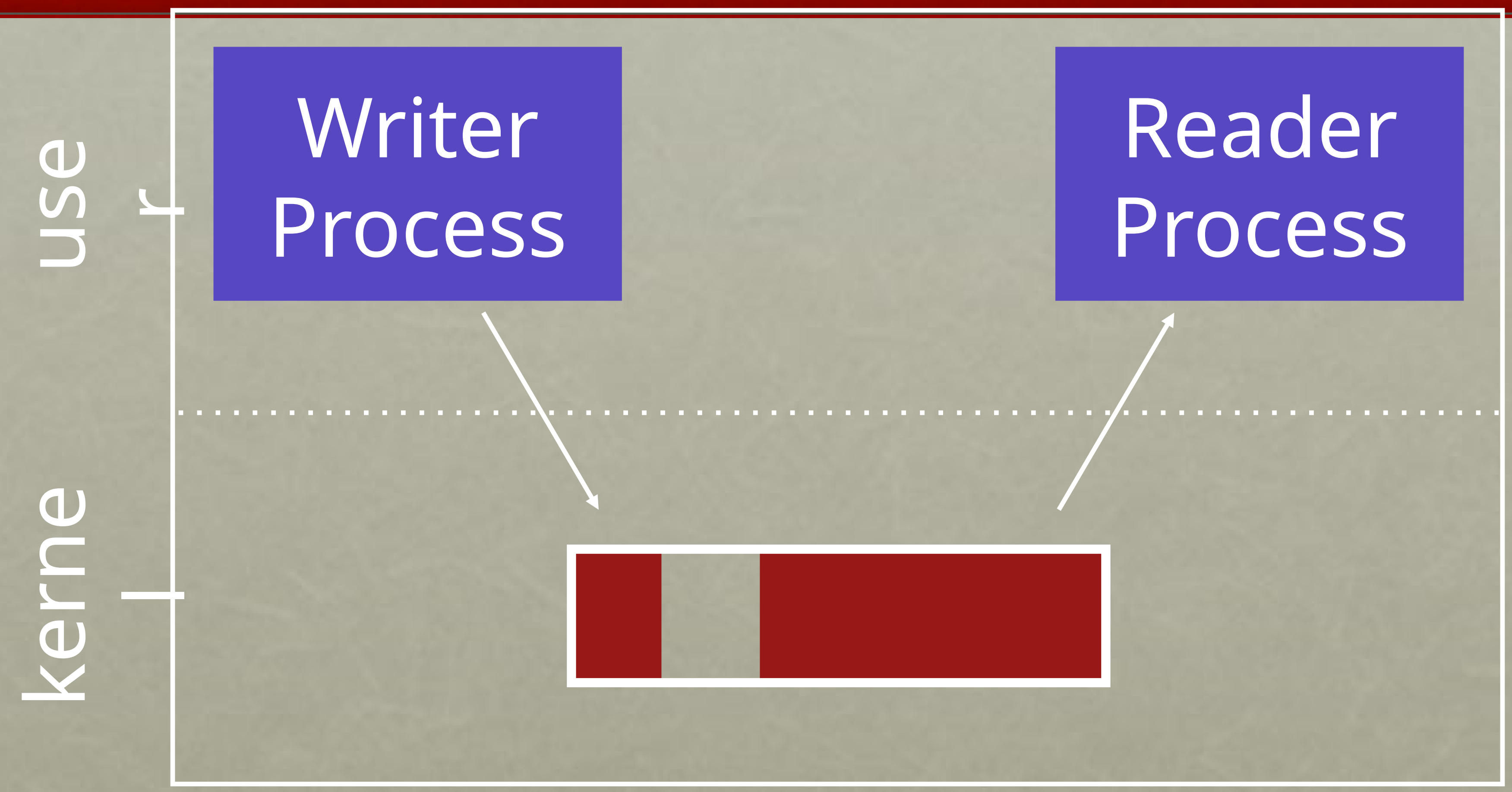


Pipe



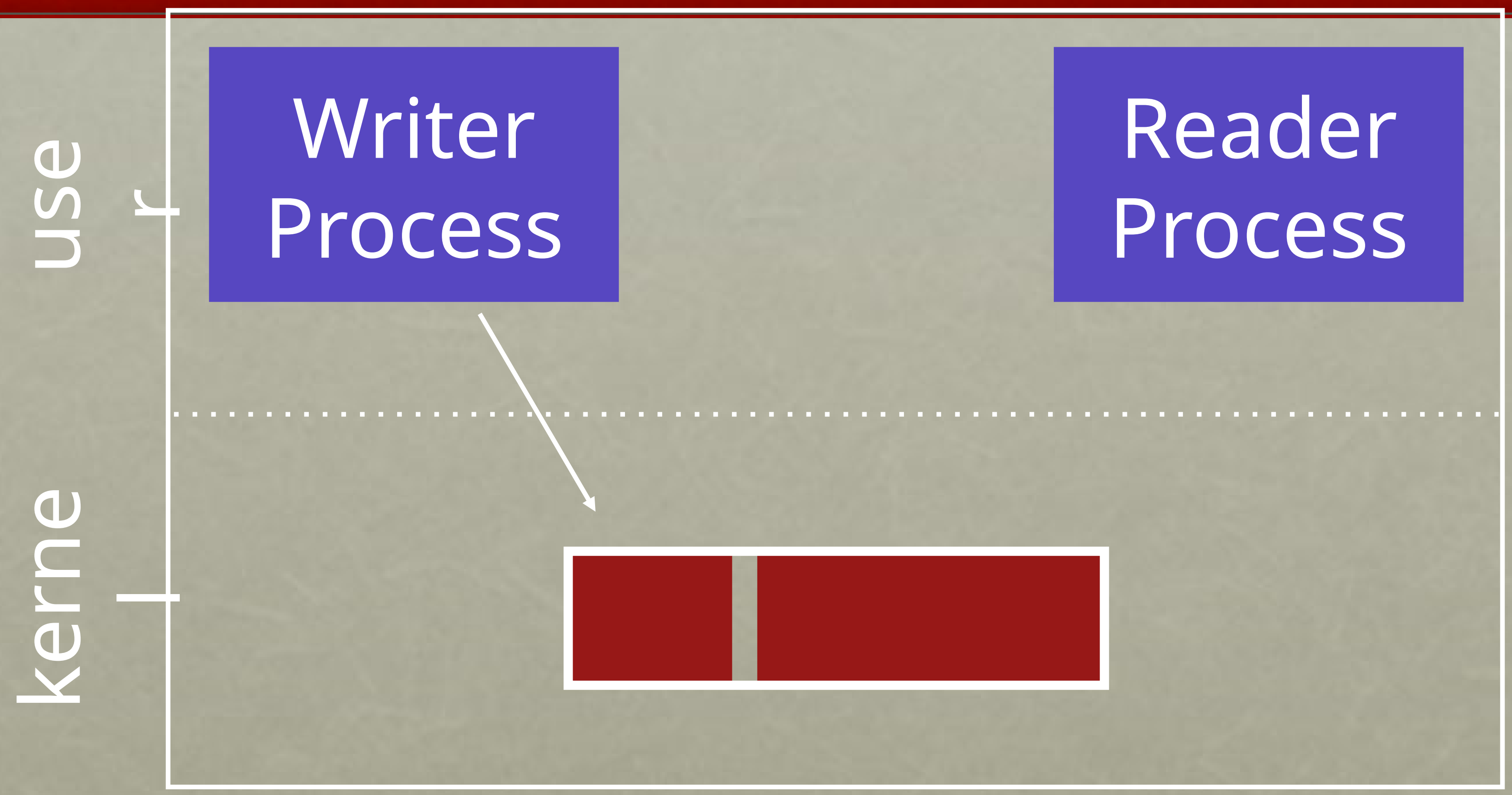
write waits for space

Pipe

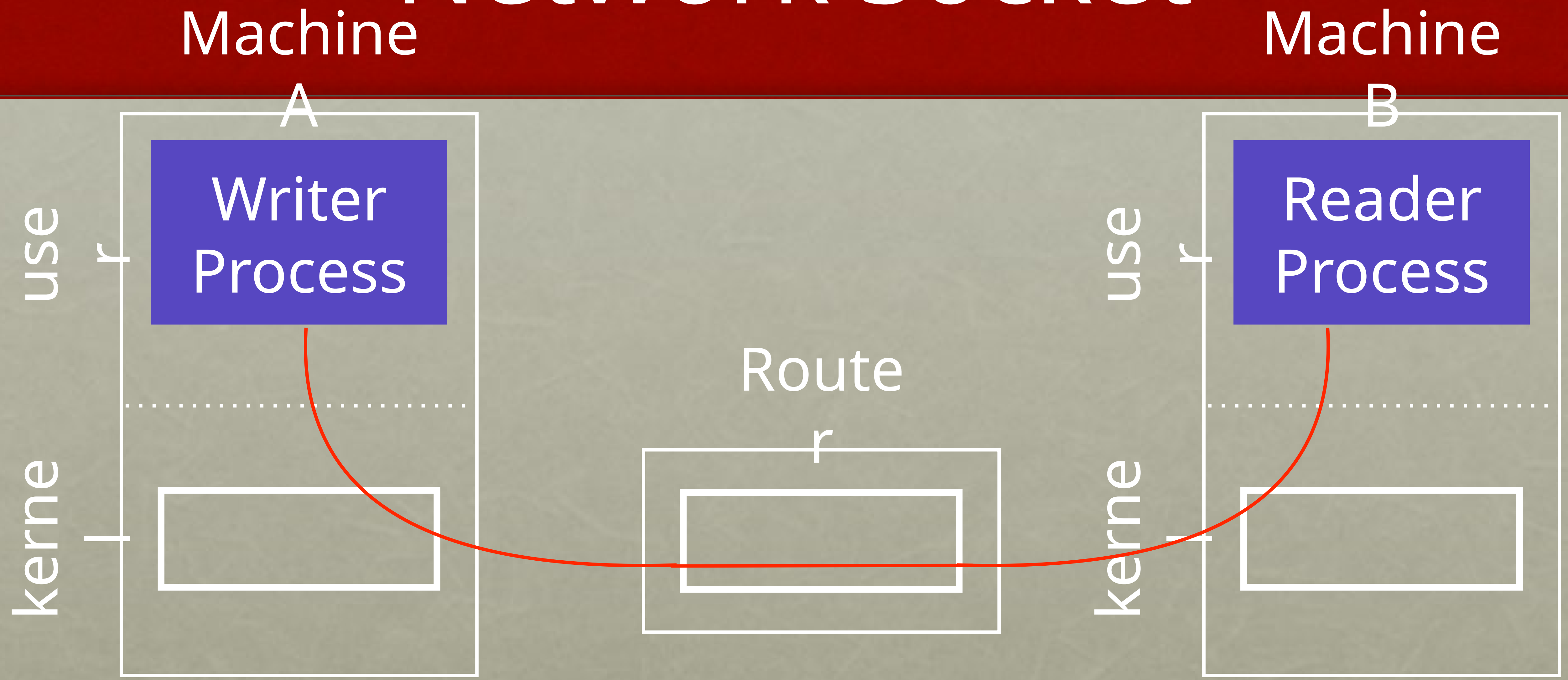


write waits for space

Pipe

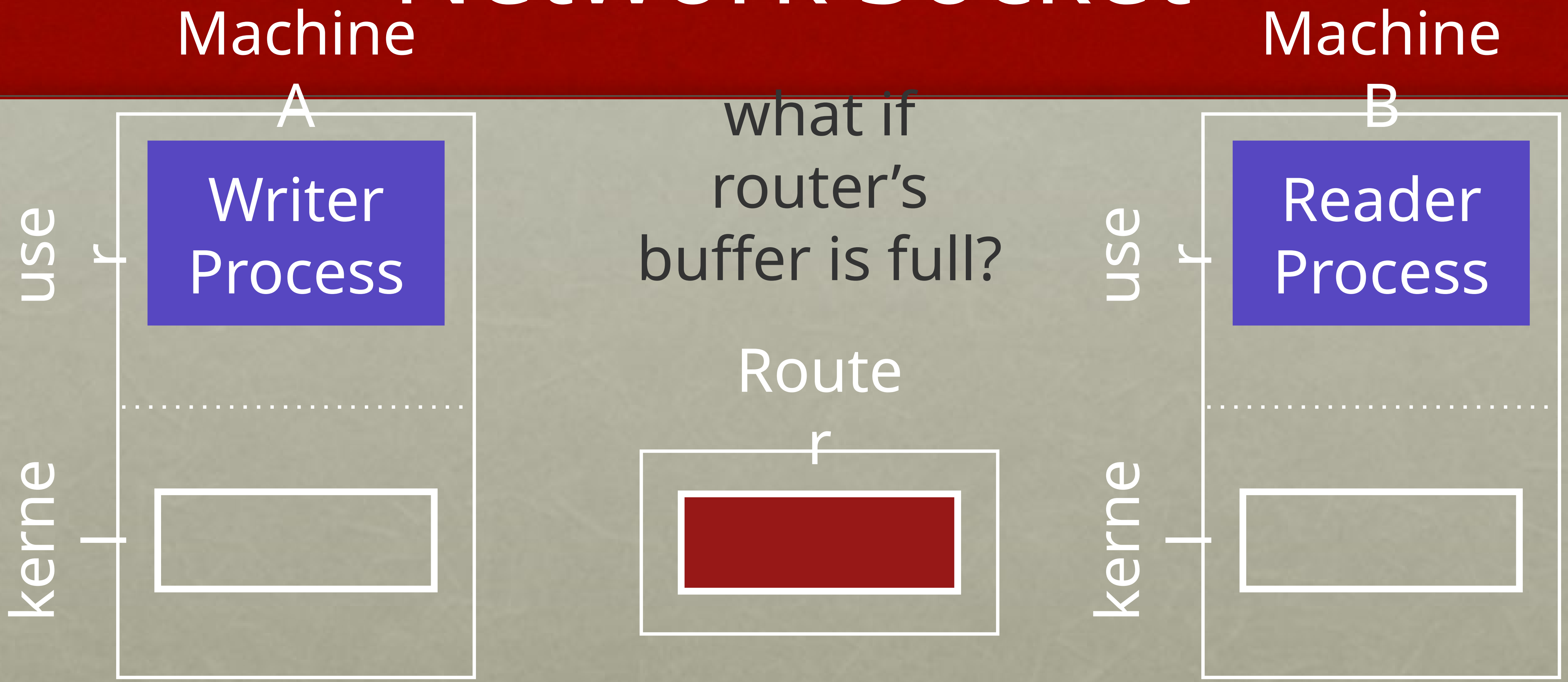


Network Socket

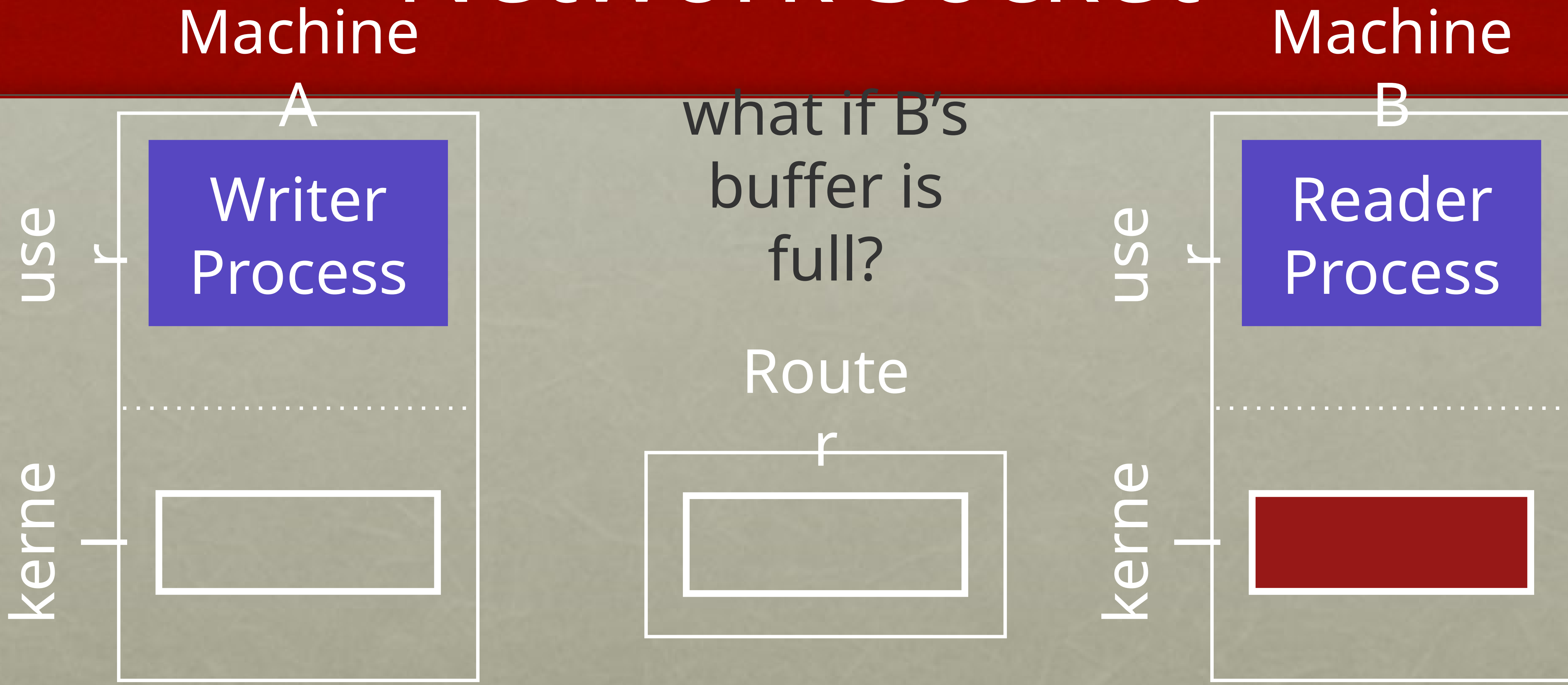


Network Socket

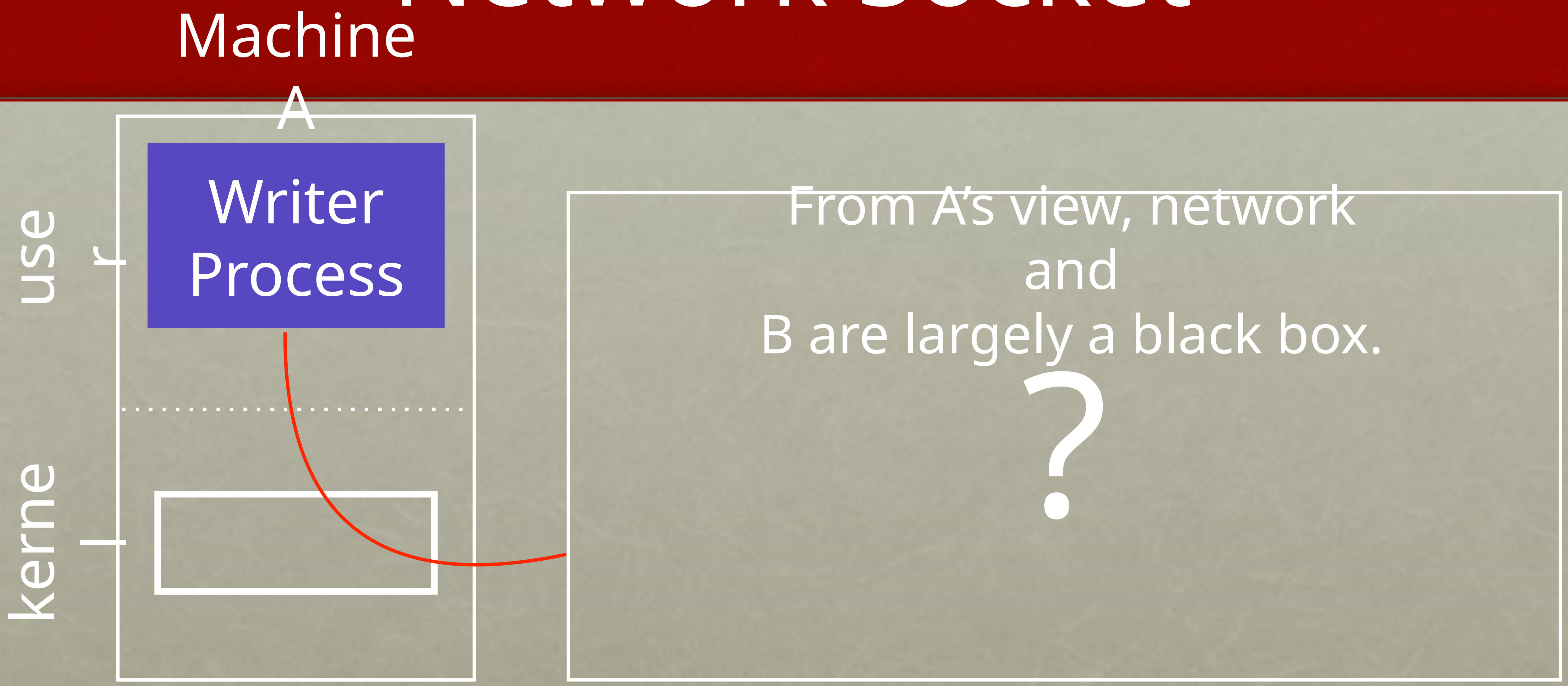
what if router's buffer is full?



Network Socket



Network Socket



Communication Overview

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC

Raw Messages: UDP

UDP : User Datagram Protocol

API:

- reads and writes over socket file descriptors
- messages sent from/to ports to target a process on machine

Provide minimal reliability features:

- messages may be lost
- messages may be reordered
- messages may be duplicated
- only protection: checksums to ensure data not corrupted

Raw Messages: UDP

Advantages

- Lightweight
- Some applications make better reliability decisions themselves (e.g., video conferencing programs)

Disadvantages

- More difficult to write applications correctly

Reliable Messages: Layering strategy

TCP: Transmission Control Protocol

Using software, build reliable, logical connections over unreliable connections

Techniques:

- acknowledgment (ACK)

Technique #1: ACK

Sender	Receiver
[send message]	
	[recv message] [send ack]
[recv ack]	

Sender knows message was received

ACK

Sender
[send
message]

Receive
r

Sender doesn't receive ACK…

What to do?

Technique #2: Timeout

Sender	Receiver
[send message] [start timer]	
... waiting for ack ...	
[timer goes off] [send message]	[recv message]
	[send ack]
[recv ack]	

Lost ACK: Issue 1

How long to wait?

Too long?

- System feels unresponsive

Too short?

- Messages needlessly re-sent
- Messages may have been dropped due to overloaded server. Resending makes overload worse!

Lost Ack: Issue 1

How long to wait?

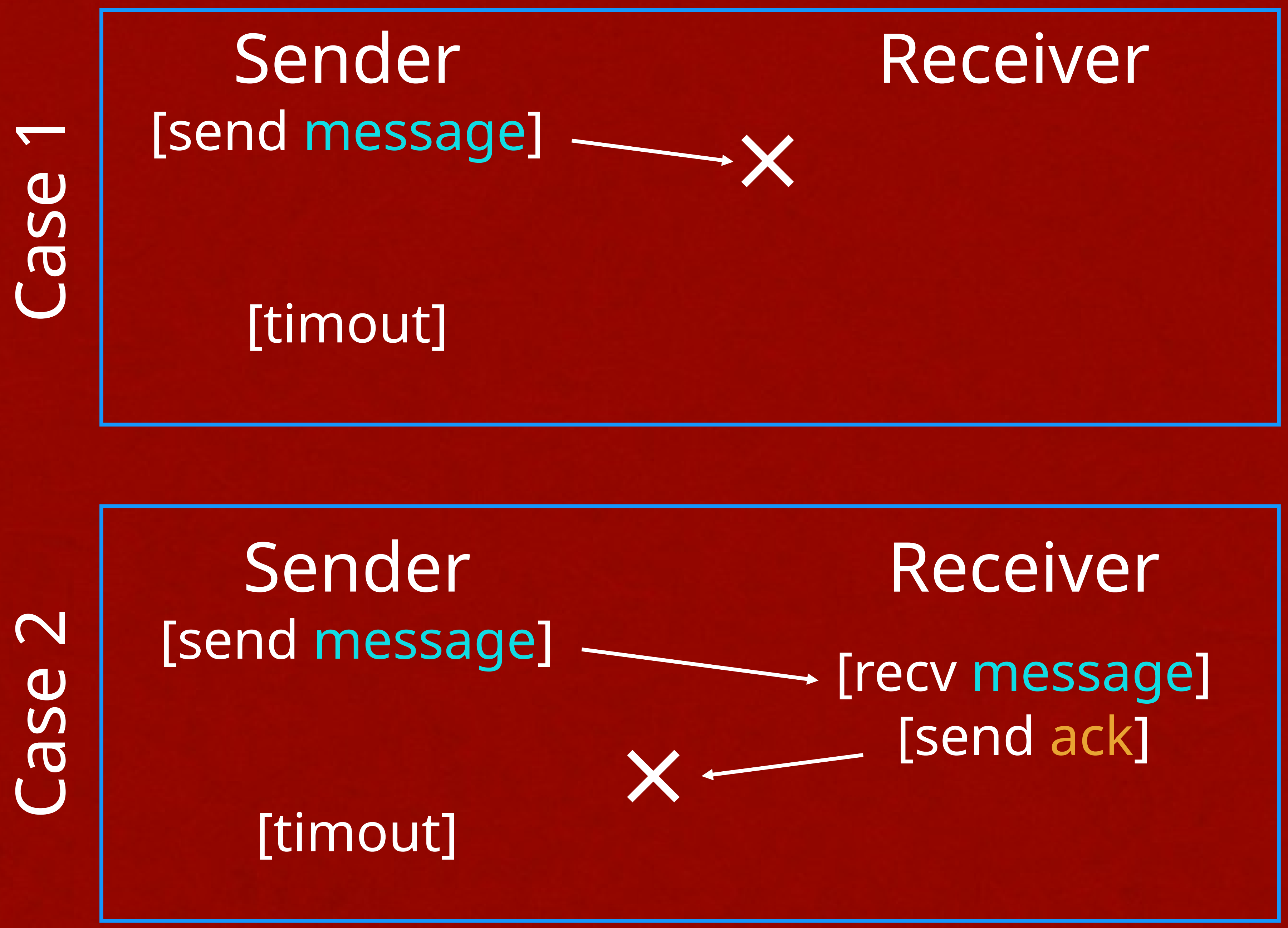
One strategy: be adaptive

Adjust time based on how long acks usually take

For each missing ack, wait longer between retries

Lost Ack: Issue 2

What does a lost ack really mean?



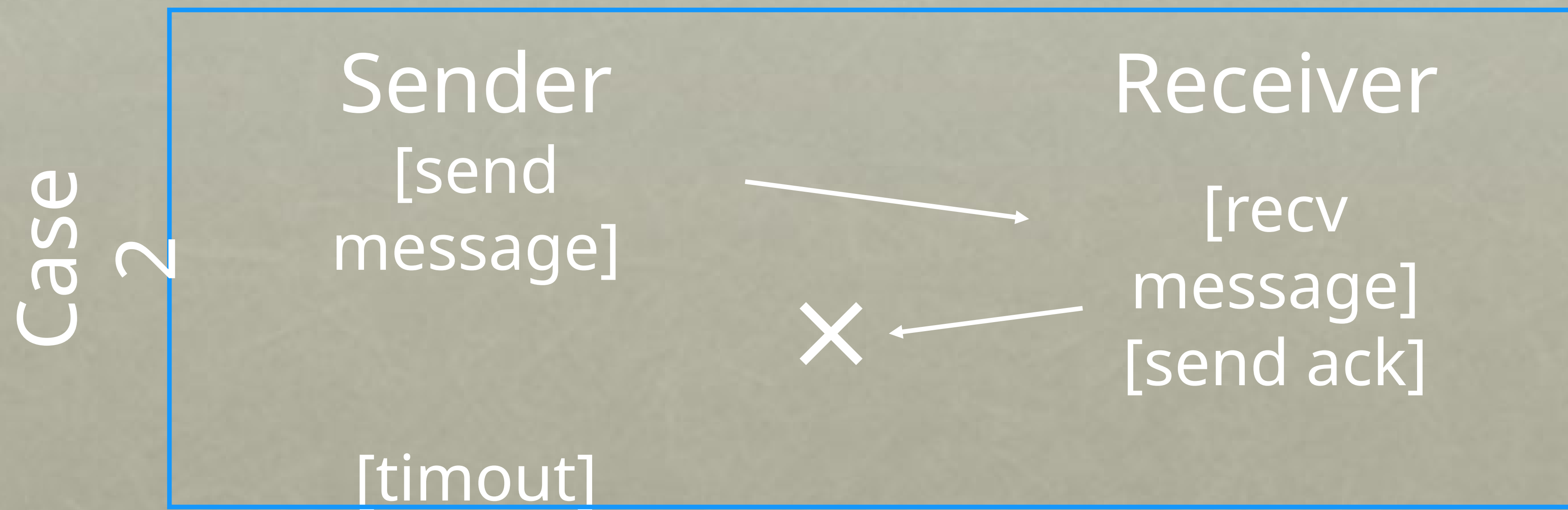
Lost ACK:
How can
sender
tell between
these
two cases?

ACK: message received exactly once

No ACK: message may or may not have been received

What if message is command to increment counter?

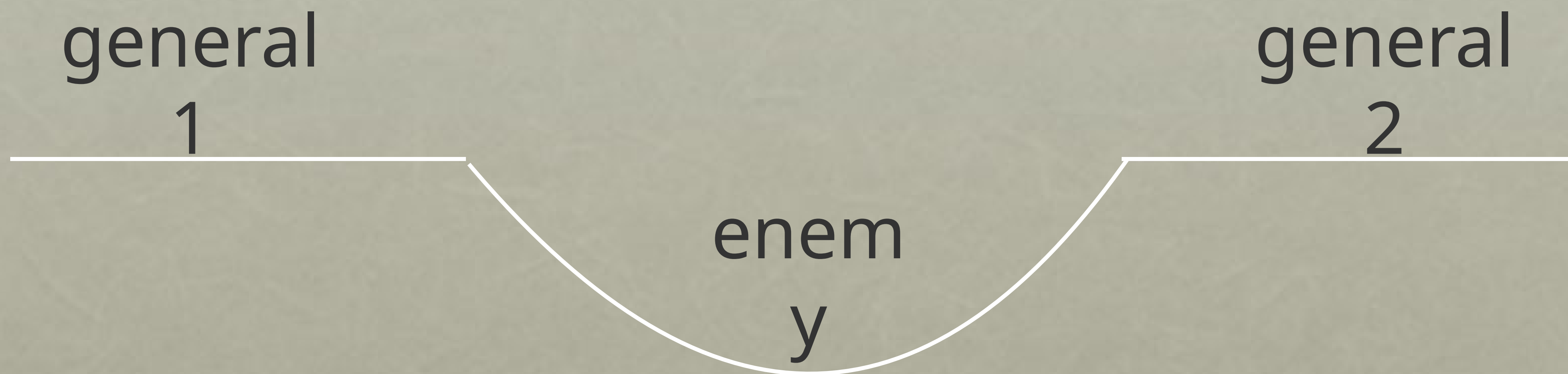
Proposed Solution



Proposal:
Sender could send an AckAck so receiver knows whether to retry sending an Ack

Sound good?

Aside:
Two Generals' Problem



Suppose generals agree after N messages

Did the arrival of the N'th message change decision?

- if yes: then what if the N'th message had been lost?

- if no: then why bother sending N messages?

Reliable Messages: Layering Strategy

Using software, build reliable, logical connections over unreliable connections

Techniques:

- acknowledgment
- timeout
- remember sent messages

Technique #3: Receiver Remembers Messages

Sender	Receiver
[send message]	[recv message]
	[send ack]
[timout] [send message]	[ignore message]
	[send ack]
[recv ack]	

how does receiver know to ignore?

Solutions

Solution 1: remember every message ever received

Solution 2: sequence numbers

- senders gives each message an increasing unique seq number
- receiver knows it has seen all messages before N
- receiver remembers messages received after N

Suppose message K is received. Suppress message if:

- K < N
- Msg K is already buffered

TCP

TCP: Transmission Control Protocol

Most popular protocol based on seq nums

Buffers messages so arrive in order

Timeouts are adaptive

Communications Overview

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC

RPC

Remote **P**rocedure **C**all

What could be easier than calling a function?

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

Very common abstraction

RPC

Machine A

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

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("hello");
}

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("hello");
}

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

RPC Tools

RPC packages help with two components

(1) Runtime library

- Thread pool
- Socket listeners call functions on server

(2) Stub generation

- Create wrappers automatically
- Many tools available (rpcgen, thrift, protobufs)

Wrapper Generation

Wrappers must do conversions:

- client arguments to message
- message to server arguments
- convert server return value to message
- convert message to client return value

Need uniform endianness (wrappers do this)

Conversion is called marshaling/unmarshaling, or serializing/deserial

Wrapper Generation: Pointers

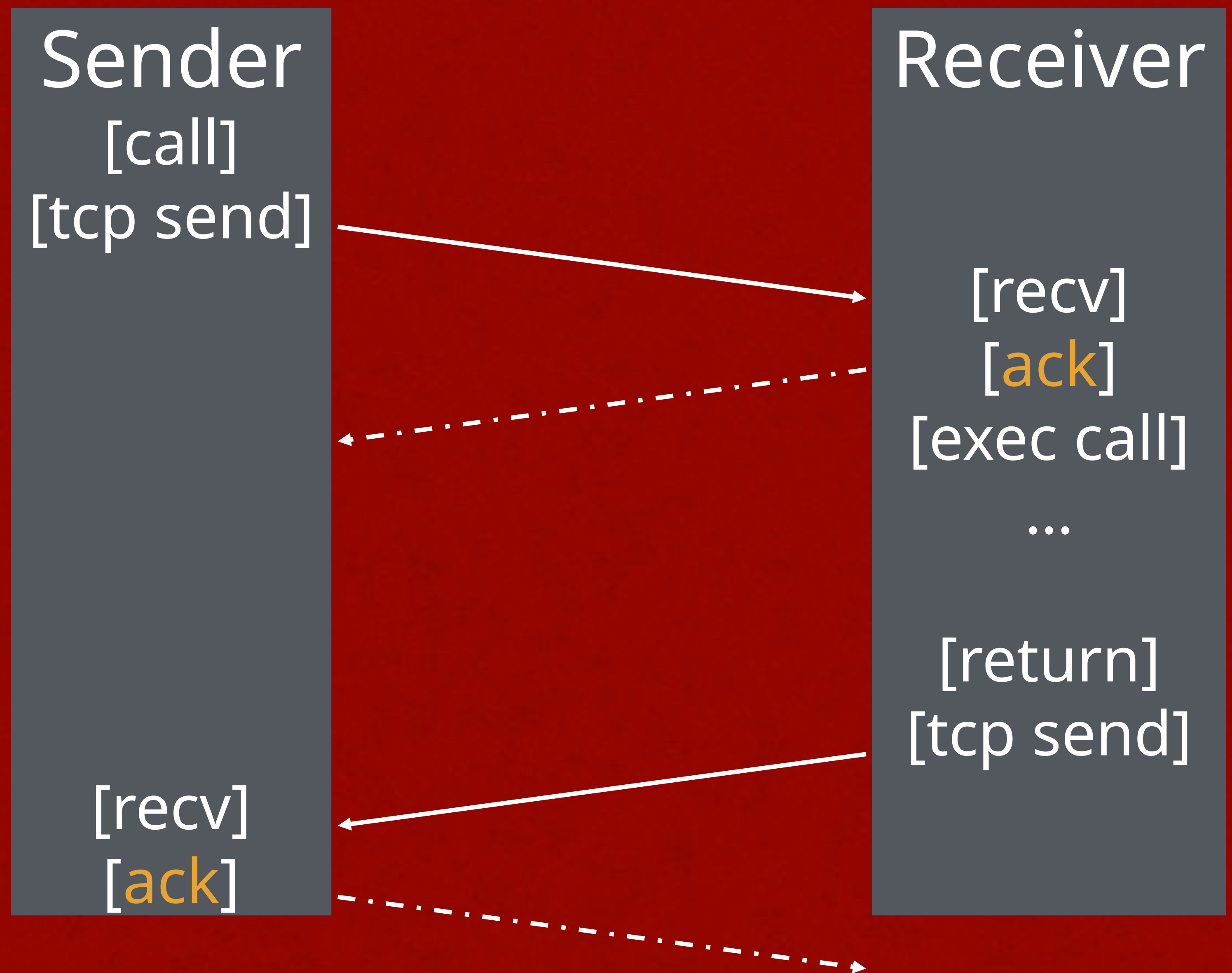
Why are pointers problematic?

Address passed from client not valid on server

Solutions?

- smart RPC package: follow pointers and copy data

RPC over TCP?



Why wasteful?

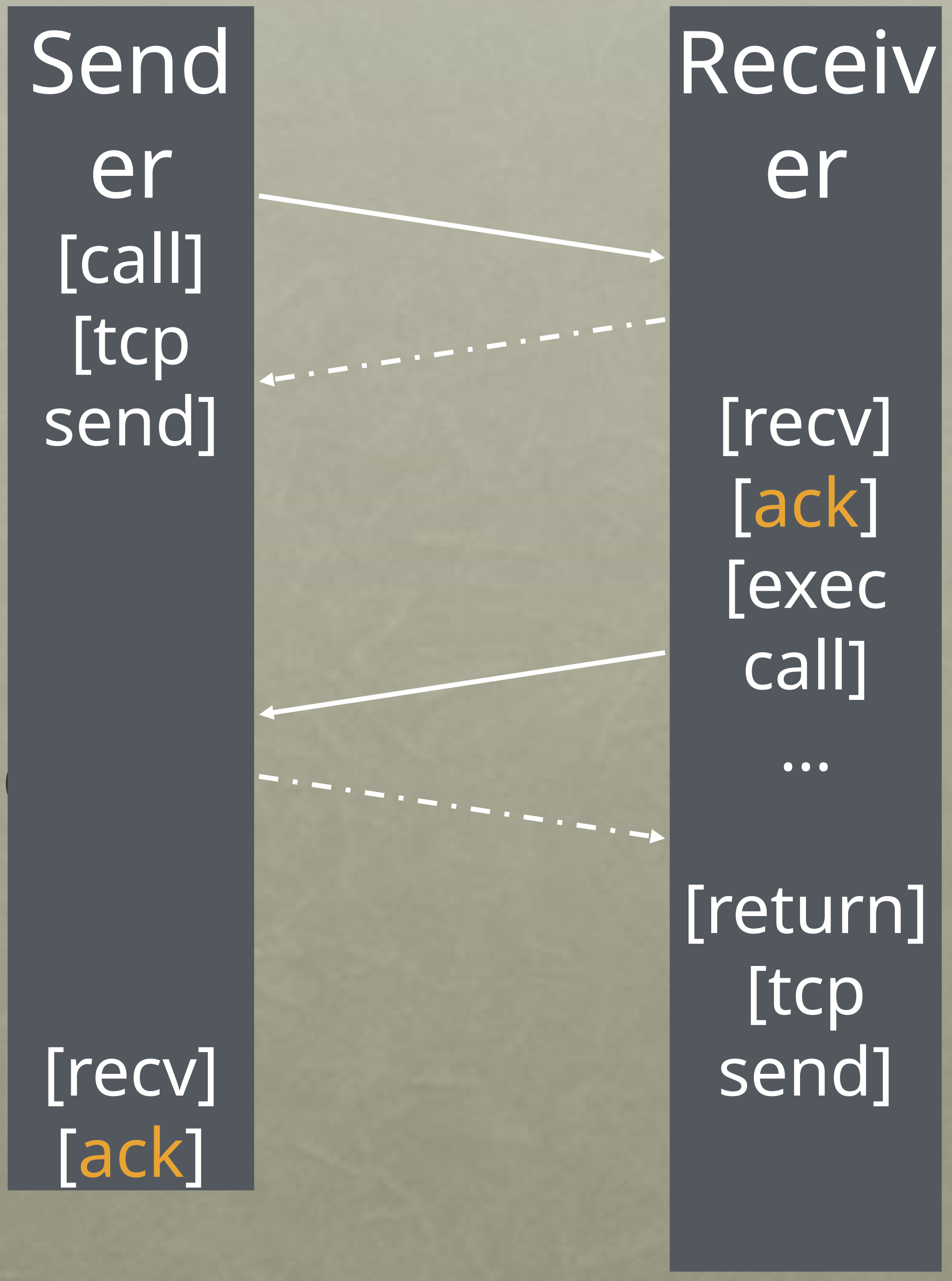
RPC over UDP

Strategy: use function return as implicit ACK

Piggybacking technique

What if function takes a long tim

- then send a separate ACK



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

NFS

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

Many companies have implemented NFS:
Oracle/Sun, NetApp, EMC, IBM

We're looking at NFSv2

- NFSv4 has many changes

Why look at an older protocol?

- Simpler, focused goals
- To compare and contrast NFS with AFS (next lecture)

Overview

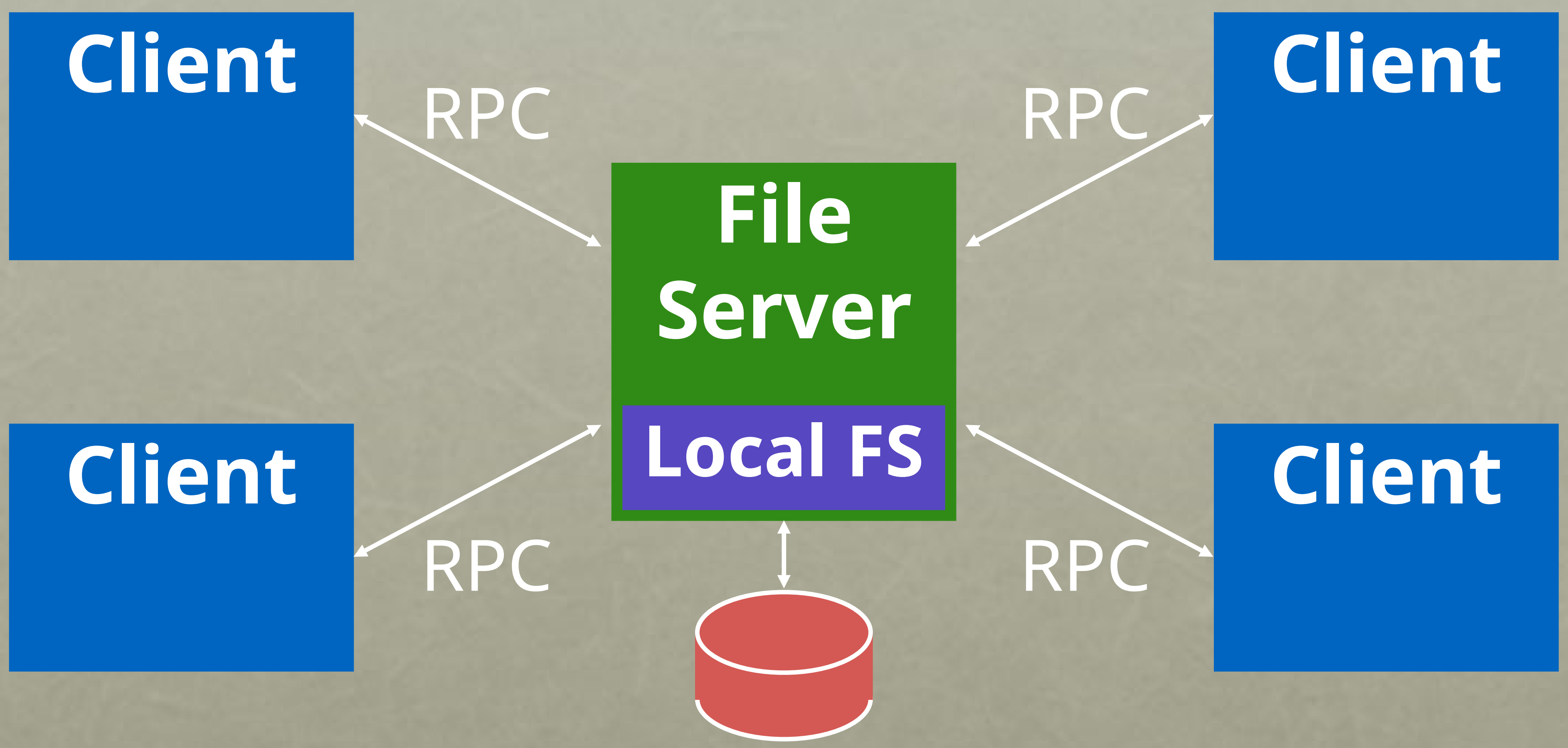
Architecture

Network API

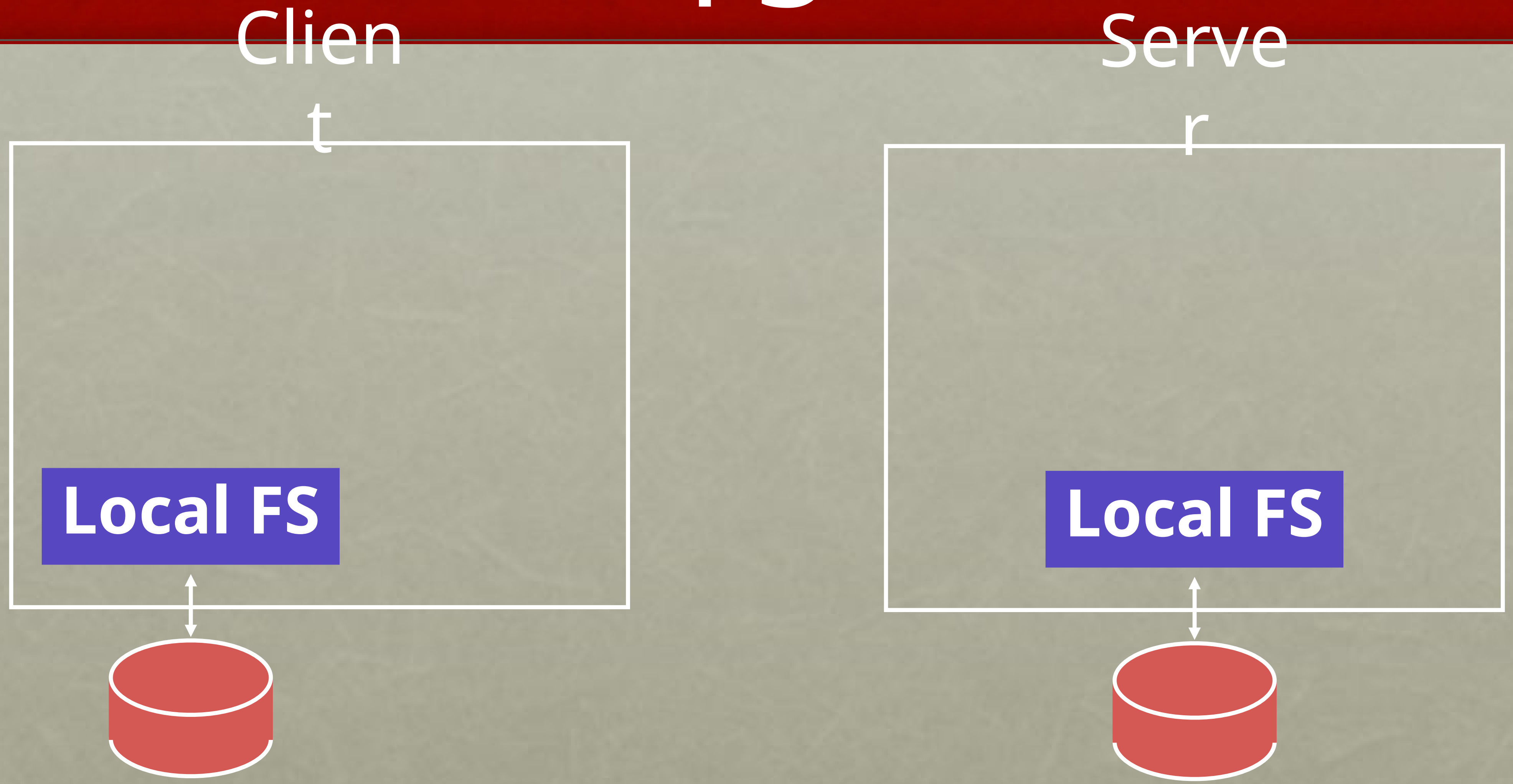
Write Buffering

Cache

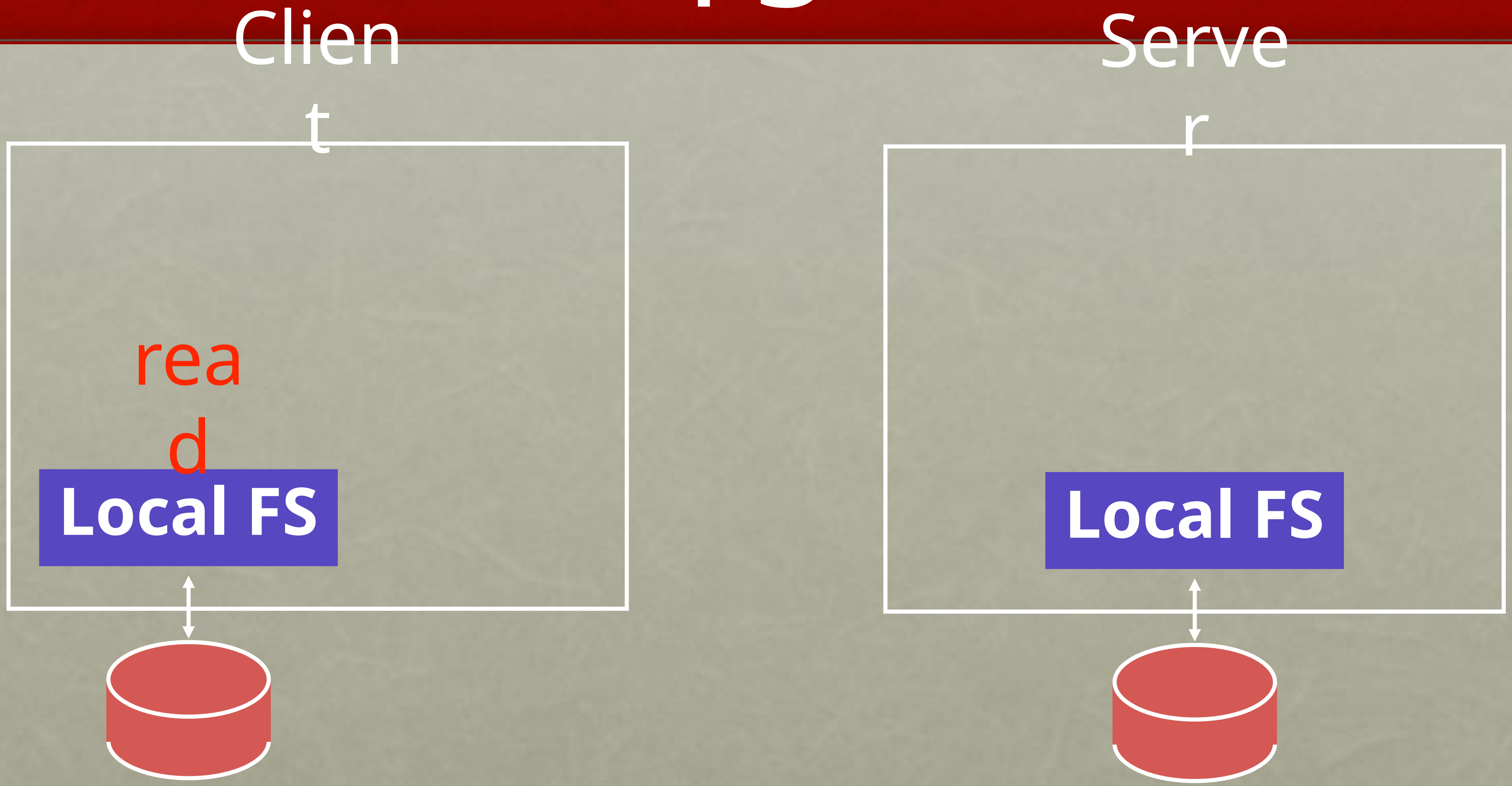
NFS Architecture



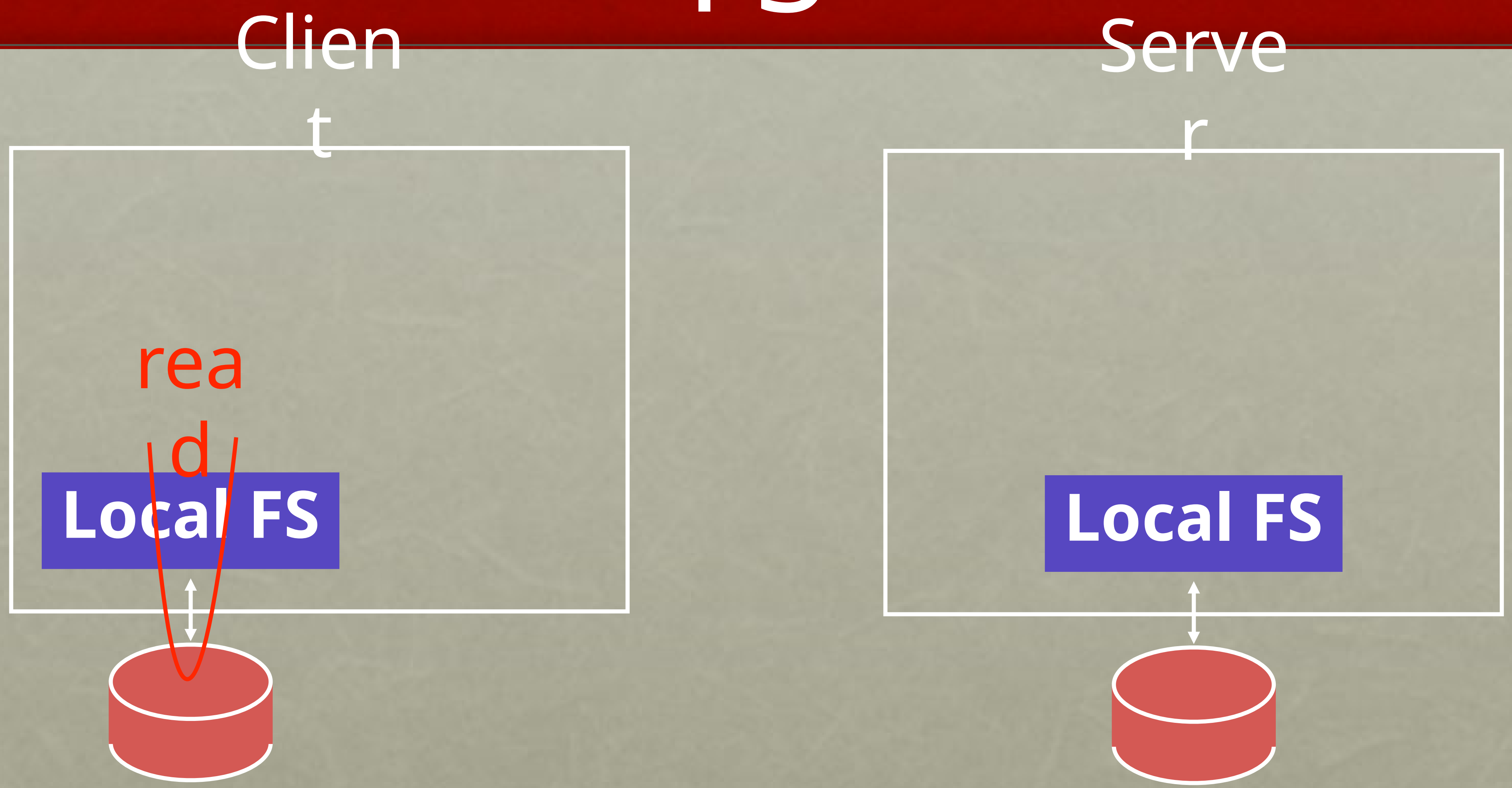
General Strategy: Export FS



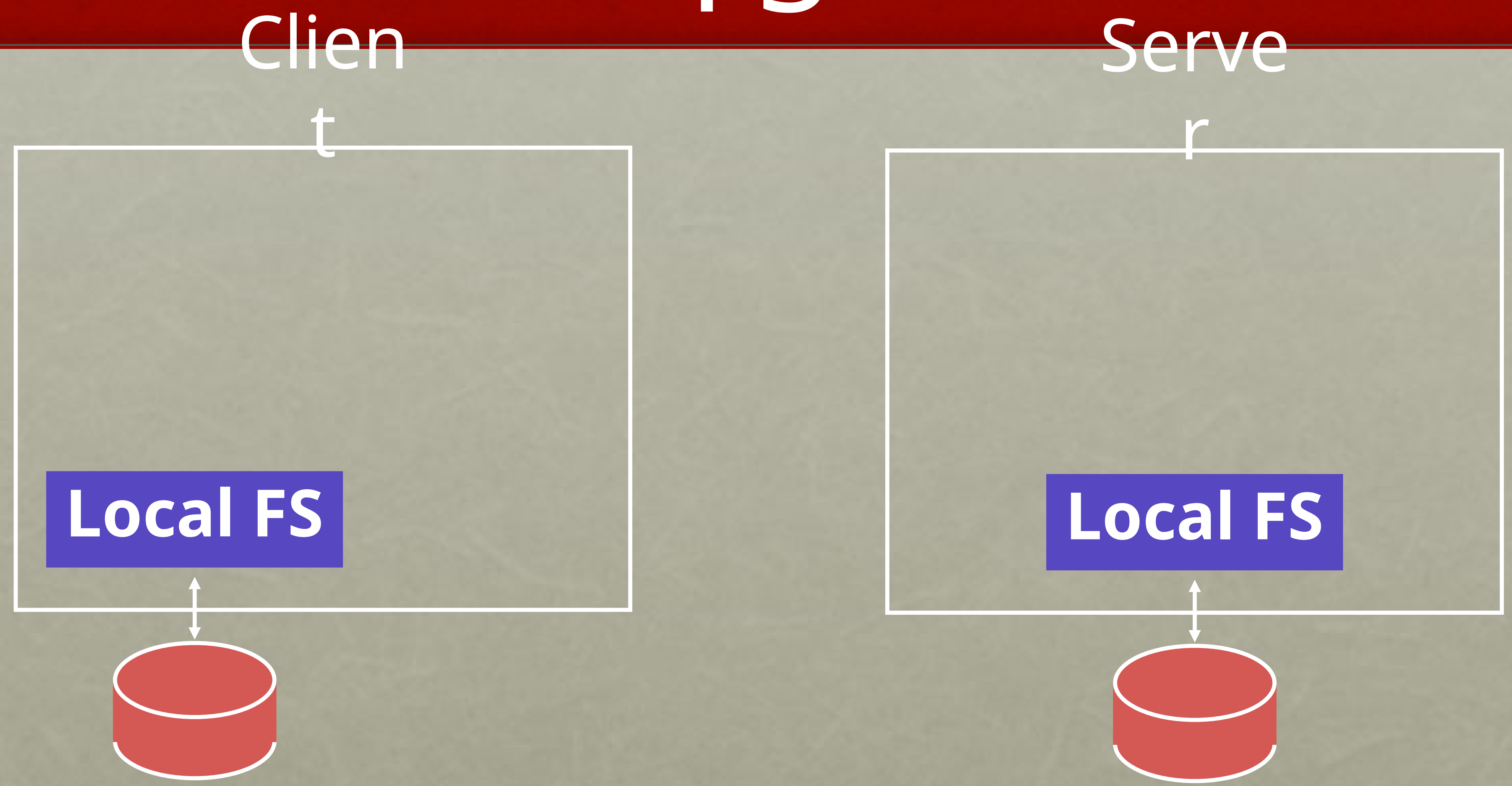
General Strategy: Export FS



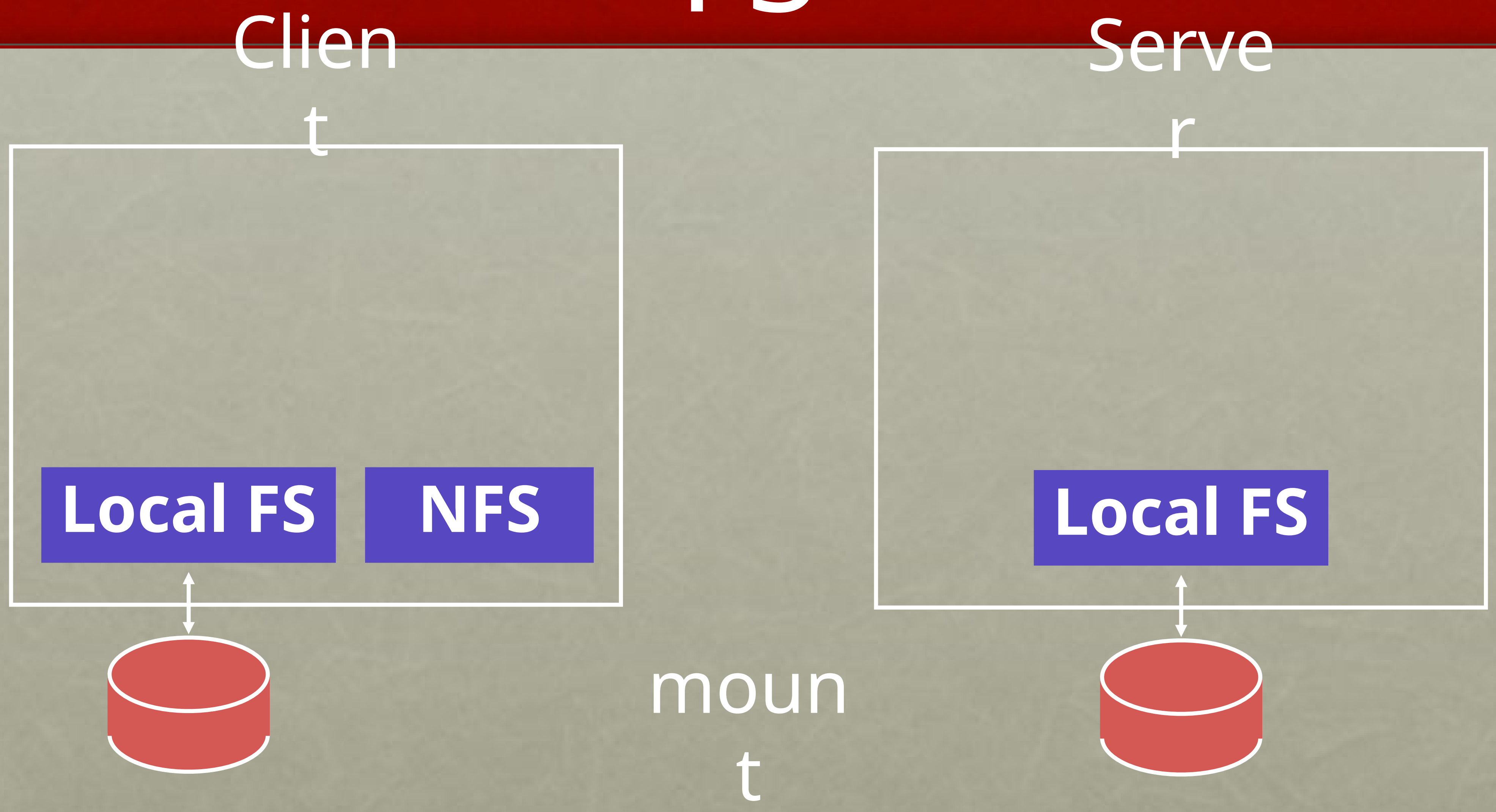
General Strategy: Export FS

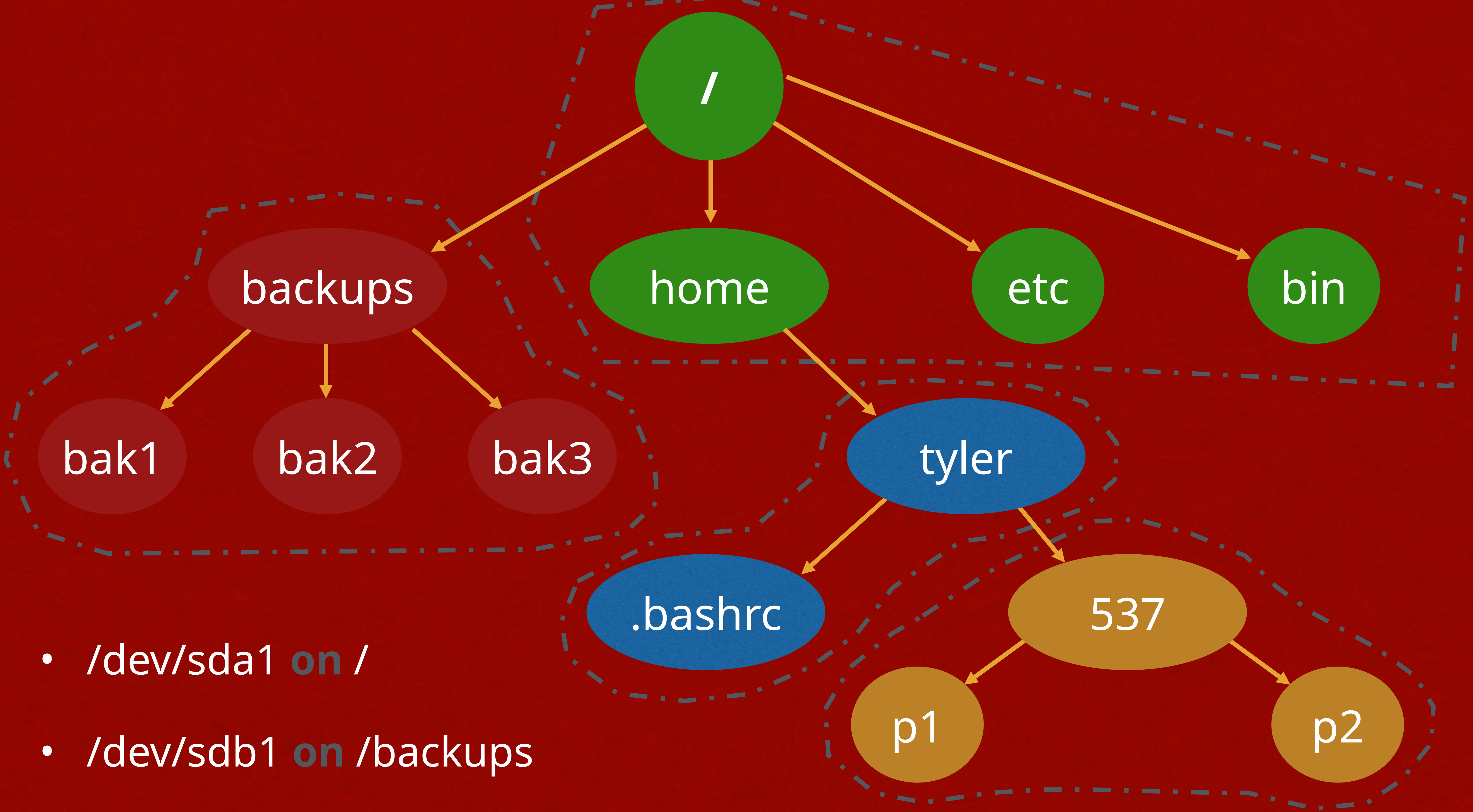


General Strategy: Export FS



General Strategy: Export FS

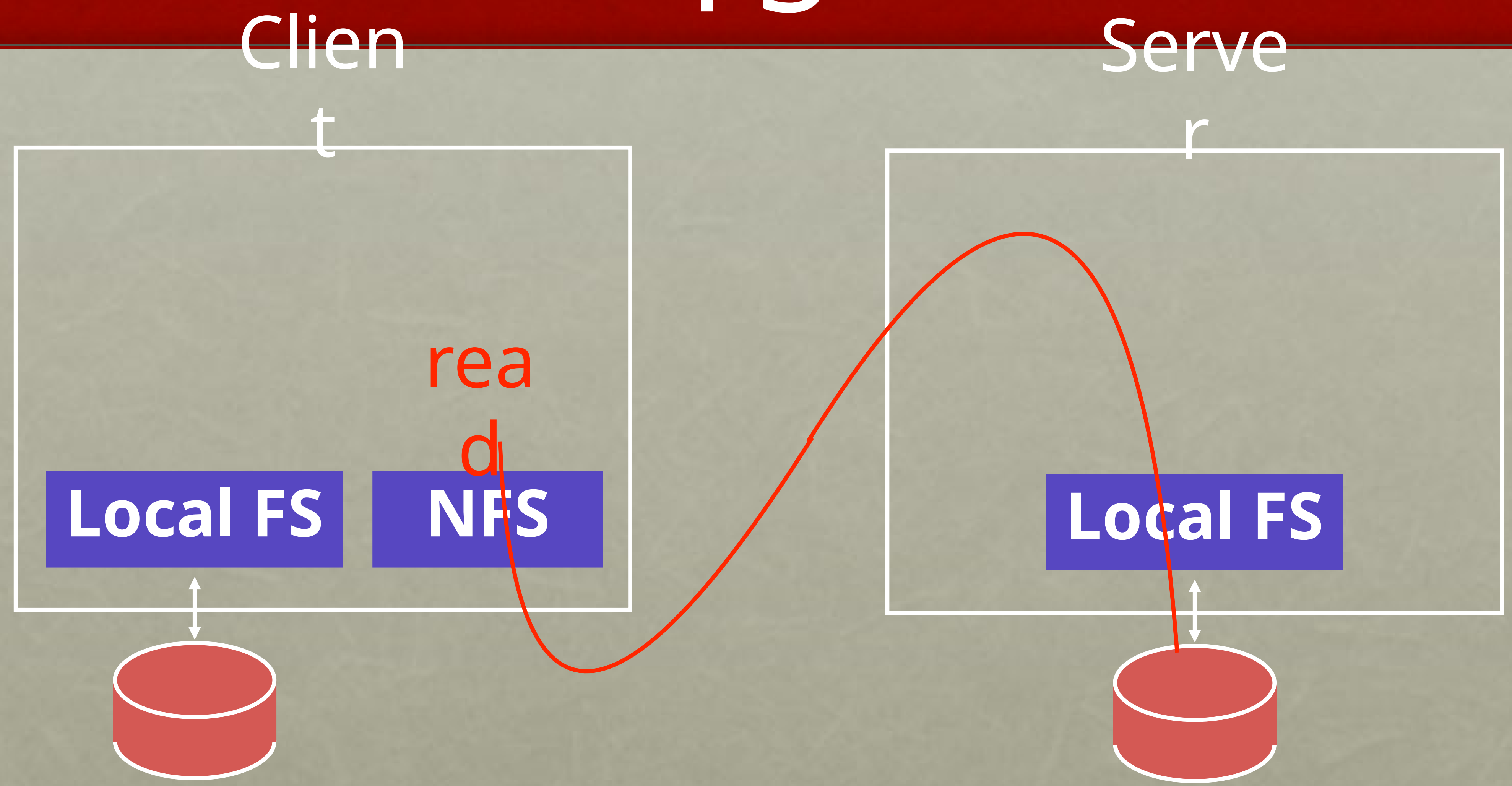




- /dev/sda1 **on** /
- /dev/sdb1 **on** /backups
- AFS **on** /home/tyler

General Strategy: Export FS

General Strategy: Export FS



Goals for NFS

Fast + simple crash recovery

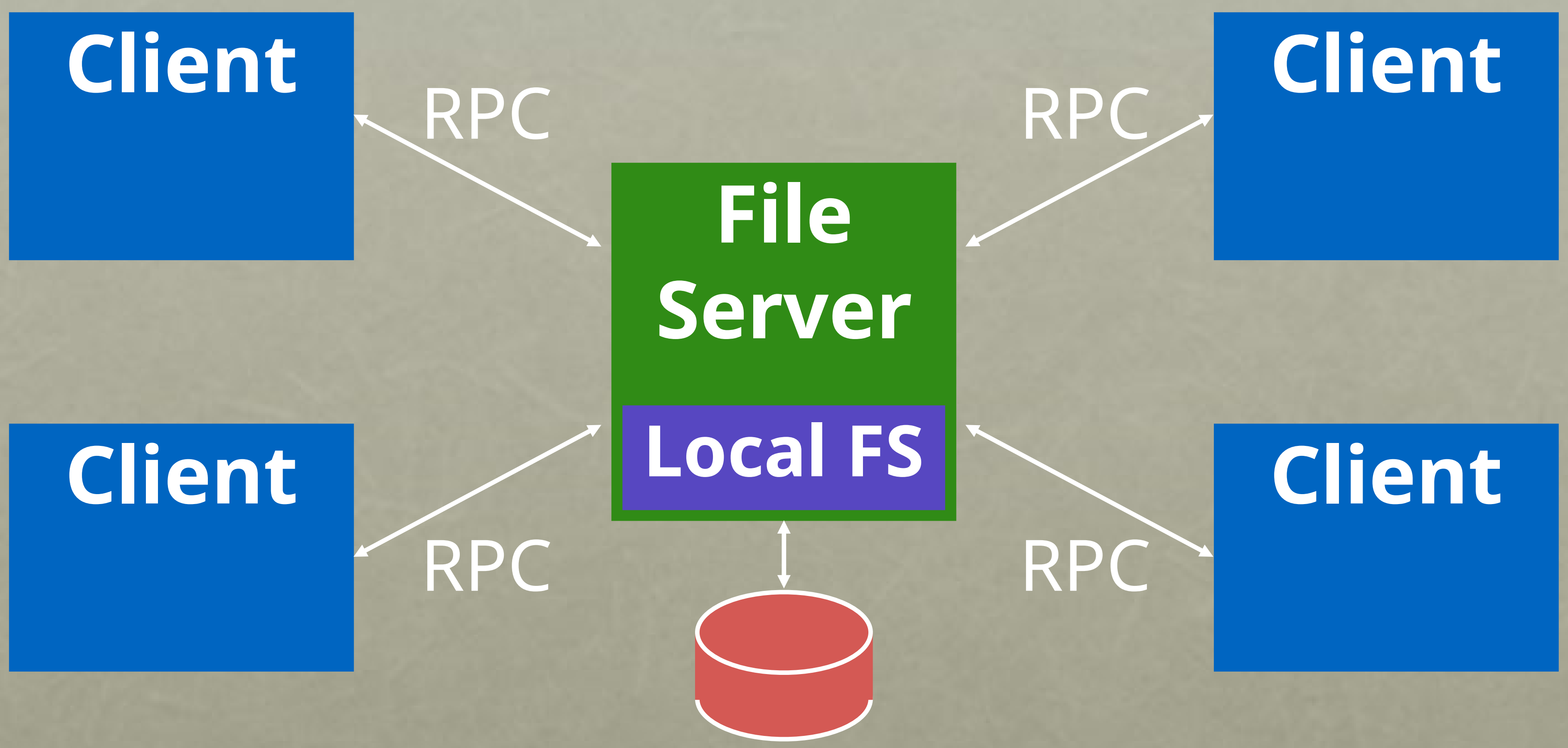
- both clients and file server may crash

Transparent access

- can’t tell accesses are over the network
- normal UNIX semantics

Reasonable performance

NFS Architecture



Overview

~~Architecture~~

Network API

Write Buffering

Cache

Strategy 1

Attempt: Wrap regular UNIX system calls using RPC

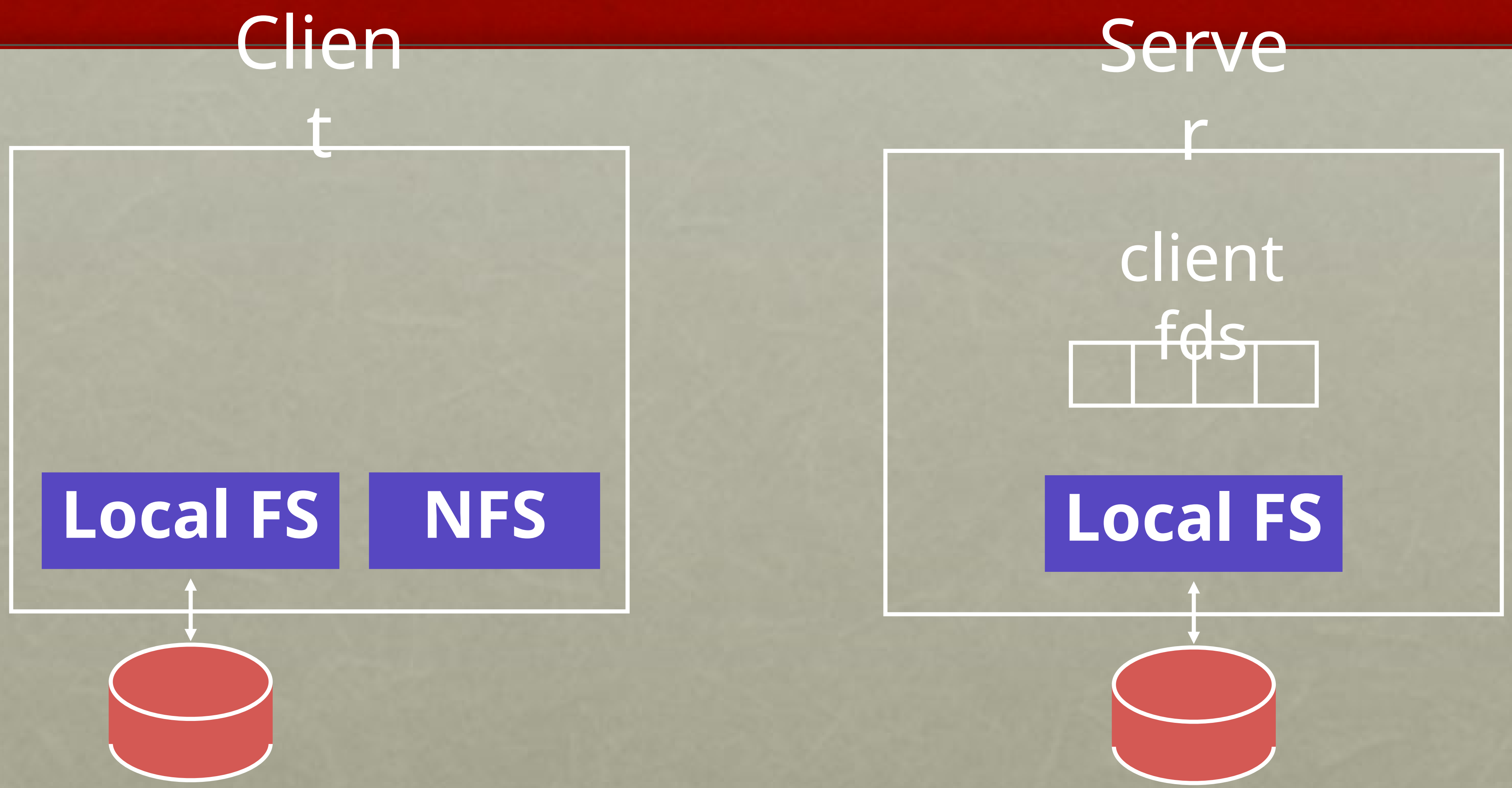
open() on client calls open() on server

open() on server returns fd back to client

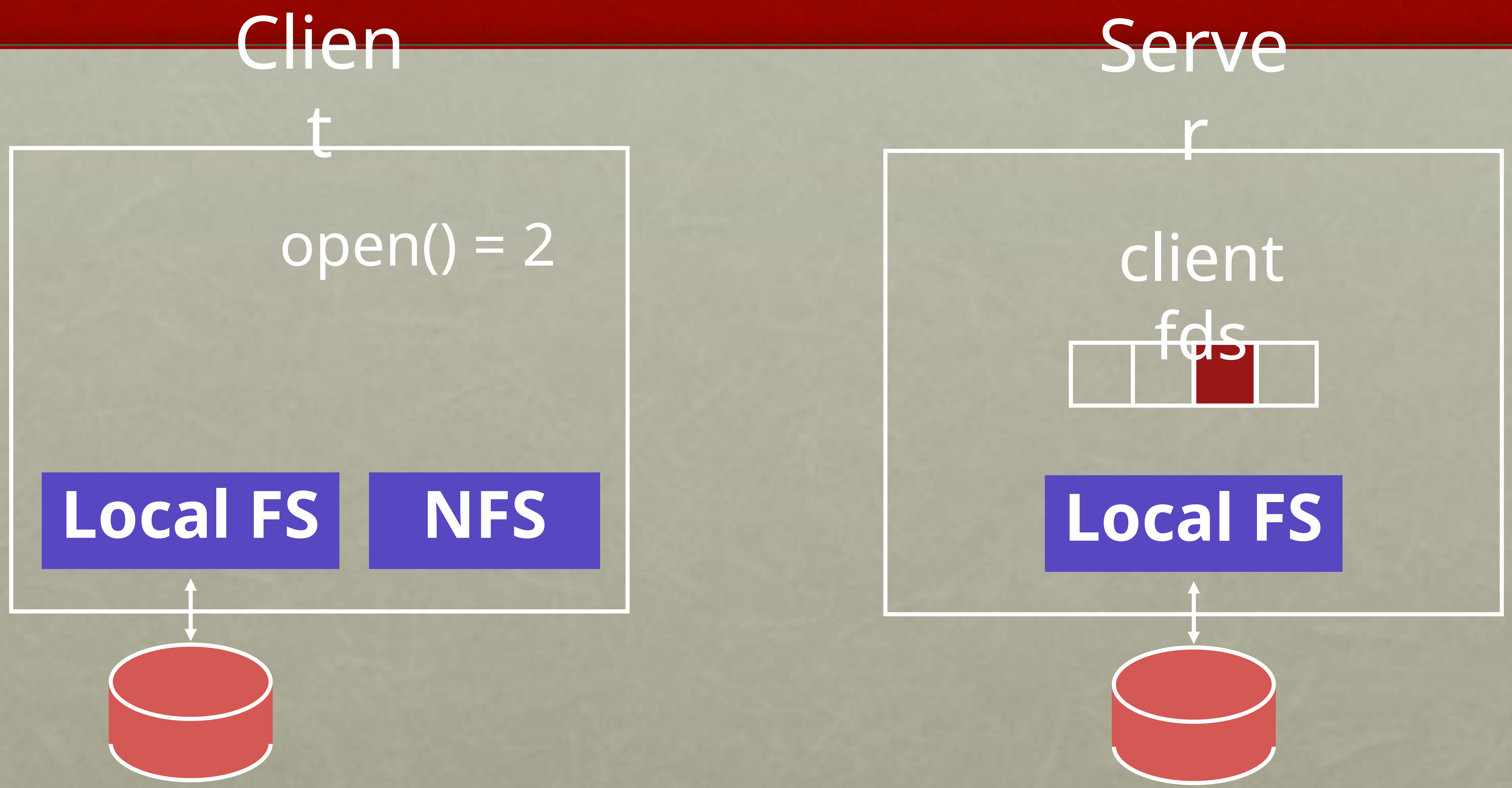
read(fd) on client calls read(fd) on server

read(fd) on server returns data back to client

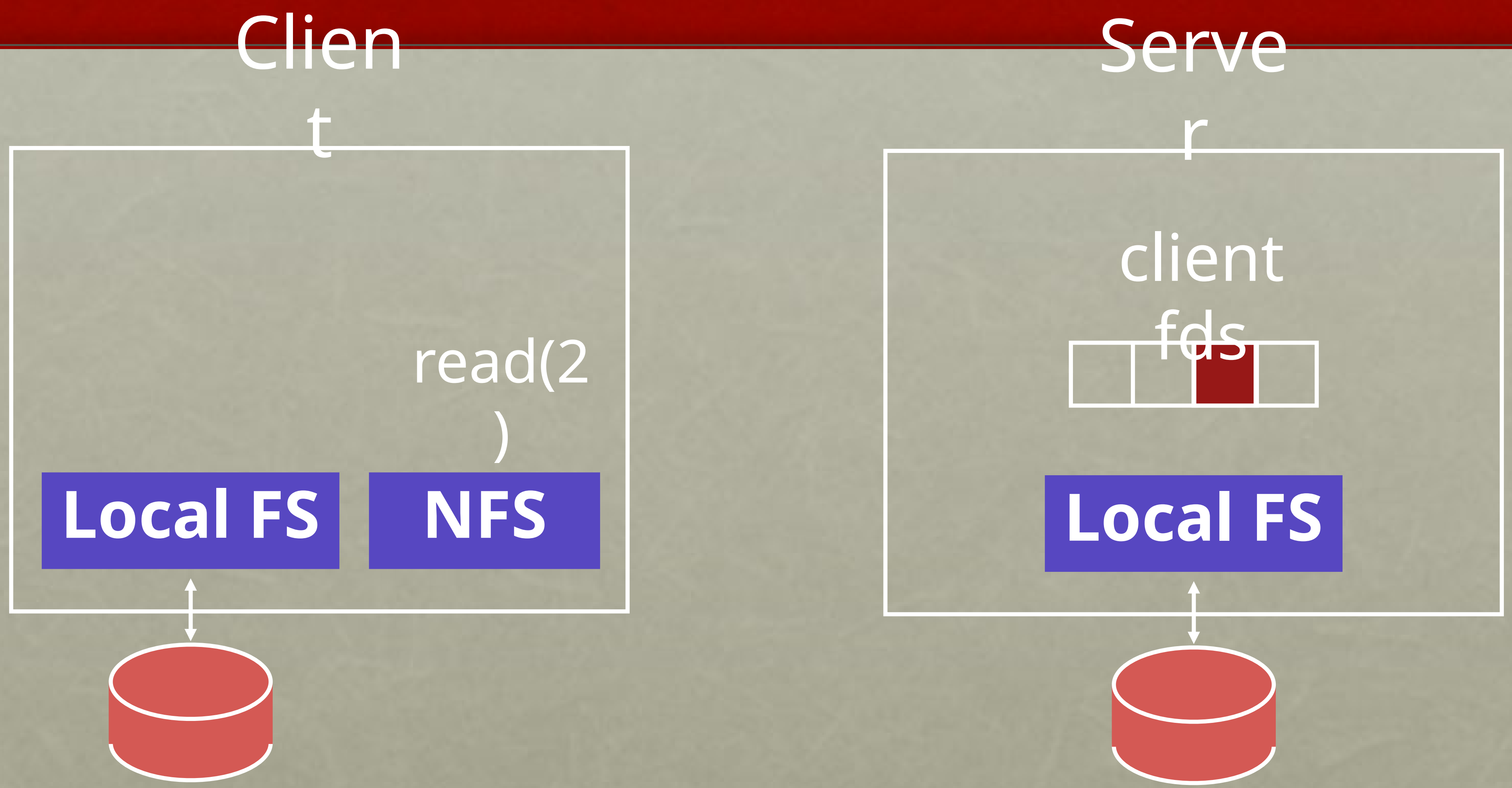
File Descriptors



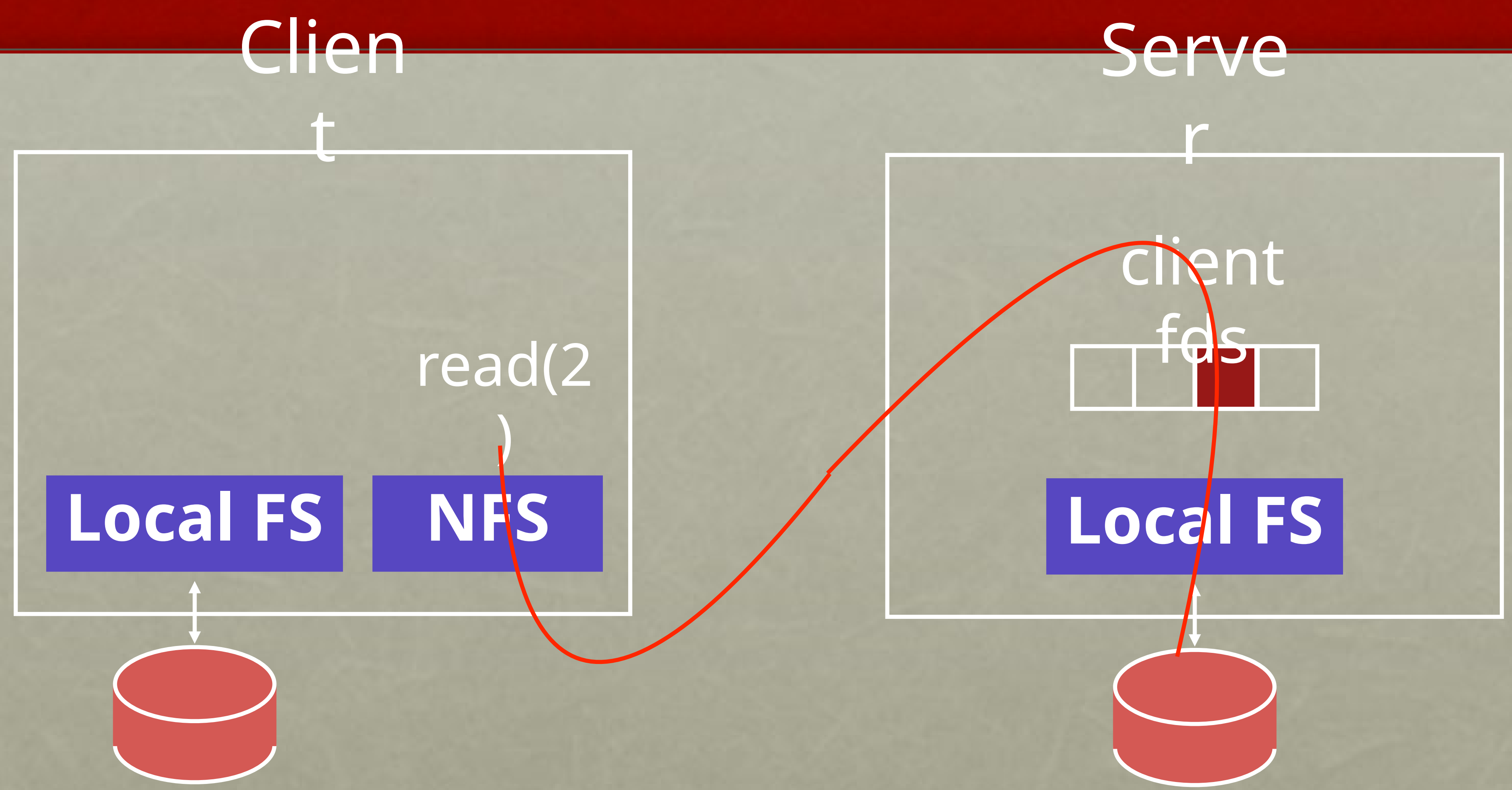
File Descriptors



File Descriptors



File Descriptors



Strategy 1 Problems

What about crashes?

int fd = open("foo", O_RDONLY);

read(fd, buf, MAX);

read(fd, buf, MAX);

…

read(fd, buf, MAX);

Server crash!

nice if acts 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? When can fds be garbage collected?

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

Strategy 2: 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.

Strategy 3: inode requests

inode = **open**(<u>char *path</u>);

pread(<u>inode</u>, <u>buf</u>, <u>size</u>, <u>offset</u>);

pwrite(<u>inode</u>, <u>buf</u>, <u>size</u>, <u>offset</u>);

This is pretty good! Any correctness problems?

If file is deleted, the inode could be reused

- Inode not guaranteed to be unique over time

Strategy 4: file handles

fh = **open**(char *path);

pread(fh, buf, size, offset);

pwrite(fh, buf, size, offset);

File Handle = <volume ID, inode #, **generation #**>

Opaque to client (client should not interpret internals)

Can NFS Protocol include Append?

fh = **open**(<u>char *path</u>);

pread(<u>fh</u>, <u>buf</u>, <u>size</u>, <u>offset</u>);

pwrite(<u>fh</u>, <u>buf</u>, <u>size</u>, <u>offset</u>);

append(<u>fh</u>, <u>buf</u>, <u>size</u>);

Problem with append()?

If RPC library retries, what happens when append() is retried?

Problem: Why is it difficult to not replay append()?

Replica Suppression is Stateful

Sender

[send message]

Receiver

[recv message]

[send ack]

[timout]

[send message]

[ignore message]

[send ack]

TCP suppresses repeated message

[recv ack]

Problem: TCP is stateful

If server crashes, forgets which RPC's have been executed!

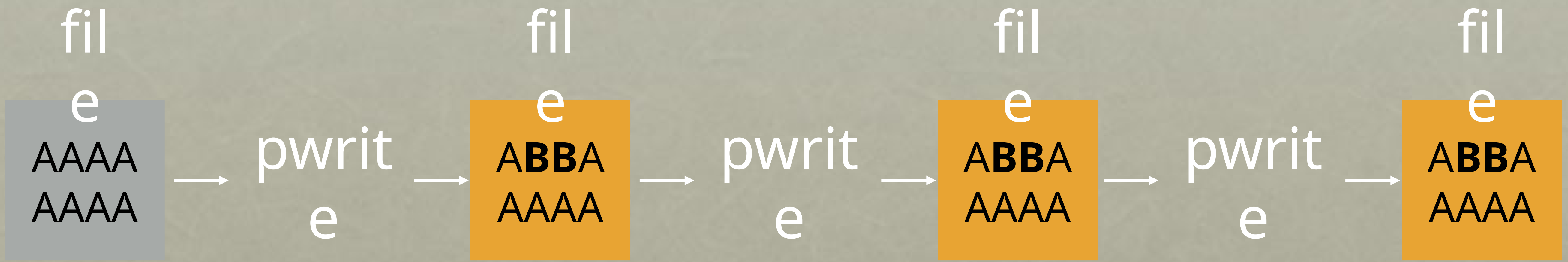
Idempotent Operations

Solution:
Design API so no harm to executing function more than once

If f() is idempotent, then:

    f() has the same effect as f(); f(); … f(); f()

pwrite is idempotent



append is NOT idempotent

What operations are Idempotent?

Idempotent

- any sort of read that doesn't change anything
- 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 5: client logic

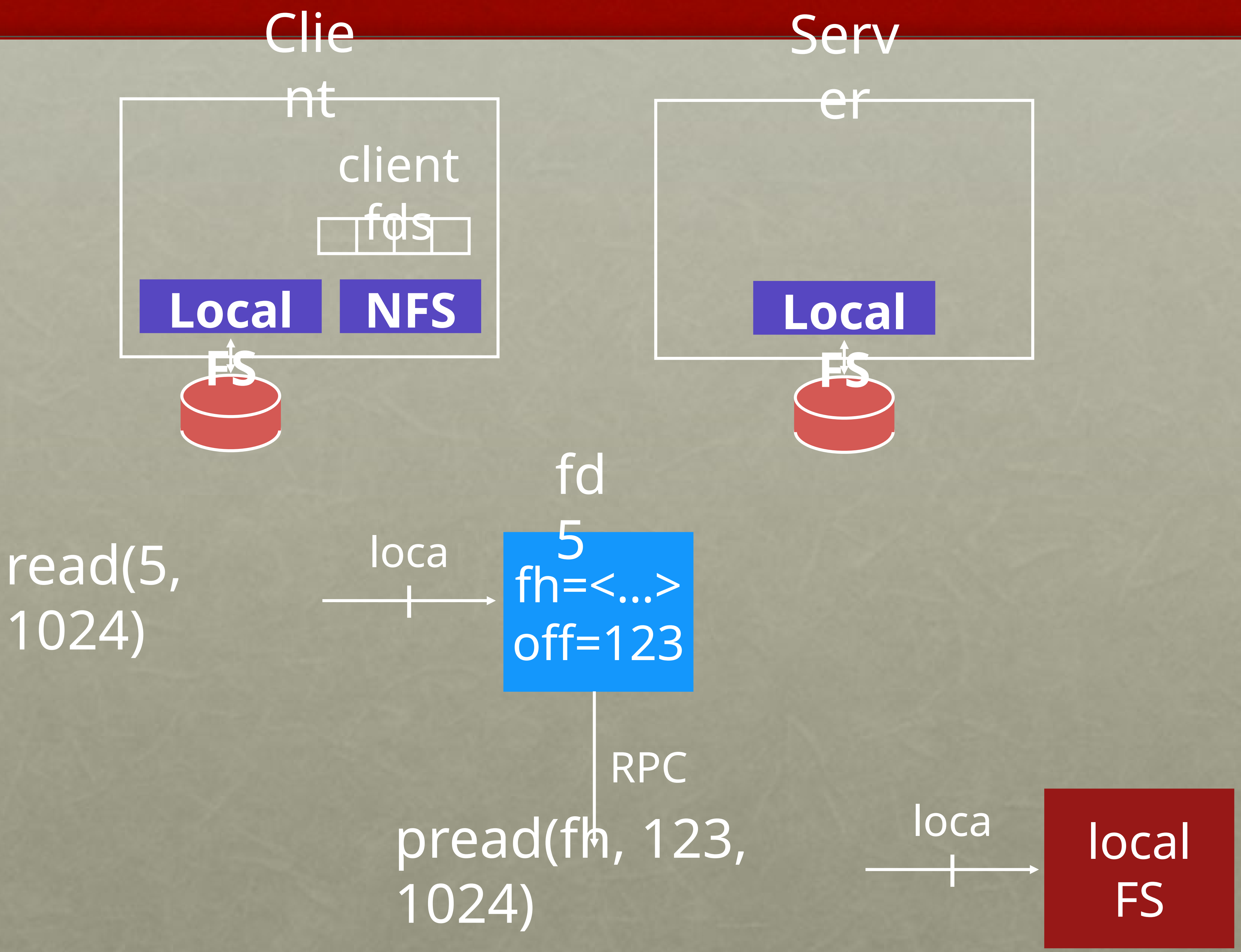
Build normal UNIX API on client side on top of idempotent, RPC-based API

Client open() creates a local fd object

It contains:

- file handle
- offset

File Descriptors



Overview

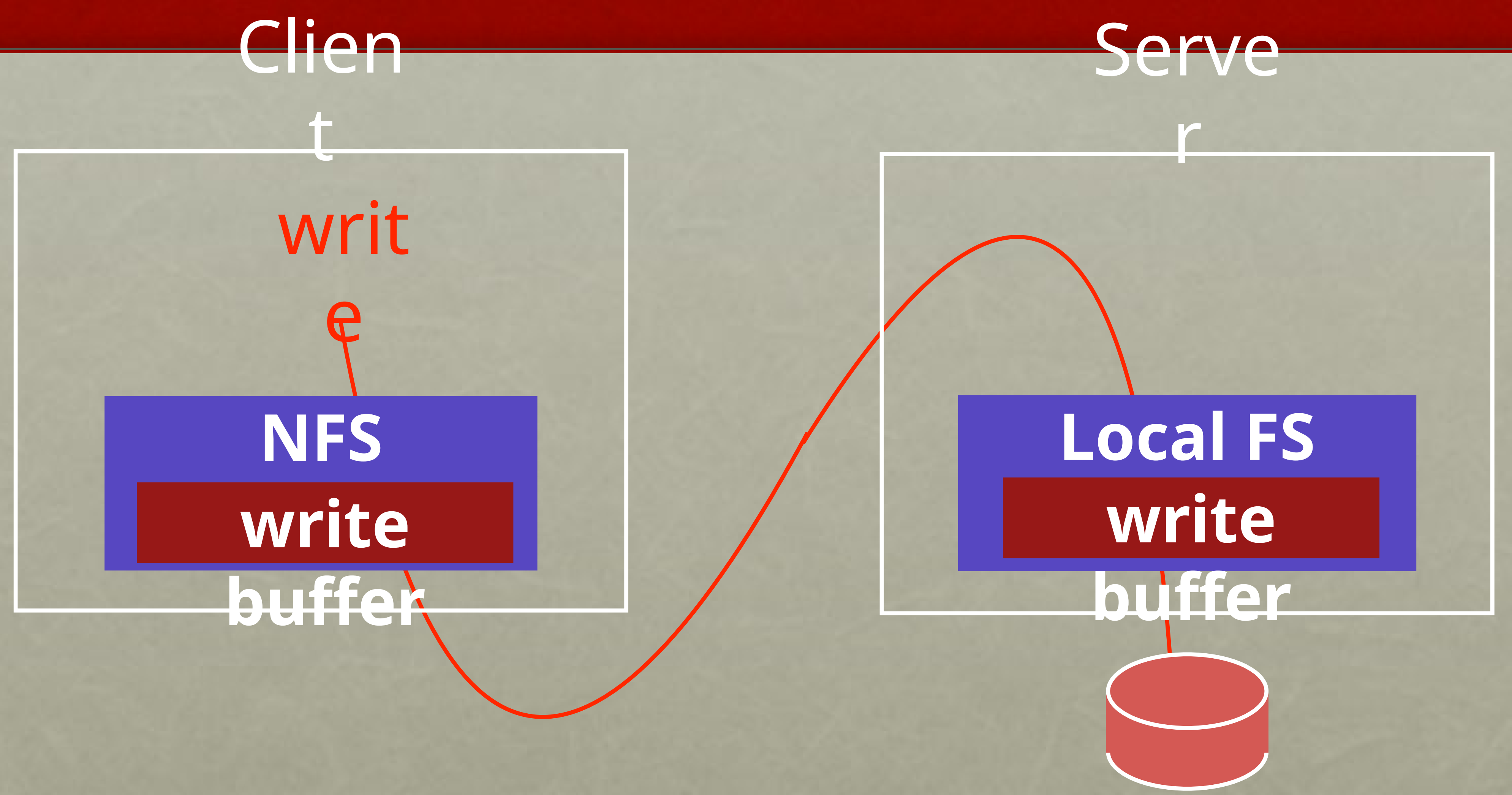
~~Architecture~~

~~Network API~~

Write Buffering

Cache

Write Buffers



server acknowledges write before write is pushed to disk;
what happens if server crashes?

Server Write Buffer Lost

client:

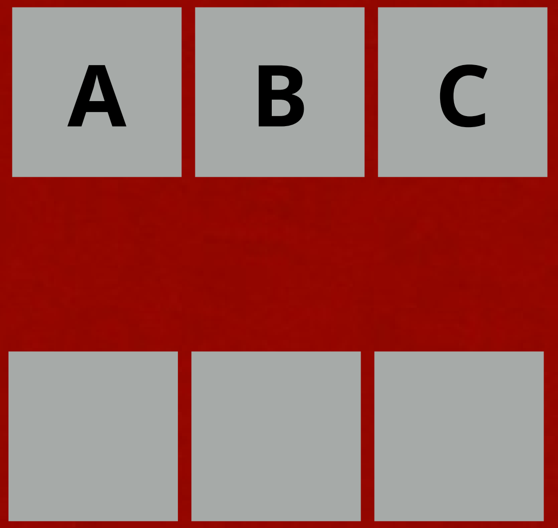
write A to 0

write B to 1

write C to 2

server mem:

server disk:



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 mem:

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 |

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

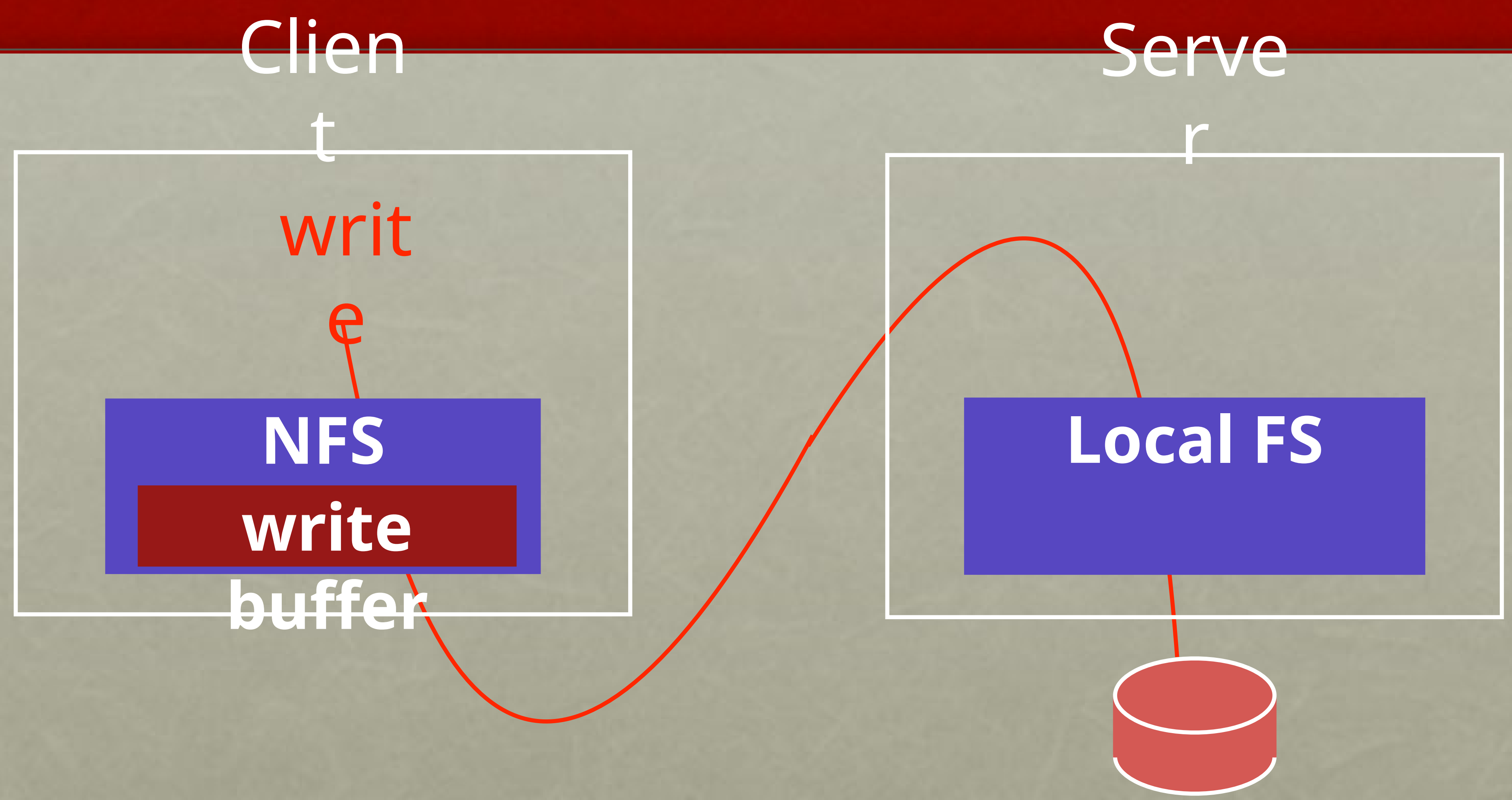
server disk:

X	B	Z

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

Solutions????

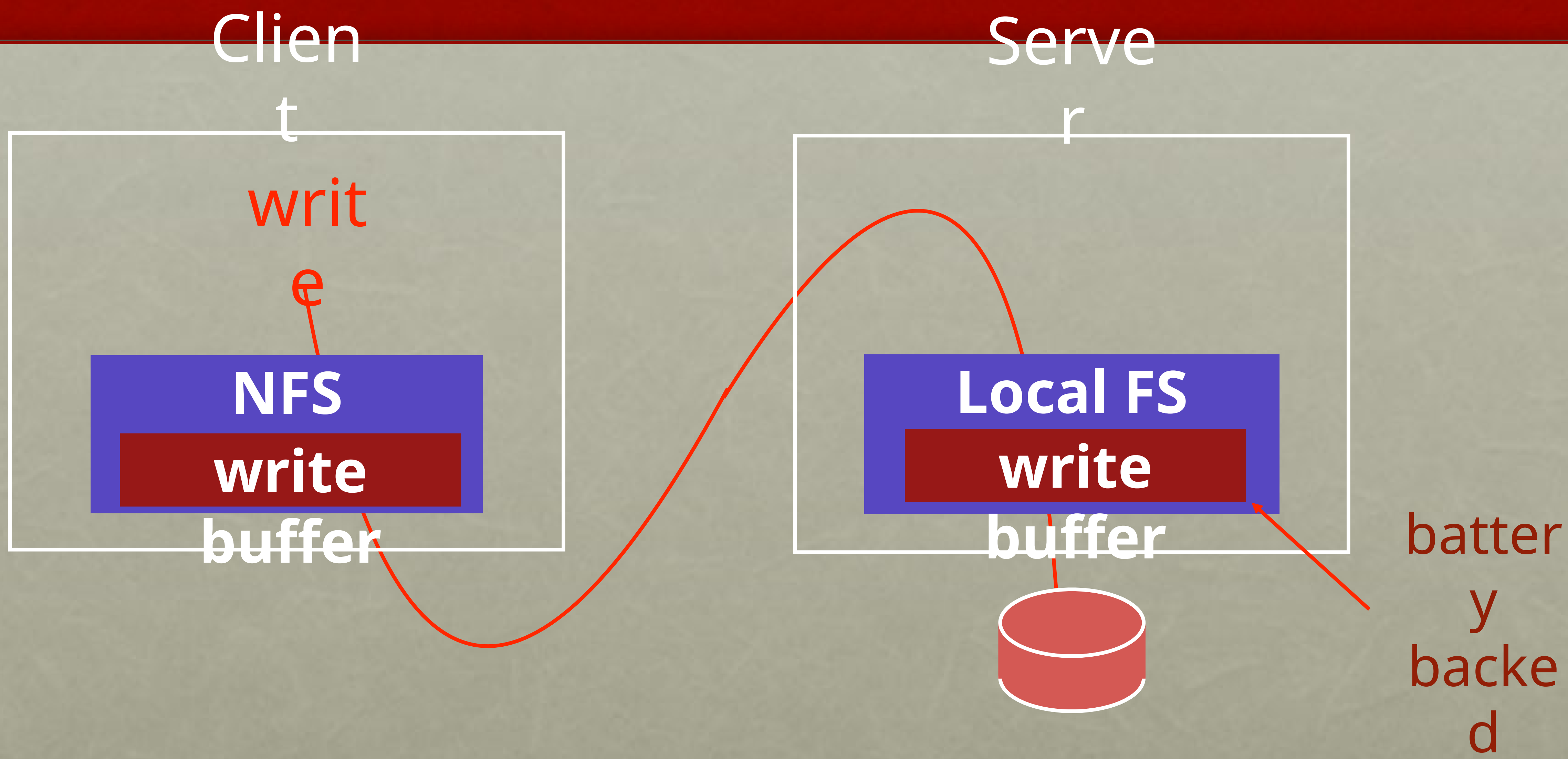
Write Buffers



1. Don’t use server write buffer
(persist data to disk before acknowledging write)

Problem: Slow!

Write Buffers



2. use persistent write buffer (more expensive)

Overview

~~Architecture~~

~~Network API~~

~~Write Buffering~~

Cache

Cache Consistency

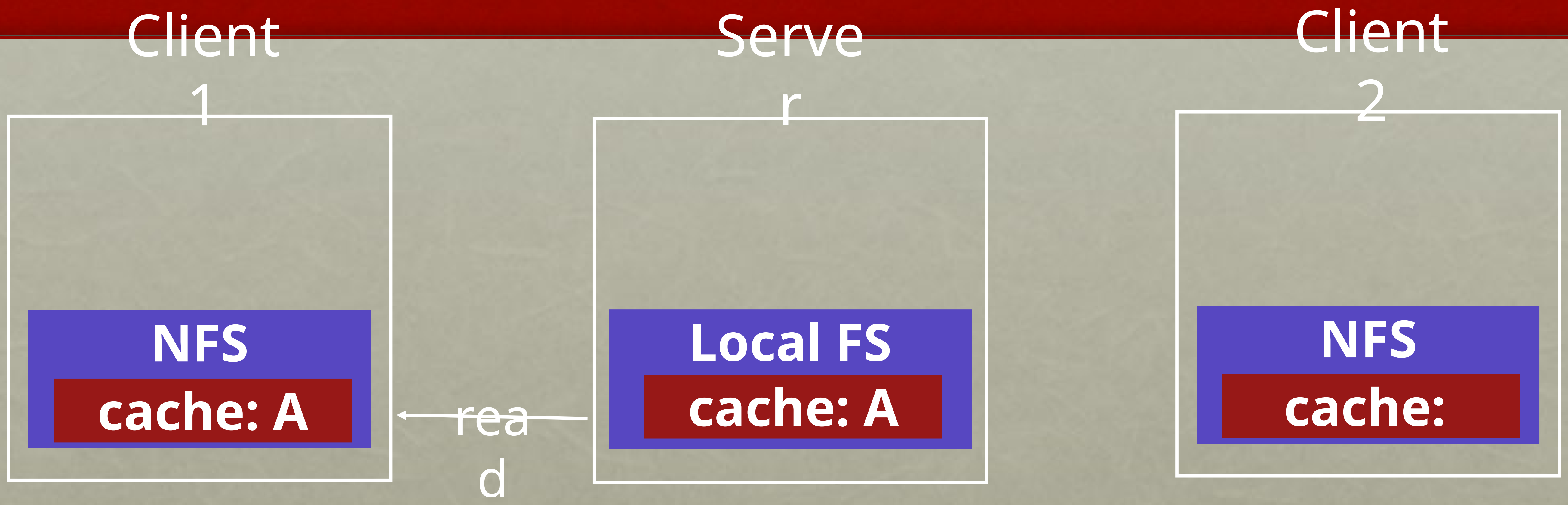
NFS can cache data in three places:

- server memory
- client disk
- client memory

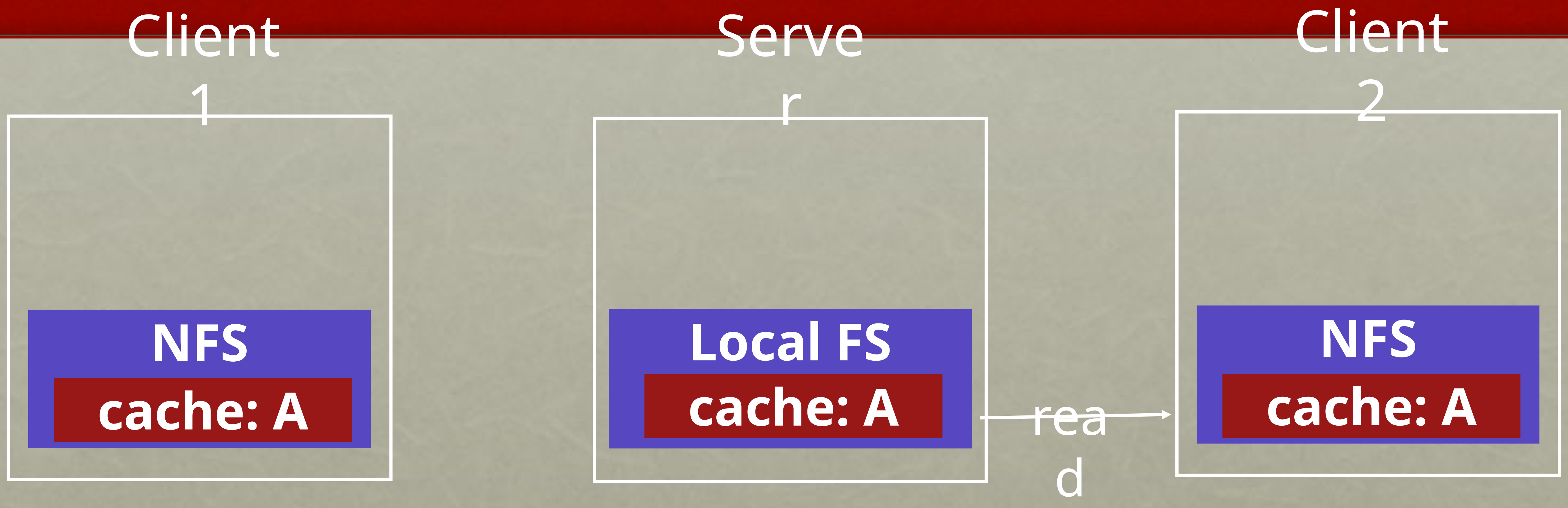
How to make sure all versions are in sync?

Distributed Cache

Cache


Client 1
Server
Client 2
NFS
cache: A
read
Local FS
cache: A
NFS
cache:

Cache

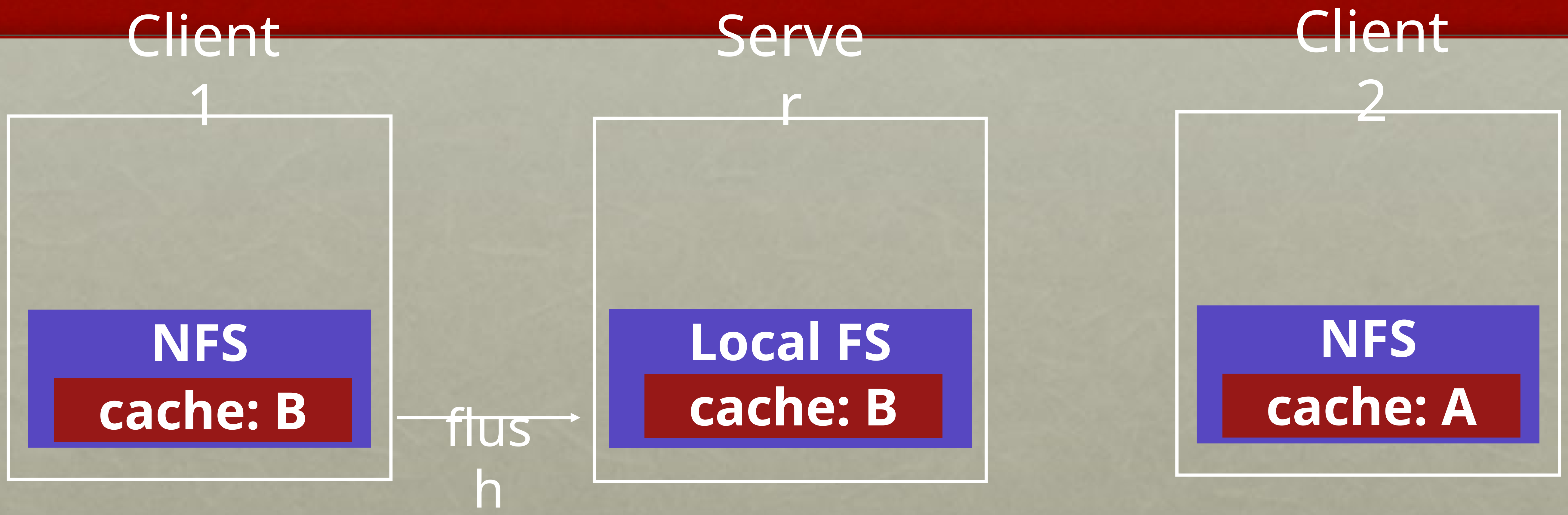

Client 1
NFS
cache: A
Server
Local FS
cache: A
read
Client 2
NFS
cache: A

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)

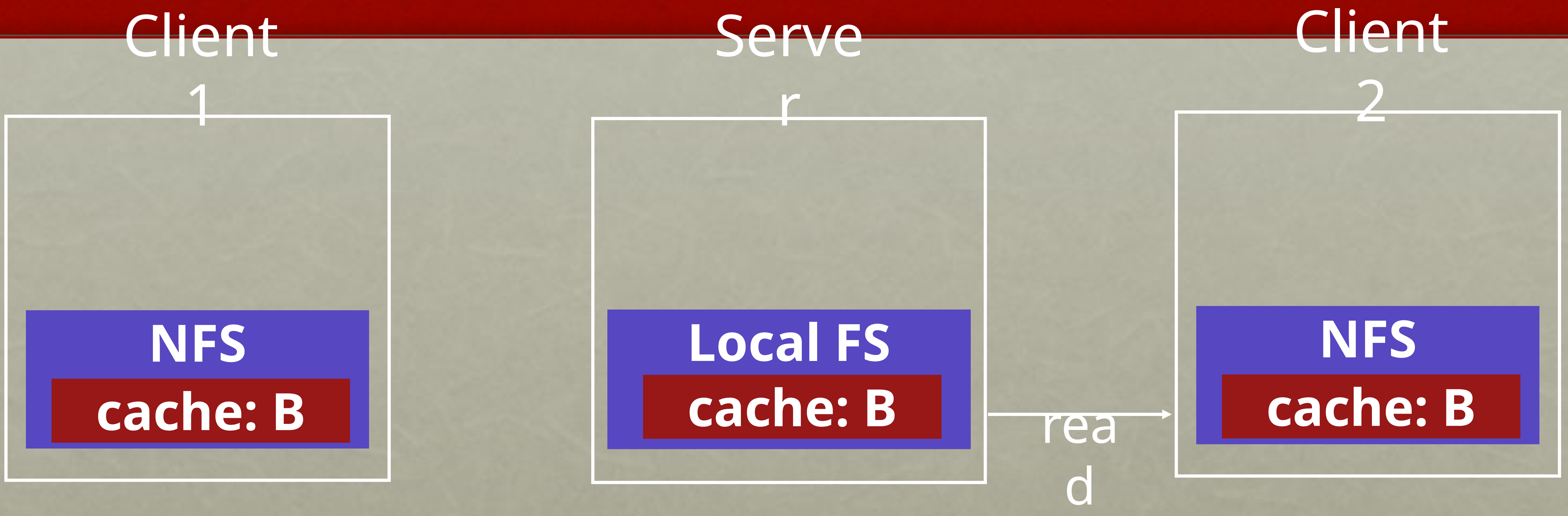
Cache



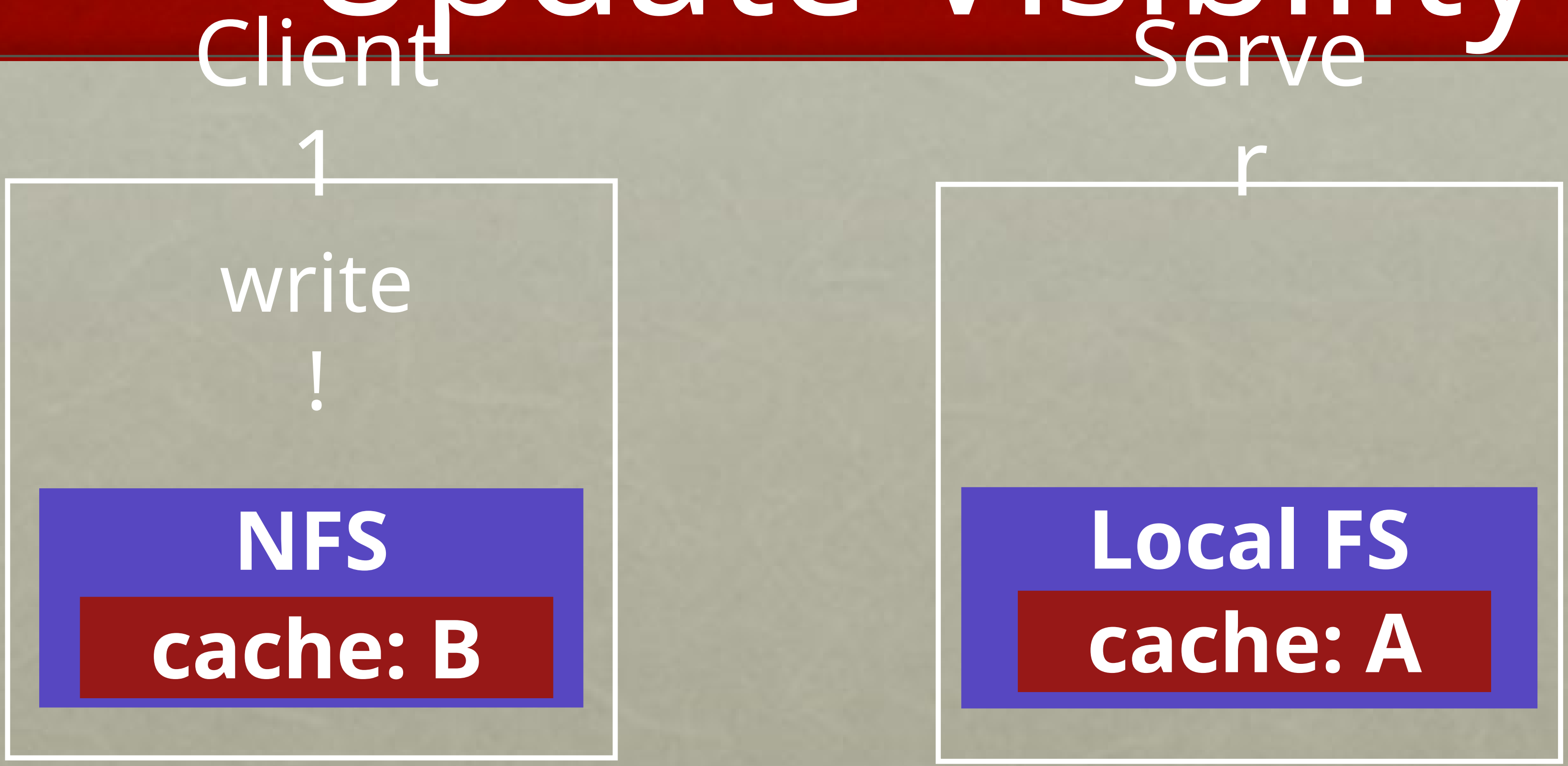
“Stale Cache” problem:
client 2 doesn’t have latest version

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

Cache


Client 1
NFS
cache: B
Server
Local FS
cache: B
read
Client 2
NFS
cache: B

Problem 1:
Update Visibility



When client buffers a write, how can server (and other clients) see update?

- Client flushes cache entry to server

When should client perform flush????? (3 reasonable options??)

NFS solution: flush on fd close

Problem 2: Stale Cache

Problem: Client 2 has stale copy of data; how can it get the latest?

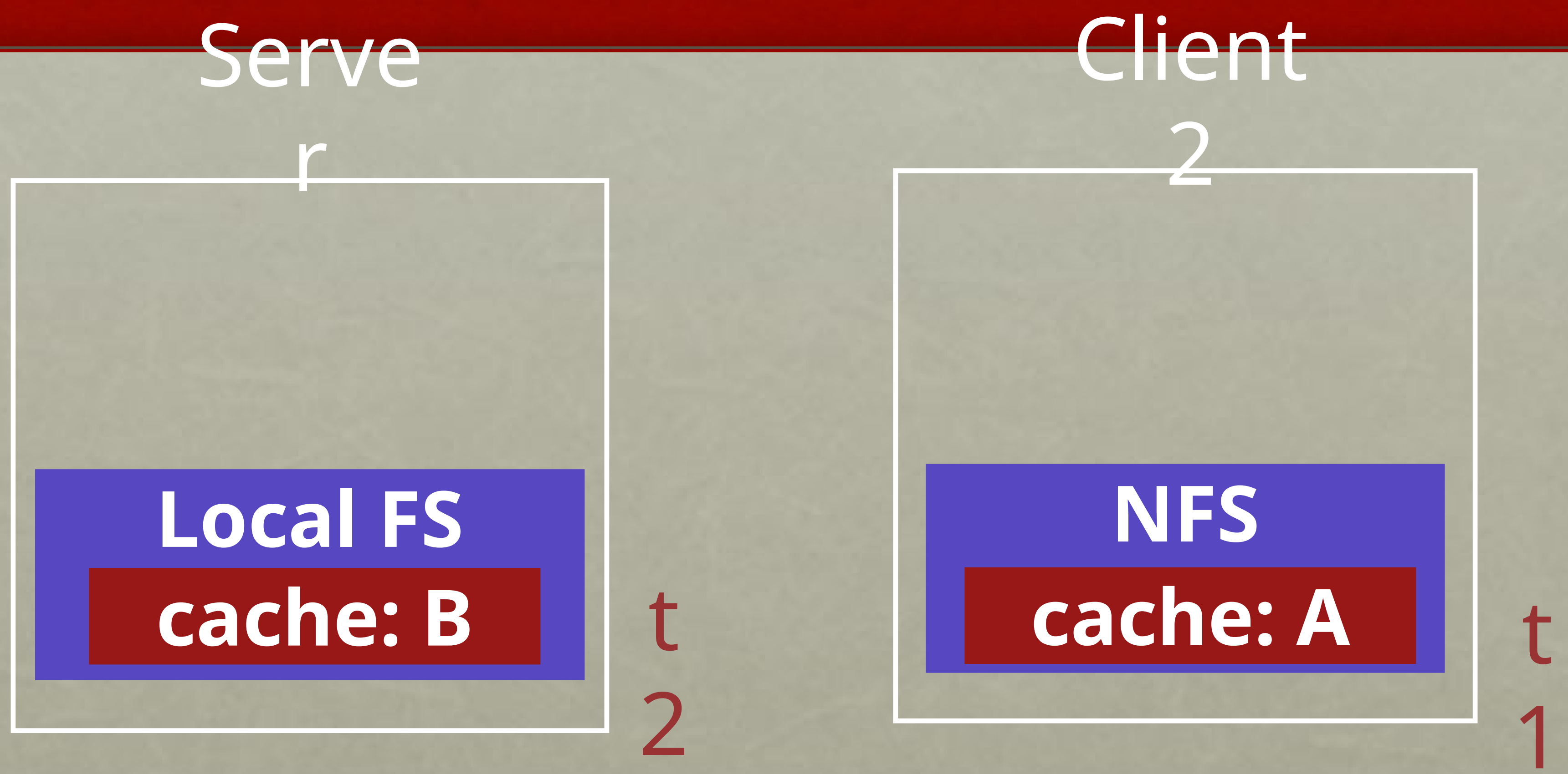
One possible solution:

- If NFS had state, server could push out update to relevant clients

NFS solution:

- Clients recheck if cached copy is current before using data

Stale Cache Solution



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)

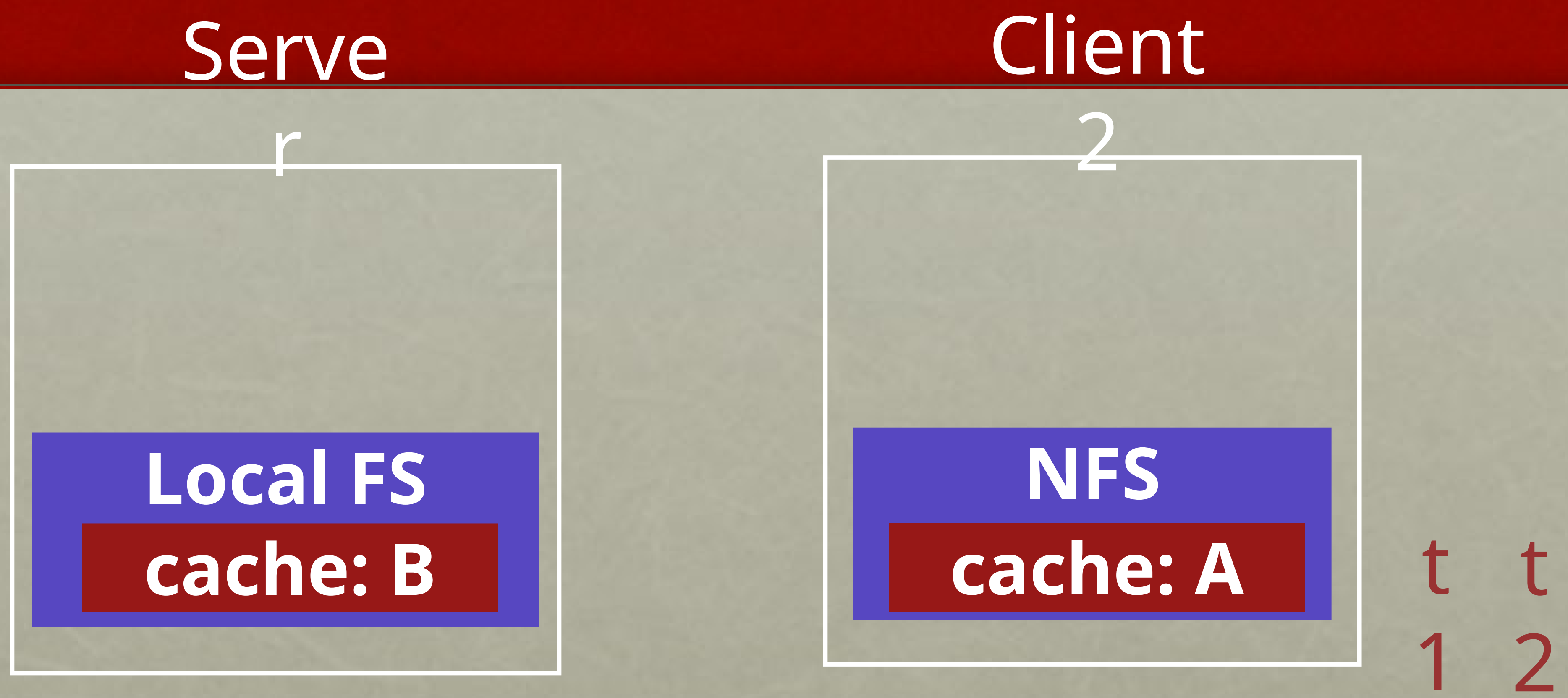
Measure then Build

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

What is the result? Never see updates on server!

Partial Solution: Make stat cache entries expire after a given time (e.g., 3 seconds) (discard t2 at client 2)

What is the result? Could read data that is up to 3 seconds old

NFS Summary

NFS handles client and server crashes very well; robust APIs are often:

- **stateless**: servers don’t remember clients

- **idempotent**: doing things twice never hurts

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

Problems:

- Consistency model is odd (client may not see updates until 3 seconds after file is closed)
- Scalability limitations as more clients call stat() on server

AFS Goals

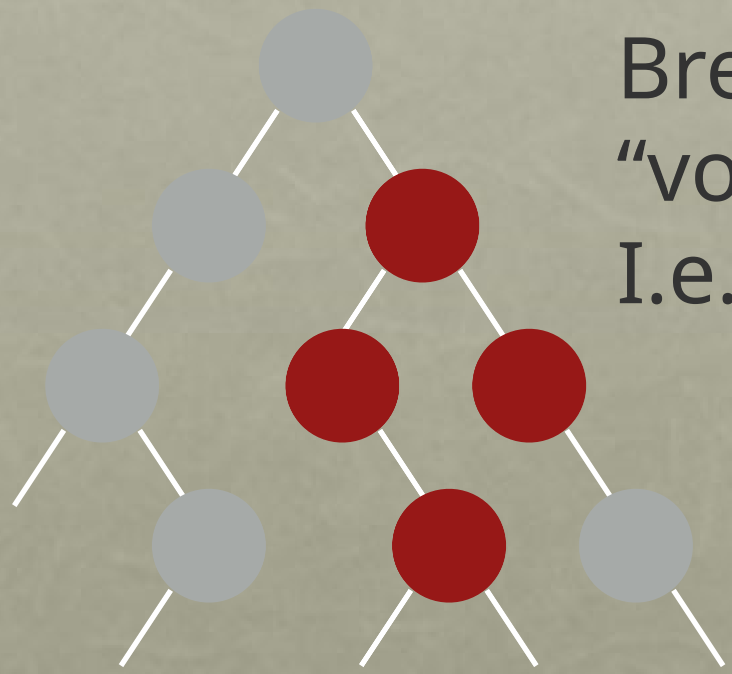
Primary goal: scalability! (many clients per server)

More reasonable semantics for concurrent file access

AFS Design

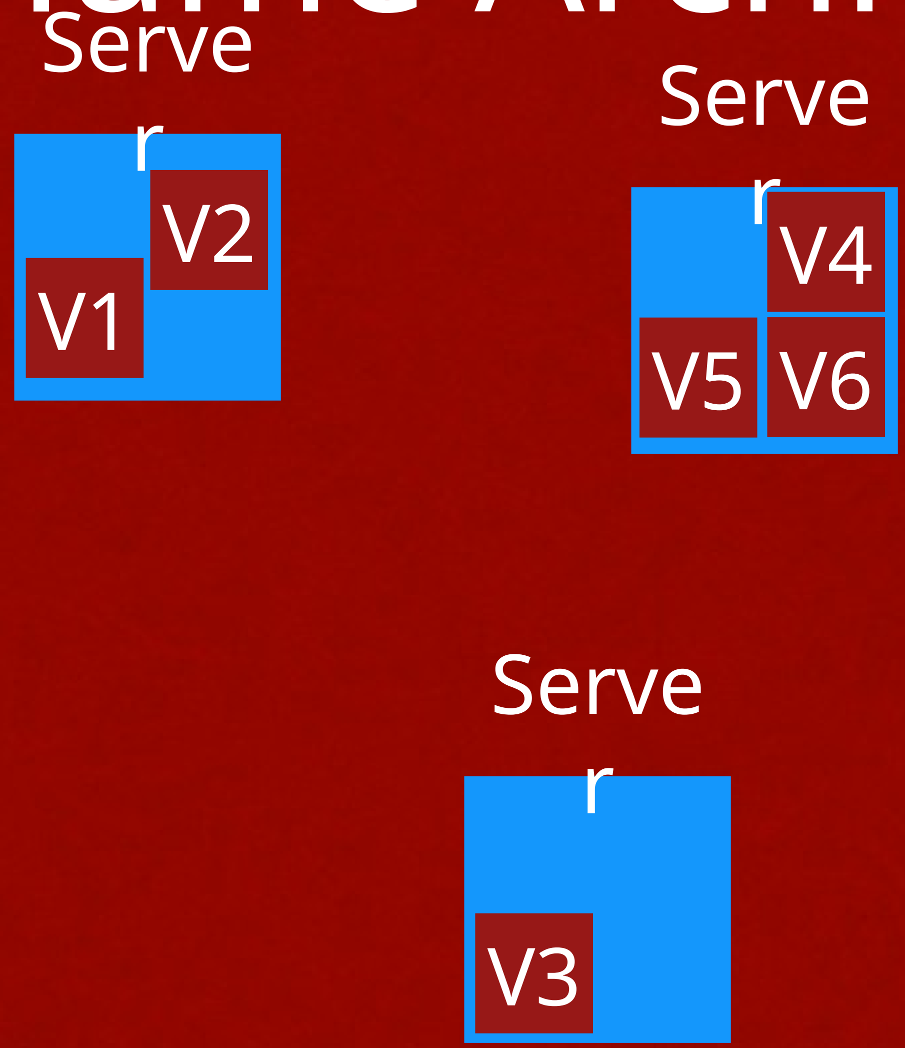
NFS: Server exports local FS

AFS: Directory tree stored across many server machines (helps scalability!)



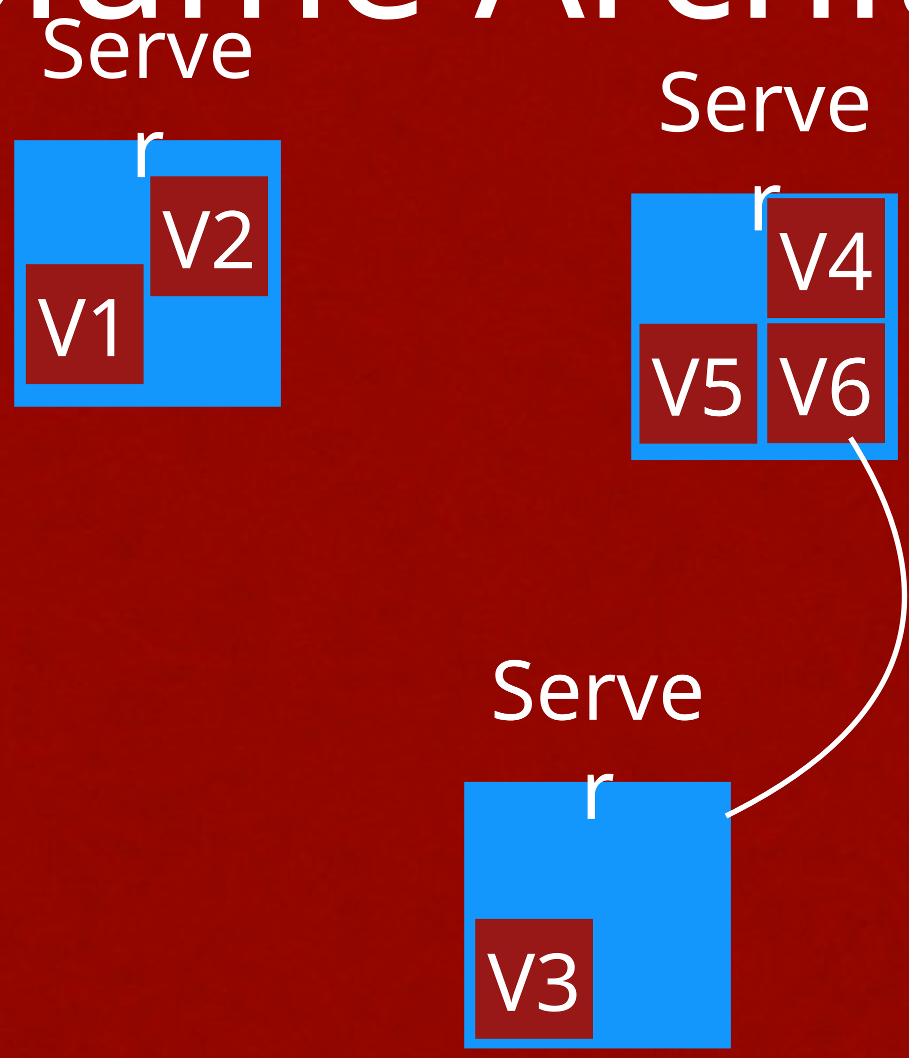
Break directory tree into "volumes"
I.e., partial sub trees

Volume Architecture



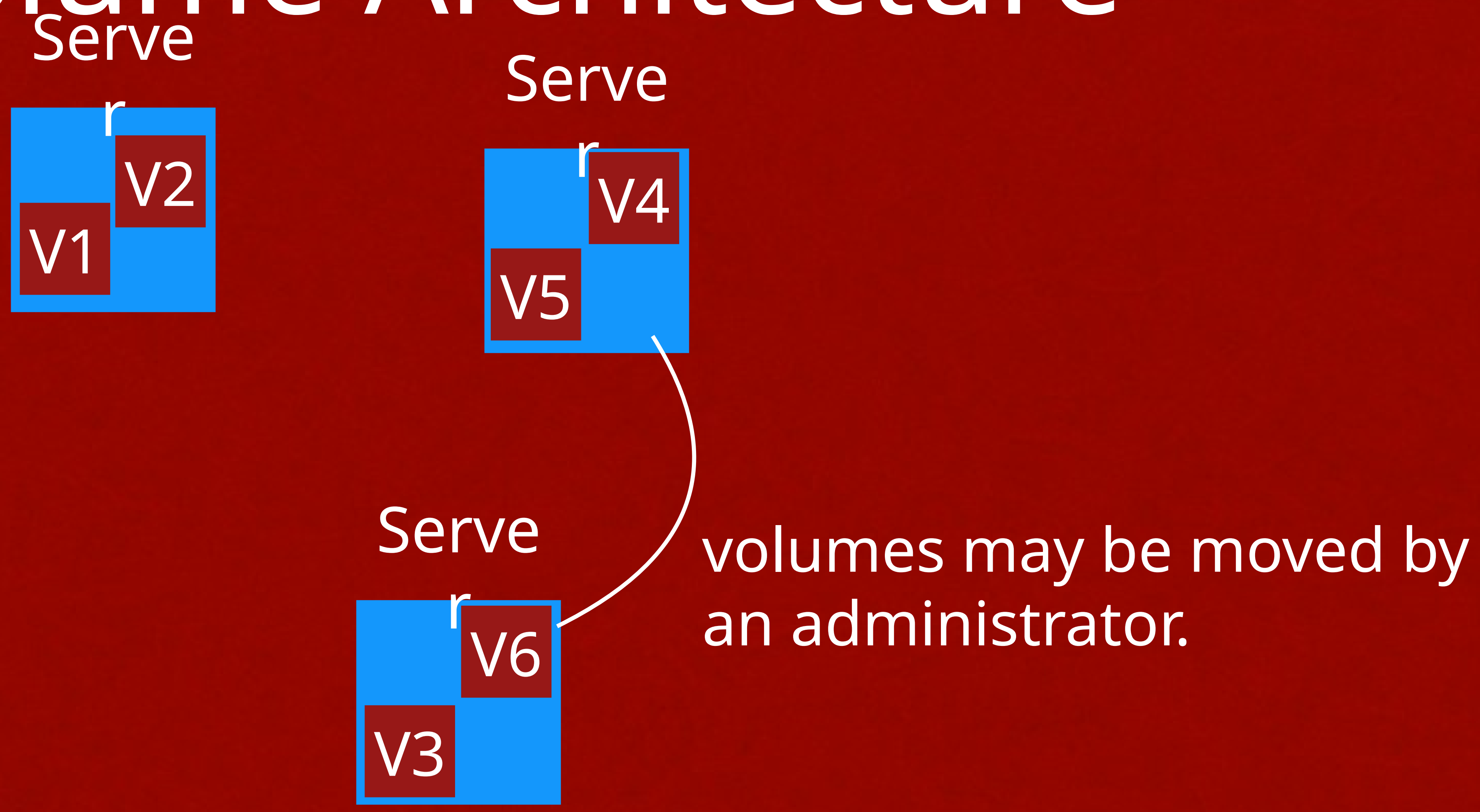
collection of servers store different volumes that together form directory tree

Volume Architecture

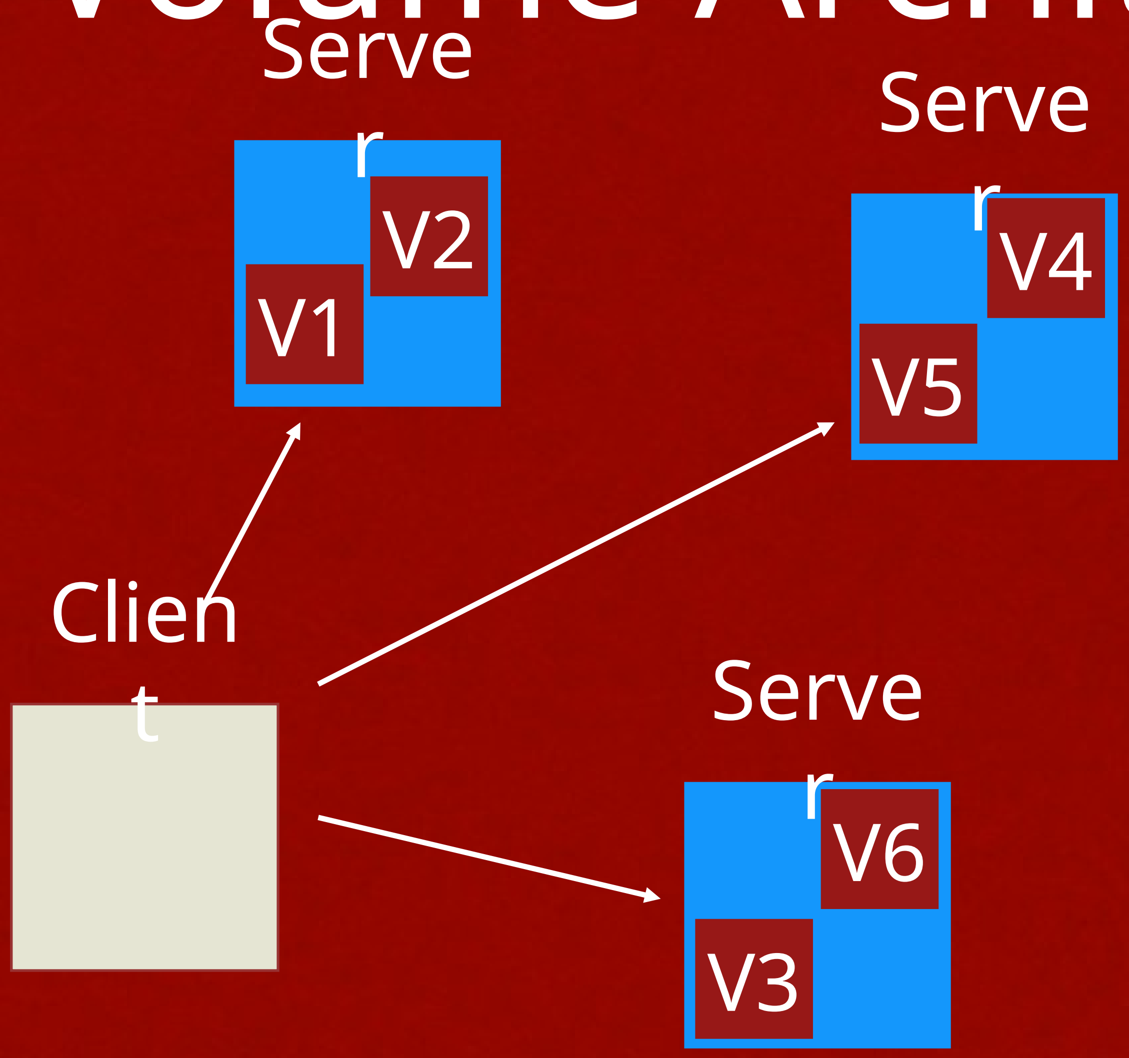


volumes may be moved by an administrator.

Volume Architecture



Volume Architecture



Client library gives seamless view of directory tree by automatically finding volumes

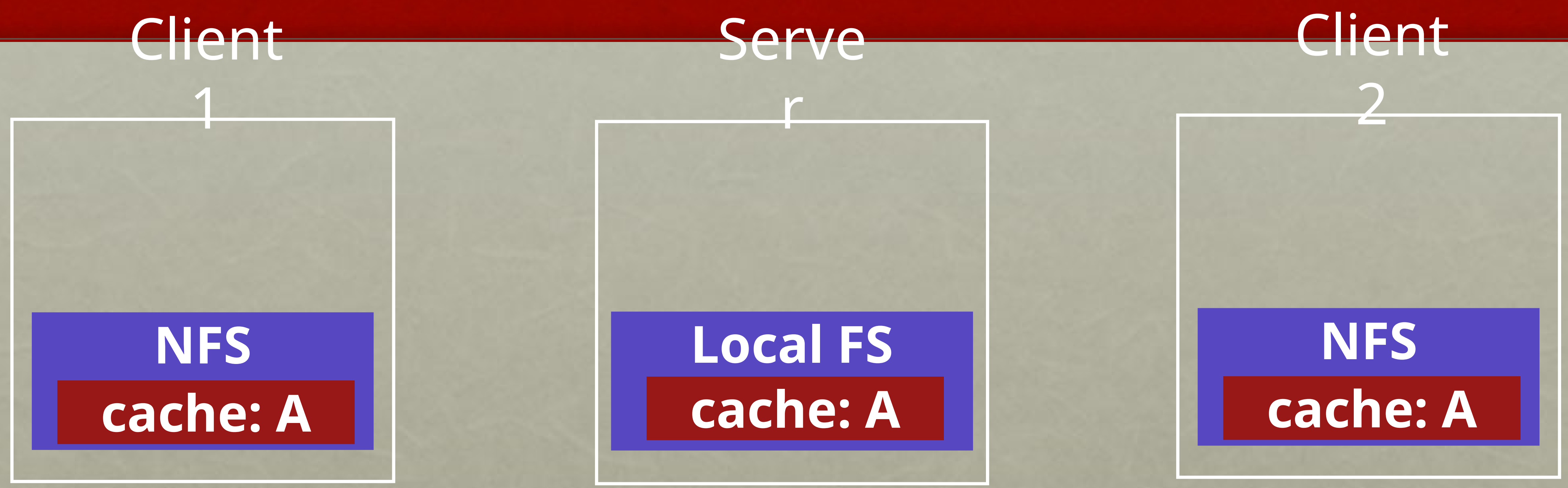
Communication via RPC
Servers store data in local file systems

AFS Cache Consistency

Update visibility

Stale cache

Update Visibility



Update Visibility

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

Update Visibility

NFS solution is to flush blocks

- on close()
- other times too – e.g., when low on memory

Problems

- flushes not atomic (one block at a time)
- two clients flush at once: mixed data

Update Visibility

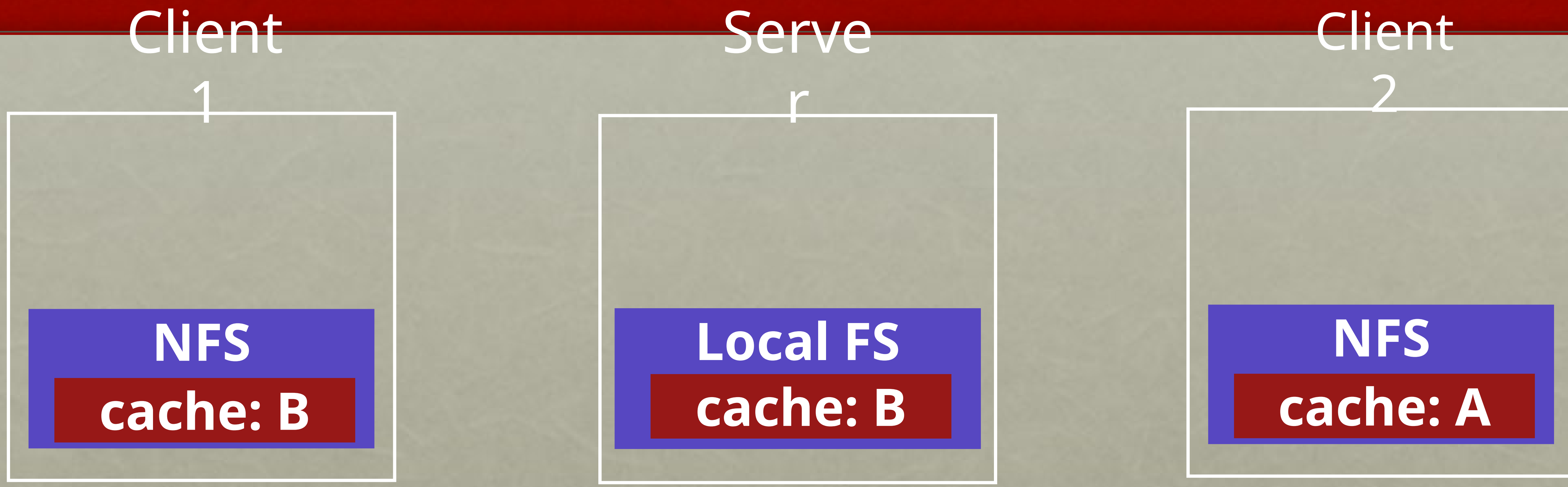
AFS solution:

- also flush on close
- buffer **whole files** on local disk; update file on server atomically

Concurrent writes?

- Last writer (i.e., last file closer) wins
- Never get mixed data on server

Cache Consistency



“Stale Cache” problem: client 2 doesn’t have latest

Stale Cache

NFS rechecks cache entries compared to server before using them, assuming check hasn't been done "recently"

How to determine how recent? (about 3 seconds)

"Recent" is too long?

client reads old data

"Recent" is too short?

server overloaded with stats

Stale Cache

AFS solution: Tell clients when data is overwritten

- Server must remember which clients have this file open right now

When clients cache data, ask for “callback” from server if changes

- Clients can use data without checking all the time

Server no longer stateless!

Callbacks: Dealing with STATE

What if client crashes?

What if server runs out of memory?

What if server crashes?

Client Crash

What should client do after reboot?
(remember cached data can be on disk too…)

Concern? may have missed notification that cached copy changed

Option 1: evict everything from cache

Option 2: ??? recheck entries before using

Low Server Memory

Strategy: tell clients you are dropping their callback

What should client do?

Option 1: Discard entry from cache

Option 2: ???

Mark entry for recheck

Server Crashes

What if server crashes?

Option: tell all clients to recheck all data before next read

Handling server and client crashes without inconsistencies or race conditions is very difficult…

Prefetching

AFS paper notes: "the study by Ousterhout *et al.* has shown that most files in a 4.2BSD environment are read in their entirety."

What are the implications for client prefetching policy?

Aggressively prefetch whole files.

Whole-File Caching

Upon open, AFS client fetches whole file (even if huge), storing in local memory or disk

Upon close, client flushes file to server (if file was written)

Convenient and intuitive semantics:

- AFS needs to do work only for open/close
    - Only check callback on open, not every read

- reads/writes are local

- Use same version of file entire time between open and close

AFS Summary

State is useful for **scalability**, but makes handling crashes hard

- Server tracks callbacks for clients that have file cached
- Lose callbacks when server crashes…

Workload drives design: **whole-file caching**

- More intuitive semantics (see version of file that existed when file was opened)

AFS vs nfs Protocols

Can you summarize the consistency semantics provided by NFSv2?

Time	Client A	Client B	Server Action?
0	fd = open("file A");		
10	read(fd, block1);		
20	read(fd, block2);		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	
50		write(fd, block1);	
60	read(fd, block1);		
70		close(fd);	
80	read(fd, block1);		
81	read(fd, block2);		
90	close(fd);		
100	fd = open("fileA");		
110	read(fd, block1);		
120	close(fd);		

When will server be contacted for NFS? For AFS?

Nfs Protocol

Time	Client A	Client B	Server Action?
0	fd = open("file A");		lookup()
10	read(fd, block1); read		read
20	read(fd, block2); read		read
30	read(fd, block1); check cache: attr expired getattr(); okay; use local		get attr
31	read(fd, block2); attr not expired, use local		
40		fd = open("file A");	lookup
50		write(fd, block1); keep local	
60	read(fd, block1); attr. expired use local data		getattr()
70		close(fd); write bl to server!	write to disk
80	read(fd, block1); attr expired: get attr. CHANGED FILE - kick out		read()
81	read(fd, block2); not in cache → read		read()
90	close(fd);		
100	fd = open("fileA");		lookup
110	read(fd, block1); attr expire; get new attr local ok		get attr
120	close(fd);		

AFS Protocol

Time	Client A	Client B	Server Action?
0	fd = open("file A");		setup callback for A
10	read(fd, block1);	send all of	file A
20	read(fd, block2); local!!		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	setup callback
50		write(fd, block1); send	all of A
60	read(fd, block1); local		
70		close(fd); send back	changes of A
80	read(fd, block1); local		break callbacks
81	read(fd, block2); local		
90	close(fd); nothing changed		
100	fd = open("fileA"); no callback!! need to	fetch A again	
110	read(fd, block1);		
120	close(fd);	send A	