DISTRIBUTED SYSTEMS: NFS

Shivaram Venkataraman

CS 537, Spring 2019

ADMINISTRIVIA

Project 4a, 4b grades out. Regrade requests by end of this week

Final Exam: Everything before the last lecture

No discussion this week

Review session on Friday at 5pm → Piazza

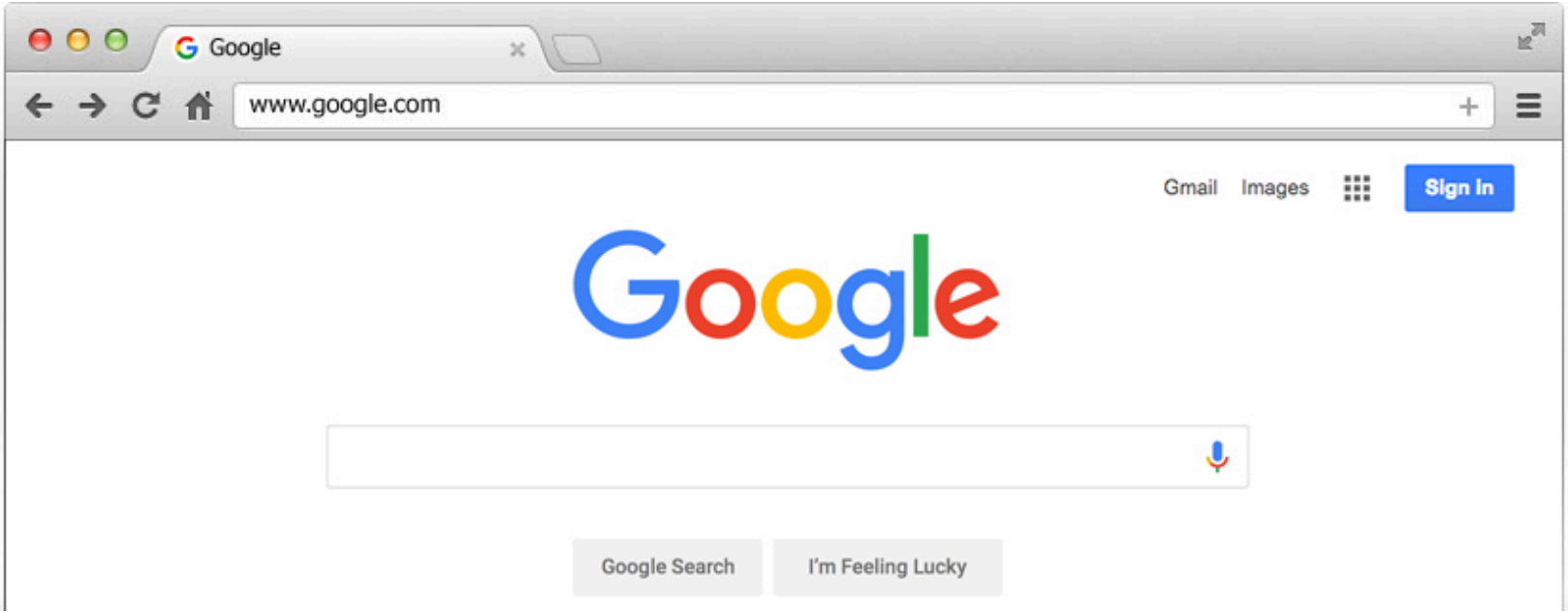
AGENDA / LEARNING OUTCOMES

How to design a distributed file system that can survive partial failures?

What are consistency properties for such designs?

RECAP

DISTRIBUTED SYSTEMS

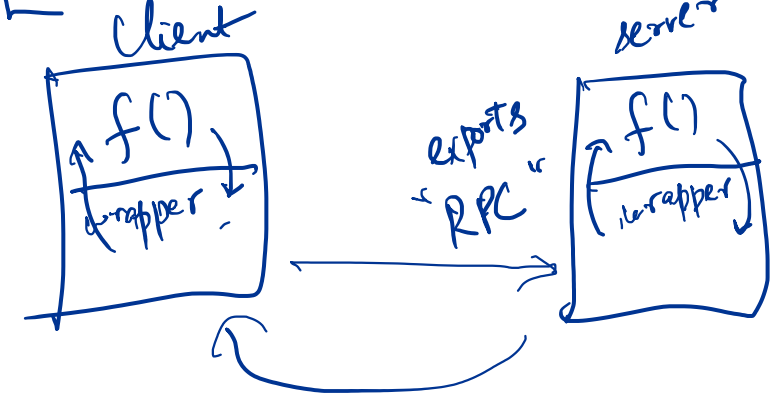


COMMUNICATION OVERVIEW

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC



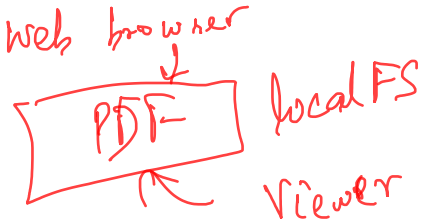
Partial failure

↳ some component of system fail. Temporary? restarted?

Machines failing, messages dropping

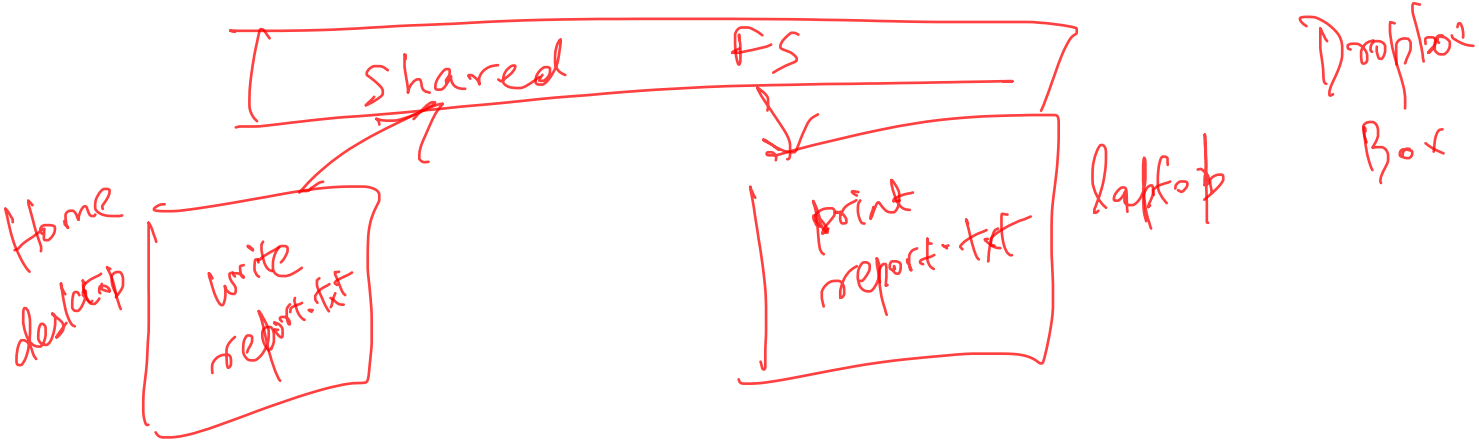
DISTRIBUTED FILE SYSTEMS

DISTRIBUTED FILE 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

1- Transparent access
   - can't tell accesses are over the network
   - normal UNIX semantics

Applications should not be aware

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

3- Reasonable performance?

NETWORK FILE SYSTEM: NFS → Sun Microsystems

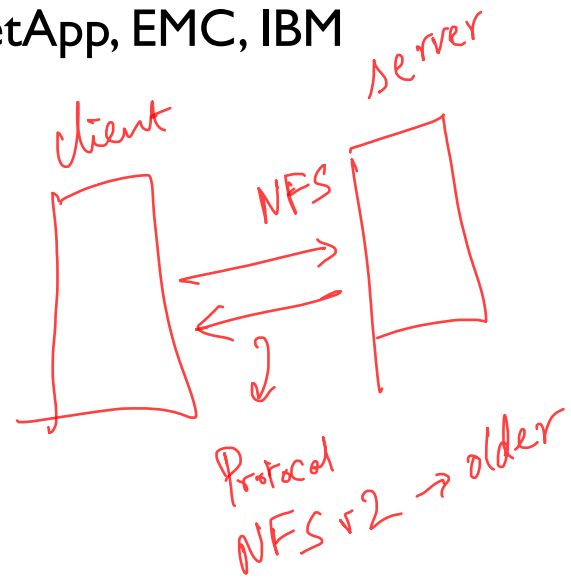
NFS: more of a protocol than a particular file system

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

AFS different protocol

We're looking at NFSv2. NFSv4 has many changes

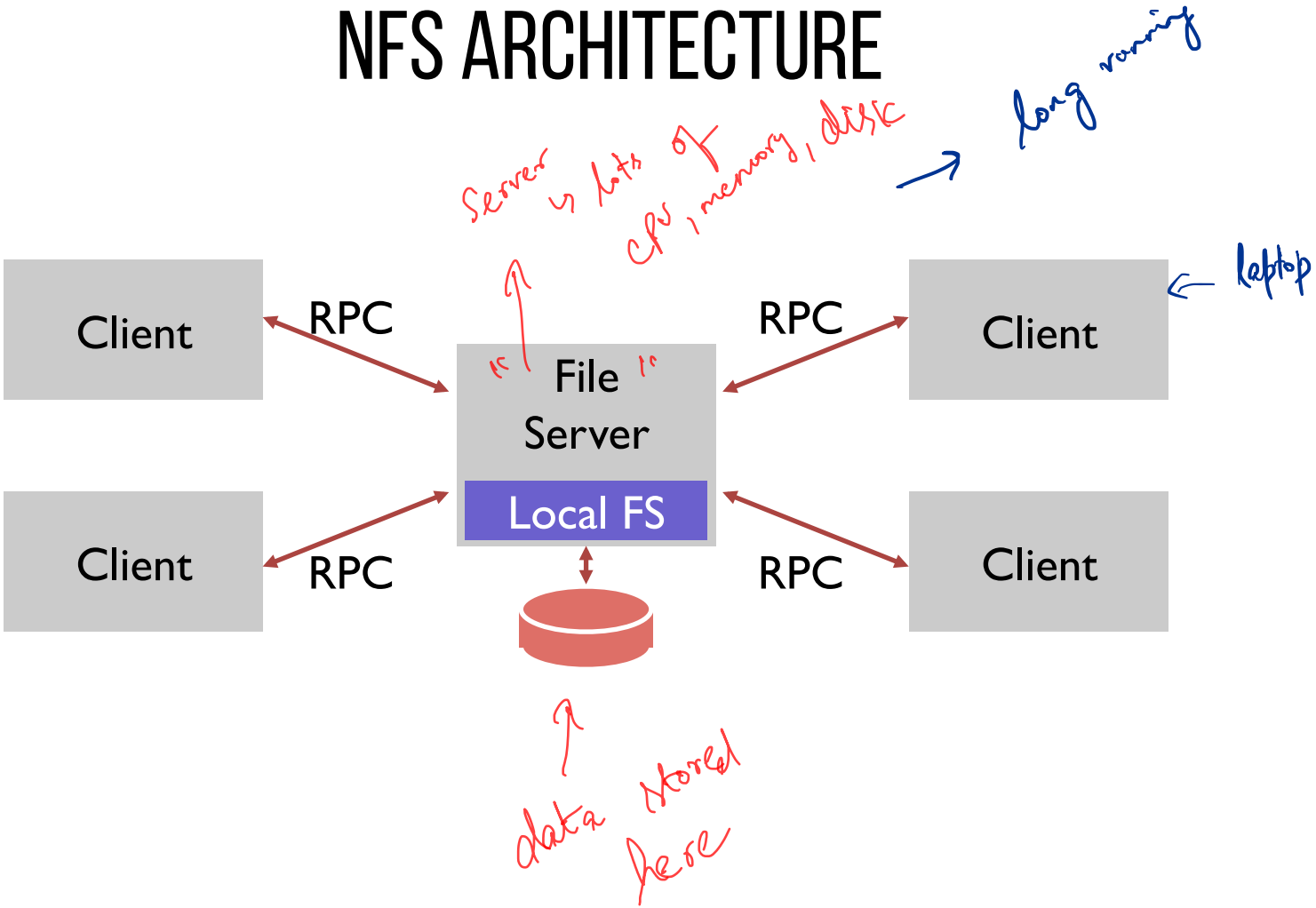
Why look at an older protocol? Simpler, focused goals

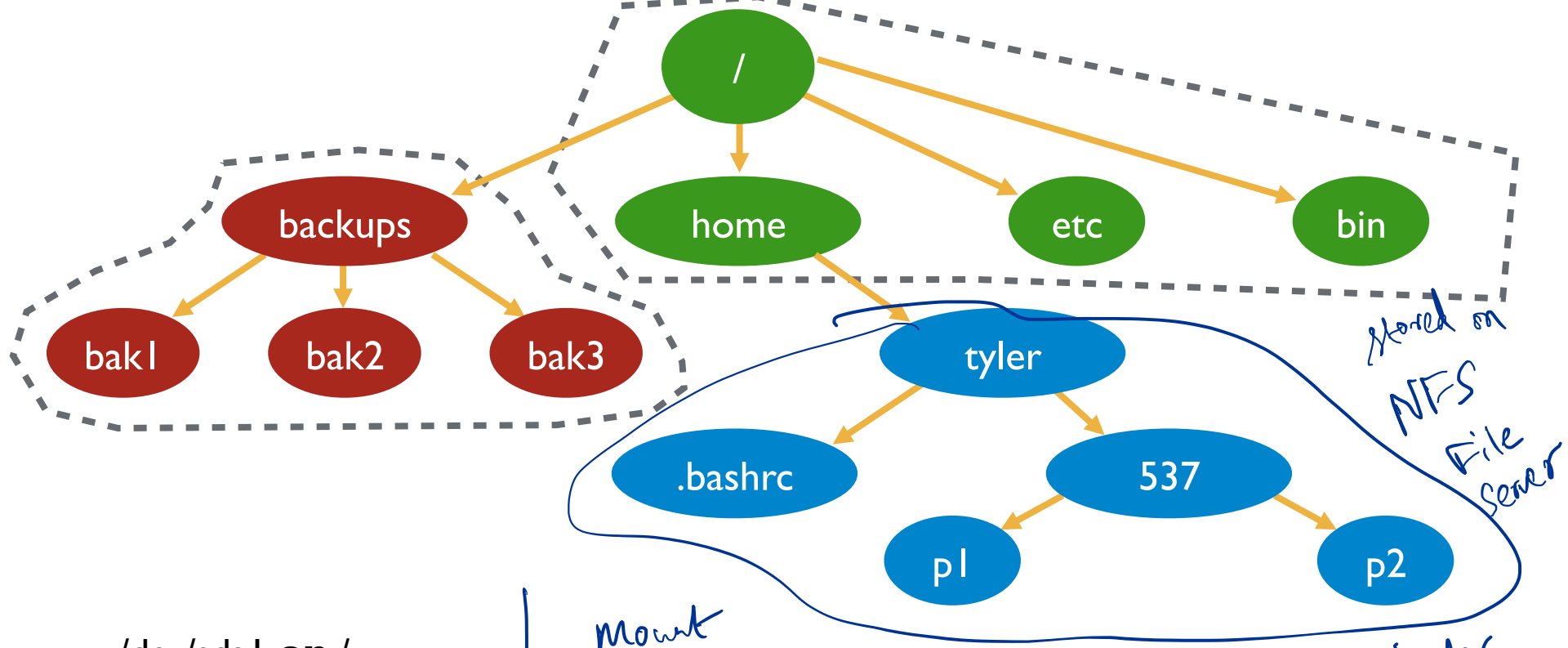


OVERVIEW

- Architecture
- Network API
- Write Buffering
- Cache

NFS ARCHITECTURE





/dev/sda1 **on** /

/dev/sdb1 **on** /backups

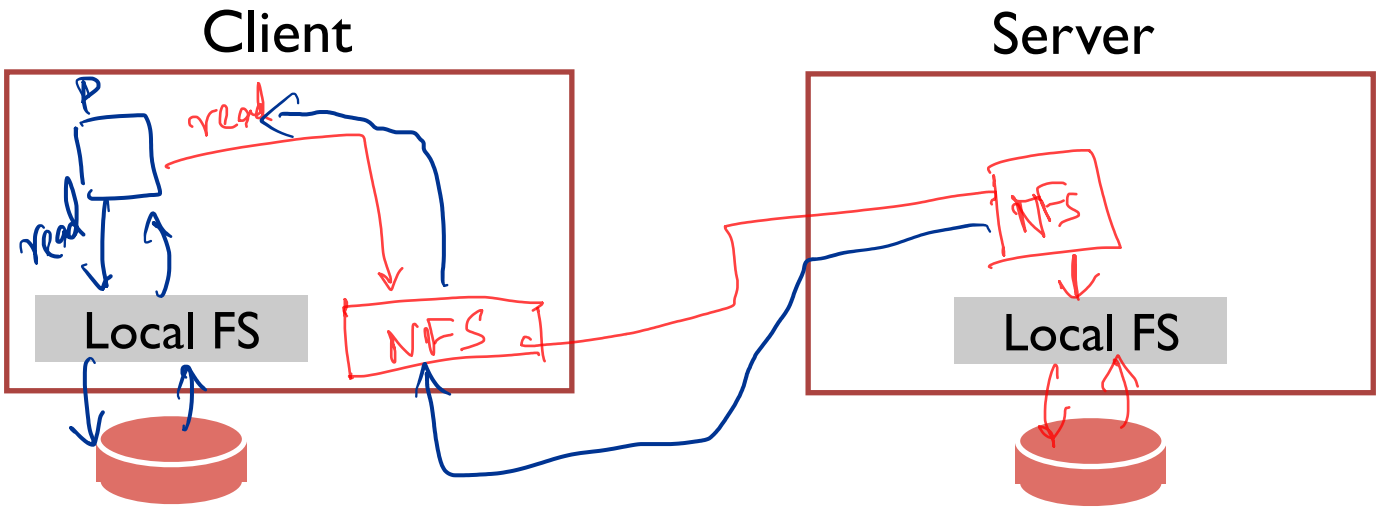
NFS **on** /home

Mount

NFS on client side

mount to a particular location

server_ip:port


Client
Server
P
read
read
Local FS
NFS
NFS
Local FS

OVERVIEW

~~Architecture~~

Network API

Write Buffering

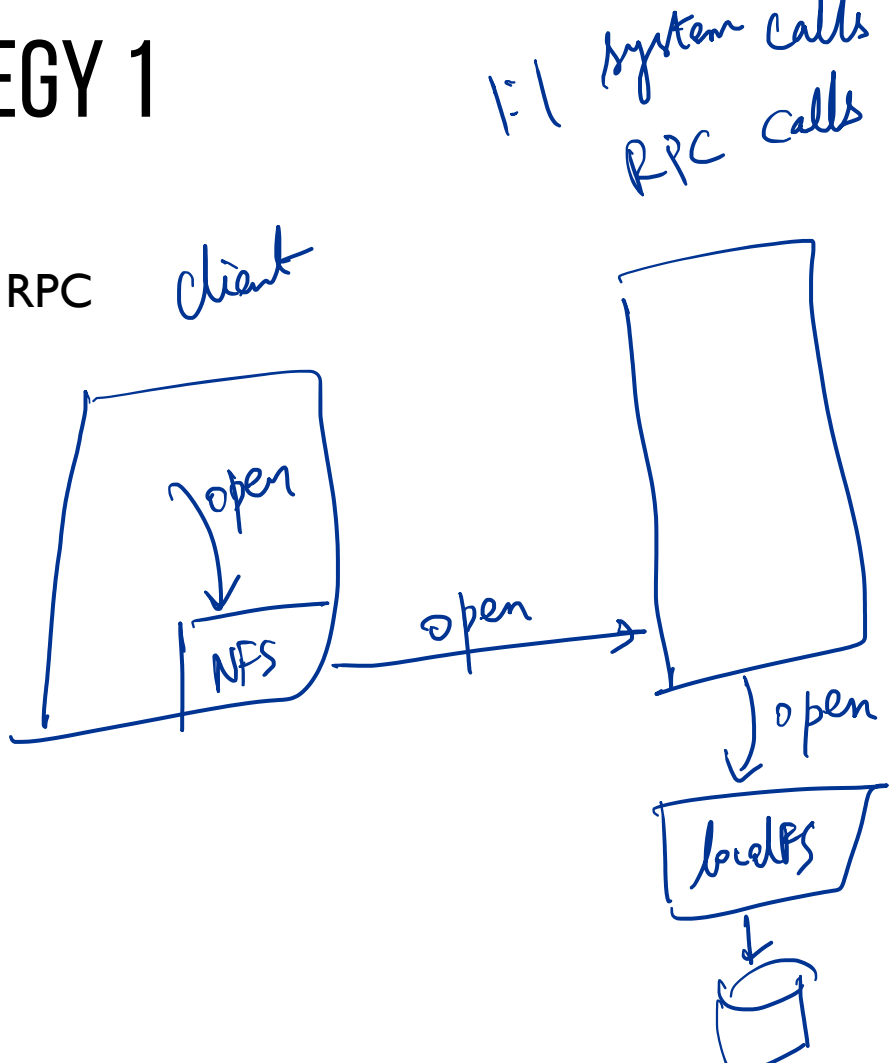
Cache

STRATEGY 1

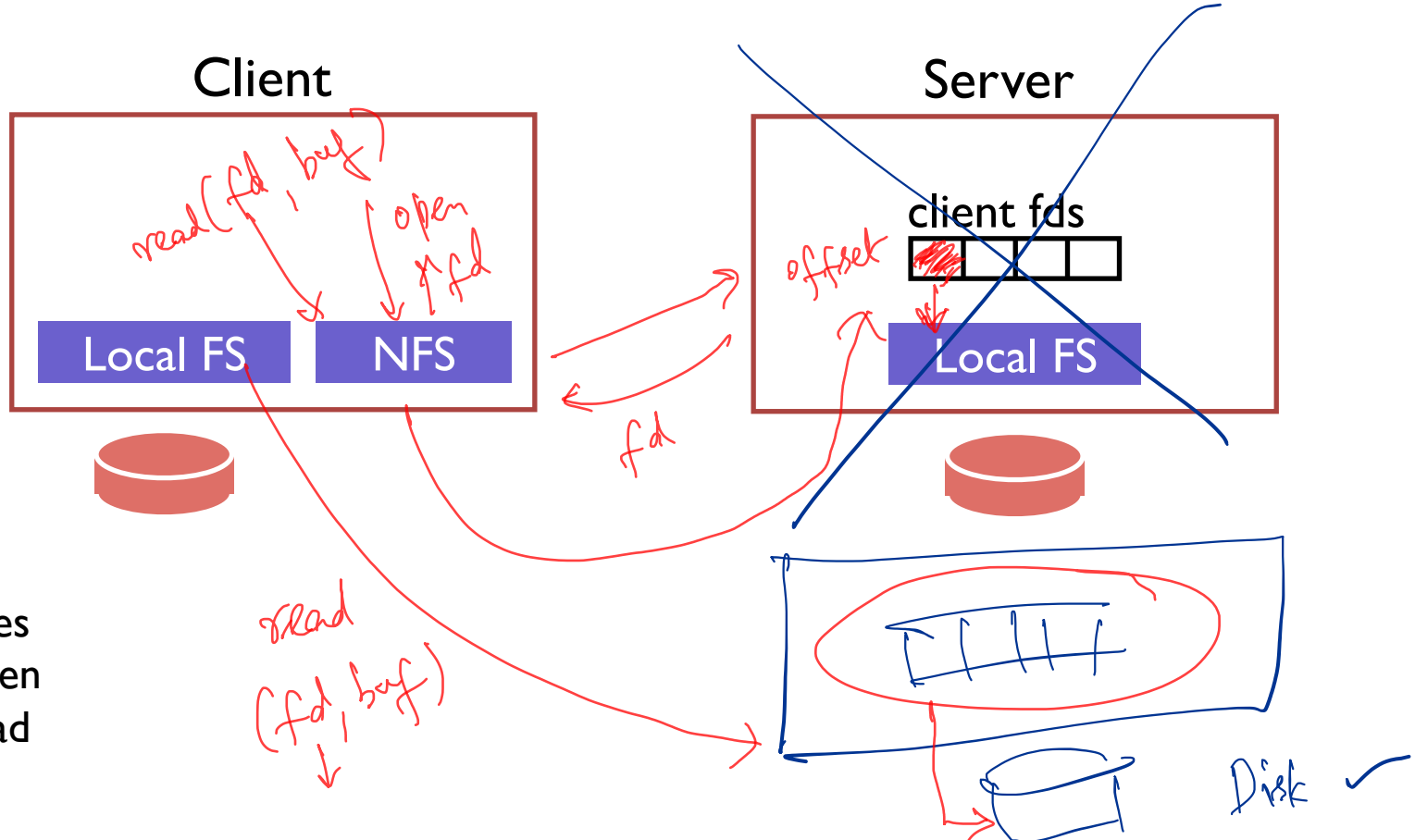
Attempt: Wrap regular UNIX system calls using RPC

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



Examples
- open
- read

STRATEGY 1: WHAT ABOUT CRASHES

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

Server crash!

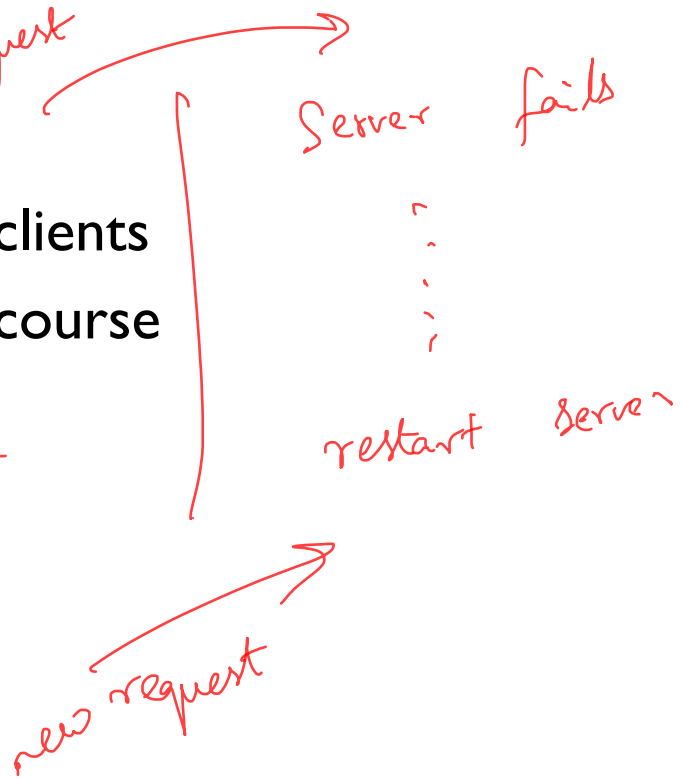
POTENTIAL SOLUTIONS

1. Run some crash recovery protocol upon reboot
   - Complex → avoid
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



↳ Data

easy to handle failures

STRATEGY 2: PUT ALL INFO IN REQUESTS

"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);

pread("/tmp/a.txt", buf, 4096, 0);

pread("/tmp/a.txt", buf1, 4096, 4096);

State is now in the request

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

Pros? Retry your request

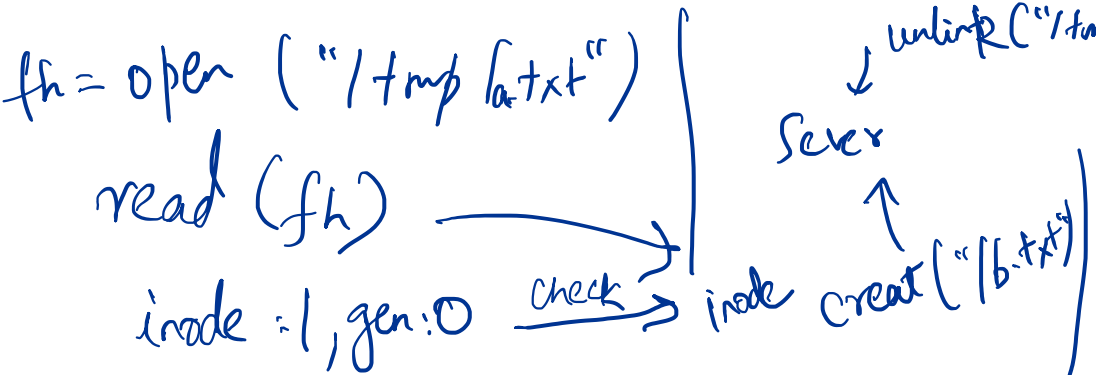
Cons? File traversal is repeated on every request

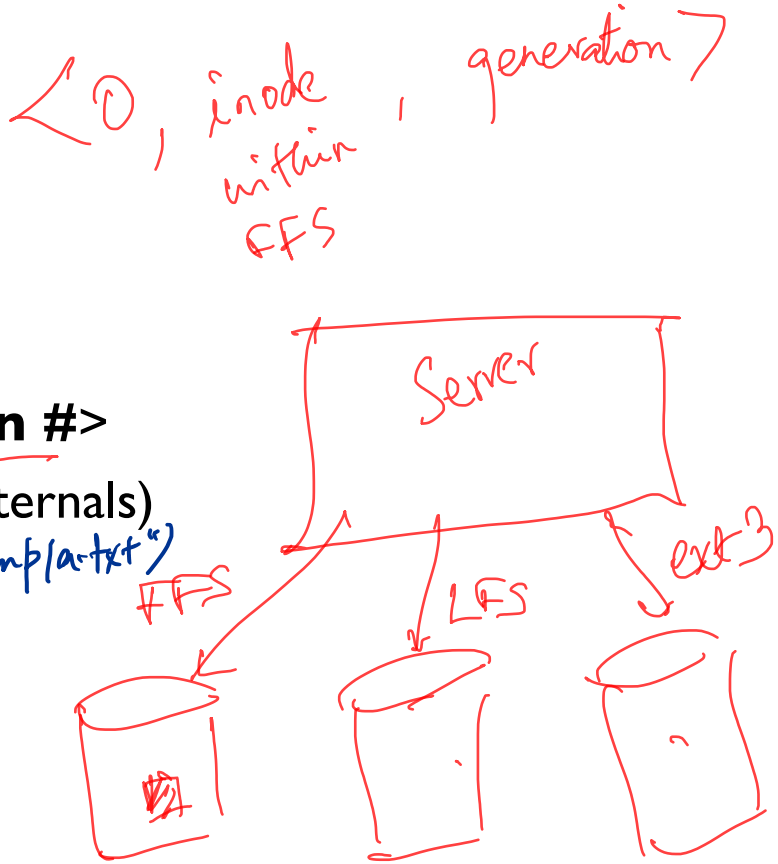
STRATEGY 3: FILE HANDLES

```
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)





```
NFSPROC_GETATTR
  expects: file handle
  returns: attributes
NFSPROC_SETATTR
  expects: file handle, attributes
  returns: nothing
NFSPROC_LOOKUP
  expects: directory file handle, name of file/directory to look up
  returns: file handle
NFSPROC_READ
  expects: file handle, offset, count
  returns: data, attributes
NFSPROC_WRITE
  expects: file handle, offset, count, data
  returns: attributes
NFSPROC_CREATE
  expects: directory file handle, name of file, attributes
  returns: nothing
NFSPROC_REMOVE
  expects: directory file handle, name of file to be removed
  returns: nothing
NFSPROC_MKDIR
  expects: directory file handle, name of directory, attributes
  returns: file handle
NFSPROC_RMDIR
  expects: directory file handle, name of directory to be removed
  returns: nothing
NFSPROC_READDIR
  expects: directory handle, count of bytes to read, cookie
  returns: directory entries, cookie (to get more entries)
```

stat

size
modified
n links

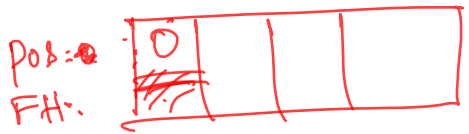
mount
↳ file handle for /

Client	Server

fd = open("/foo", ...);
Send LOOKUP (rootdir FH, "foo")

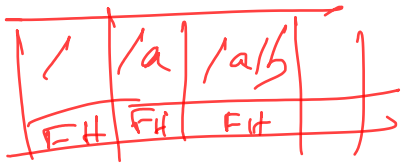
Receive LOOKUP request
look for "foo" in root dir
return foo's FH + attributes

Receive LOOKUP reply
allocate file desc in open file table
store foo's FH in table
store current file position (0)
return file descriptor to application

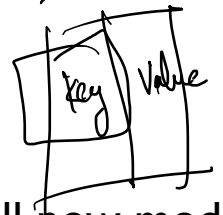


Traversal

/a/b/tmp.txt
/ → Lookup (a)
/a → Lookup (b)
/a/b → Lookup (tmp)



Blob stores or KV stores



BUNNY 21

TODO.txt paper.pdf

https://tinyurl.com/cs537-sp19-bunny21

latency

We'll now model the time of certain operations in NFS. The only costs to worry about are network costs. Assume any "small" message takes S units of time from one machine to another, whereas a "bigger" message (e.g., size of a disk block) takes B units. If a message is larger than 4KB, it should take proportionally longer (2B for 8KB)

↳ bandwidth 775

1. How long does it take to open a 100-block (400 KB) file called /a/b/c.txt for the first time? (assume root directory file handle is already available) *

read(fd, buf, size) → 4K

Client Server

open /a/b.txt

Lookup("/a", b.txt)

Lookup("/", a) S S

cache FH of /a

Lookup("/a", b) S

-1 S

Lookup("/a/b", c.txt) S S

6S

2. How long does it take to read the whole file?

Pread (FH, 0) S

B

Pread (FH, 4K) S

B

Block oriented

100B + 100S

CAN NFS PROTOCOL INCLUDE APPEND?

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

append(fh, buf, size);
```

append (FH , "BB" , 2)

LOOKUP one for step in path

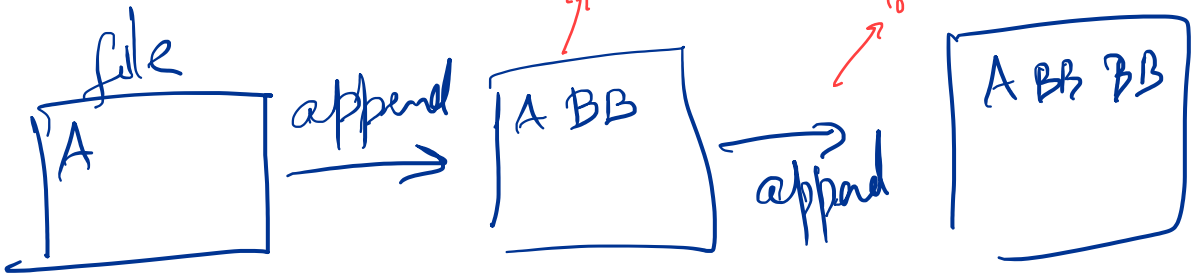
block level READ

block level WRITE

PWRITE VS APPEND

pwrite(file, "BB", 2, 2);

append(file, "BB");

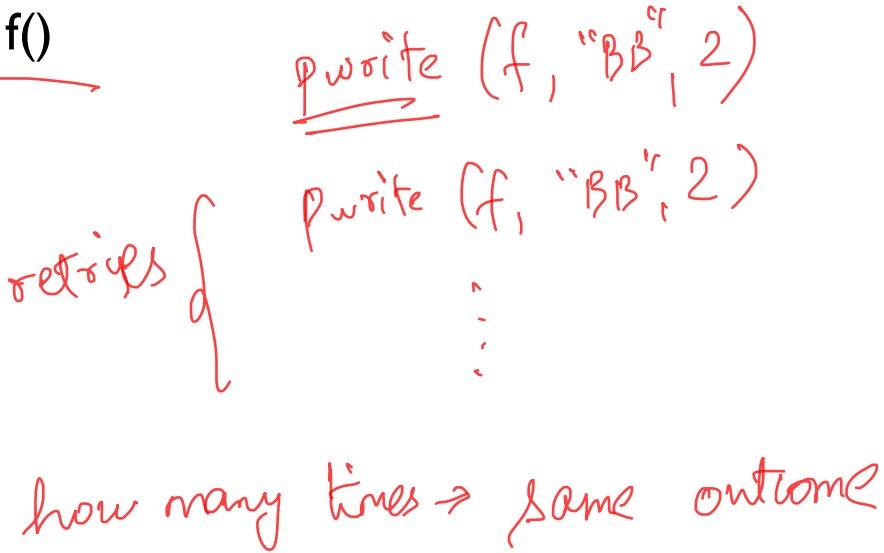


IDEMPOTENT OPERATIONS

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()



CRASHES WITH IDEMPOTENT OPERATIONS

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

Server crash!

WHAT OPERATIONS ARE IDEMPOTENT?

Idempotent

- any sort of read that doesn't change anything
- pwrite offset, contents

Not idempotent

- append

What about these?

- mkdir
- creat

OVERVIEW

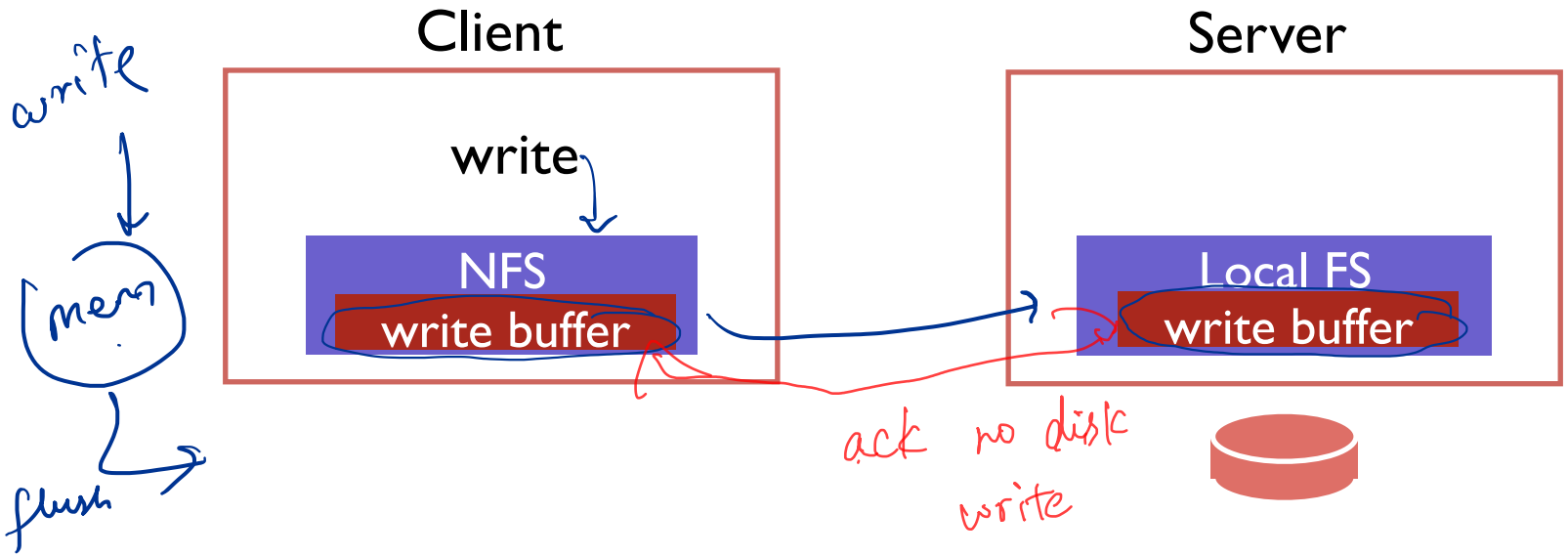
~~Architecture~~

~~Network API~~

Write Buffering

Cache

WRITE BUFFERS



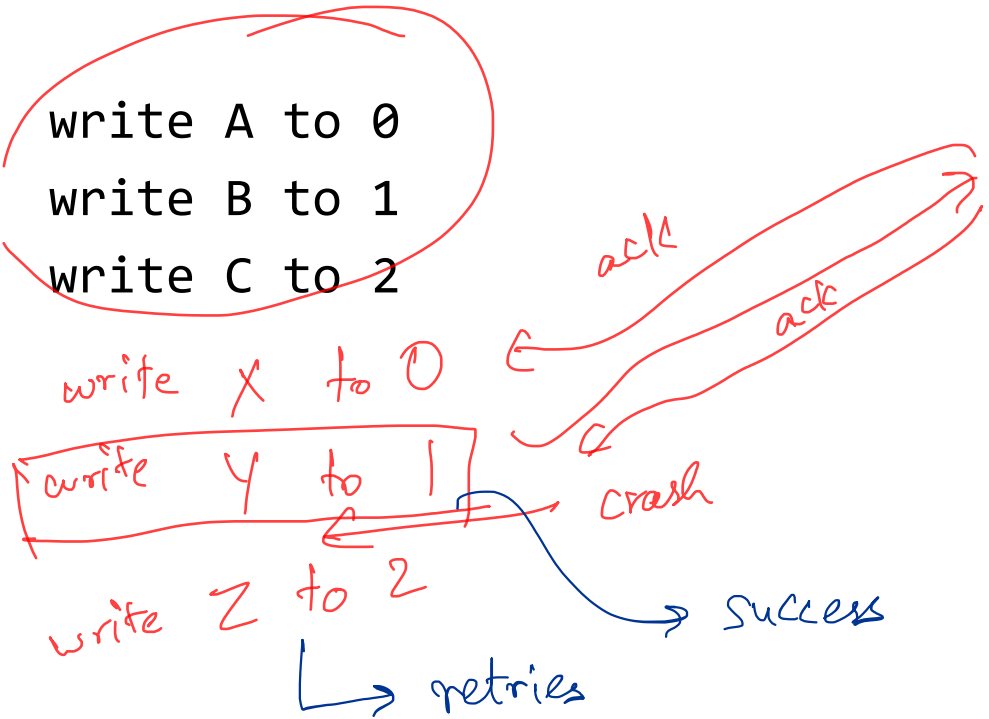
Server acknowledges write before write is pushed to disk;

What happens if server crashes?

SERVER WRITE BUFFER LOST

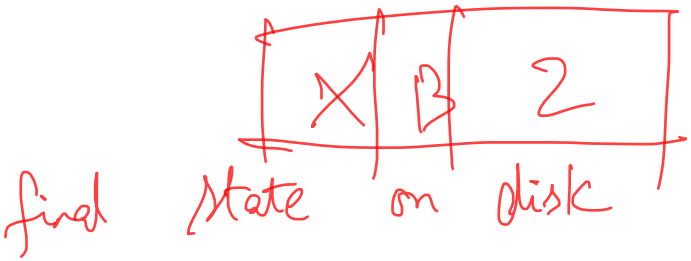
client:

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



server mem: A B Z

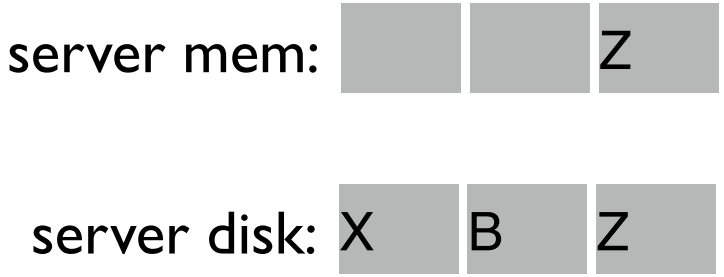
server disk: X B Z



server acknowledges write before write is pushed to disk

SERVER WRITE BUFFER LOST

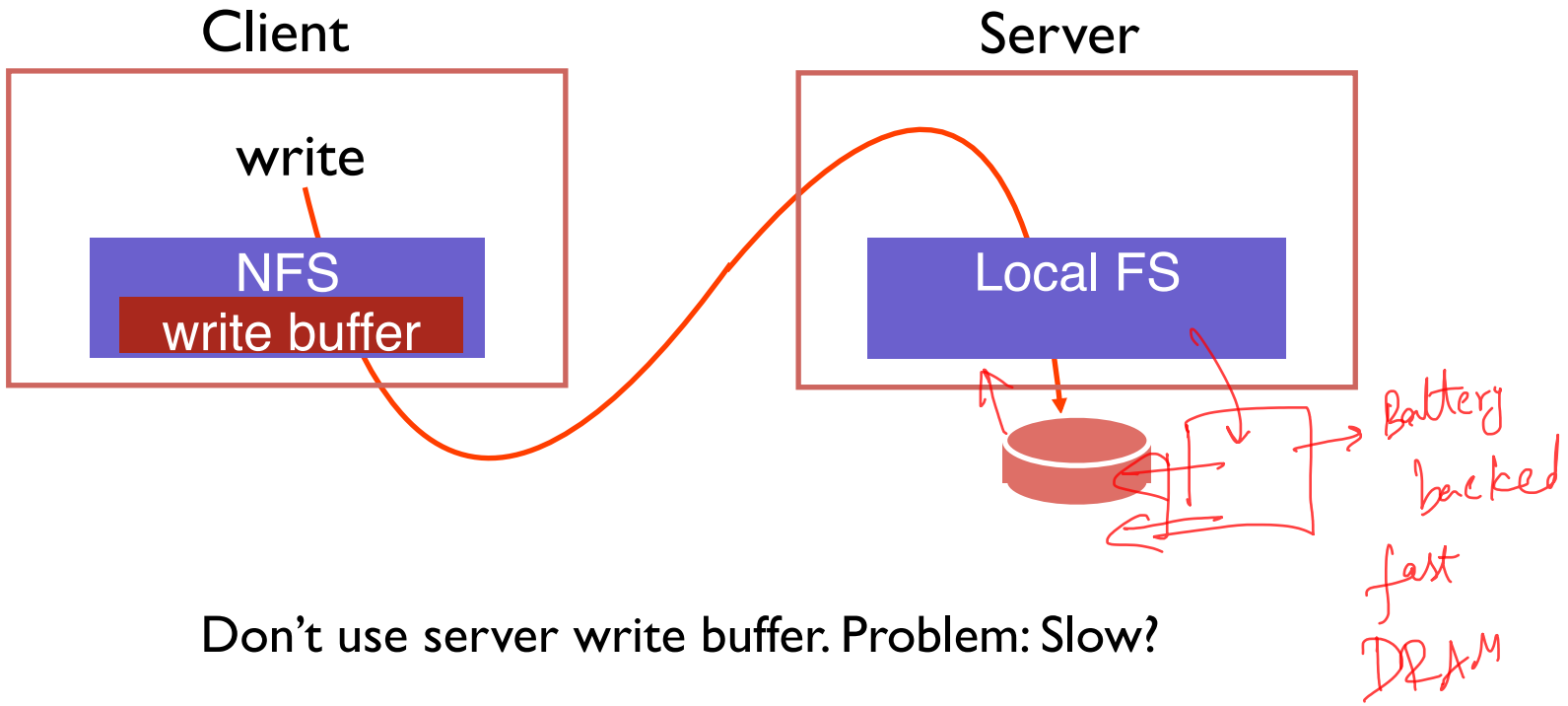
```
Client:

 write A to 0

 write B to 1

 write C to 2

 write X to 0

 write Y to 1

 write Z to 2
```



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

Solutions?

WRITE BUFFERS



Don't use server write buffer. Problem: Slow?

Use persistent write buffer (more expensive)

~~Architecture~~

~~Network API~~

~~Write Buffering~~

Cache

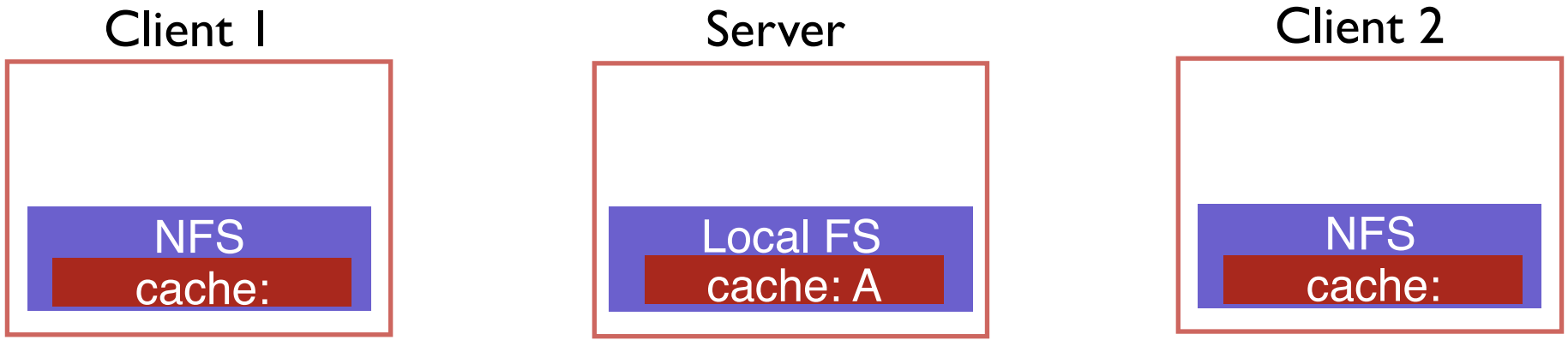
CACHE CONSISTENCY

NFS can cache data in three places:

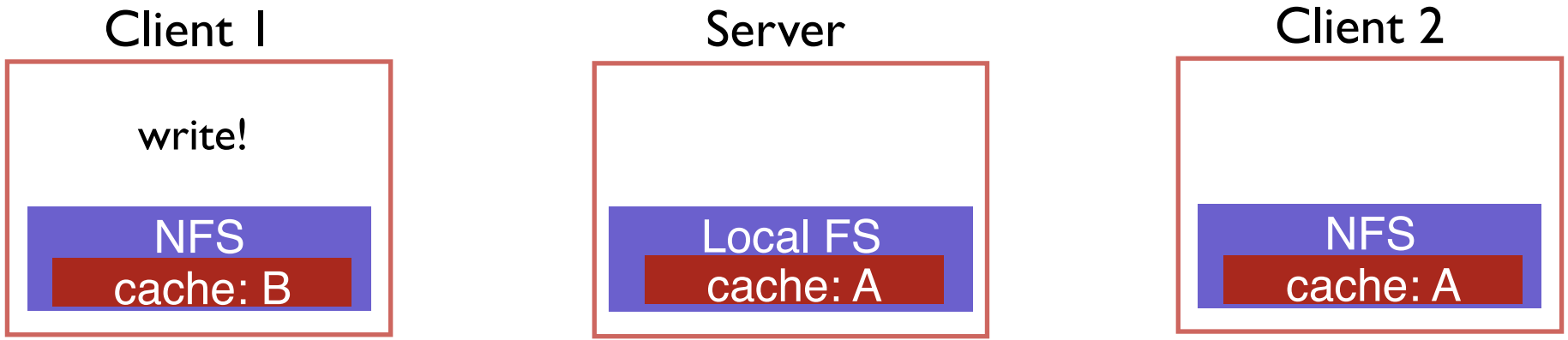
- server memory
- client disk
- client memory

How to make sure all versions are in sync?

DISTRIBUTED CACHE



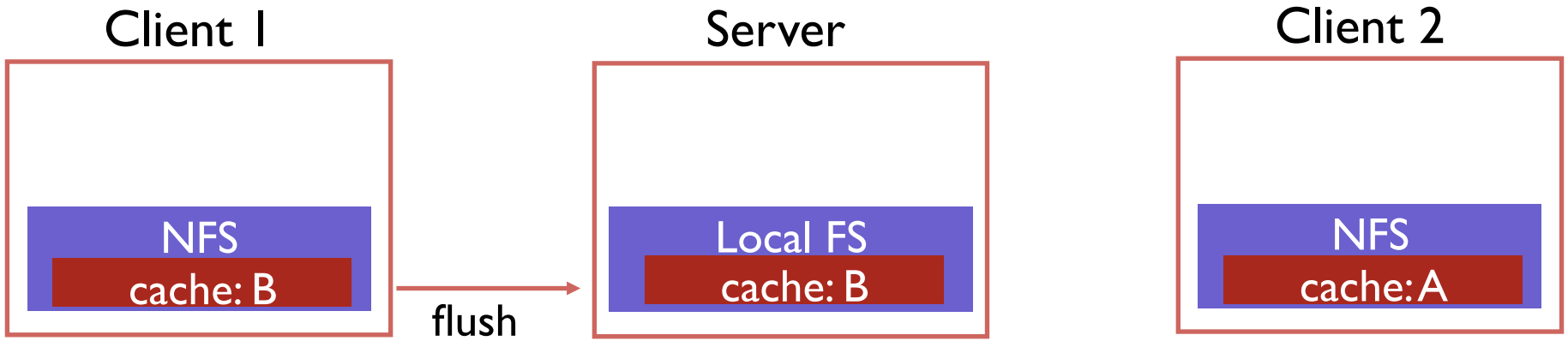
CACHE



"Update Visibility" problem: server doesn't have latest version

What happens if Client 2 (or any other client) reads data?

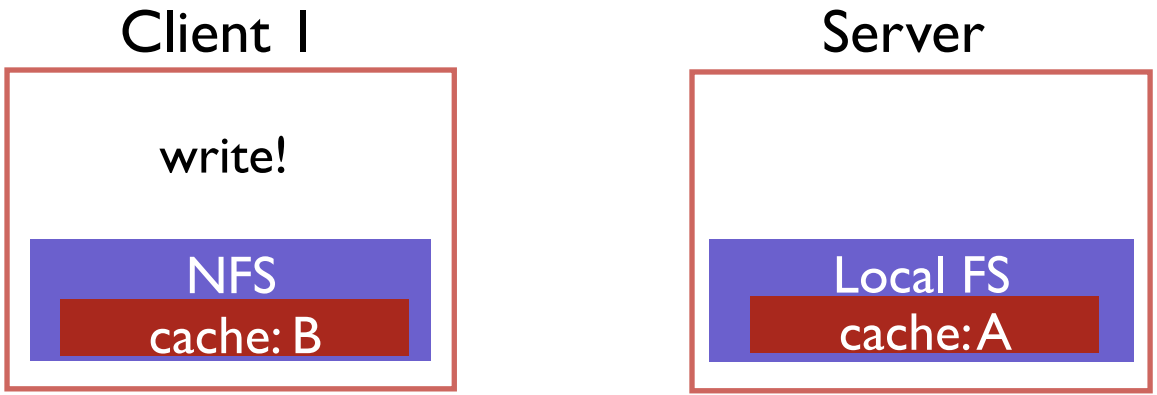
CACHE



"Stale Cache" problem: client 2 doesn't have latest version

What happens if Client 2 reads data?

PROBLEM 1: UPDATE VISIBILITY



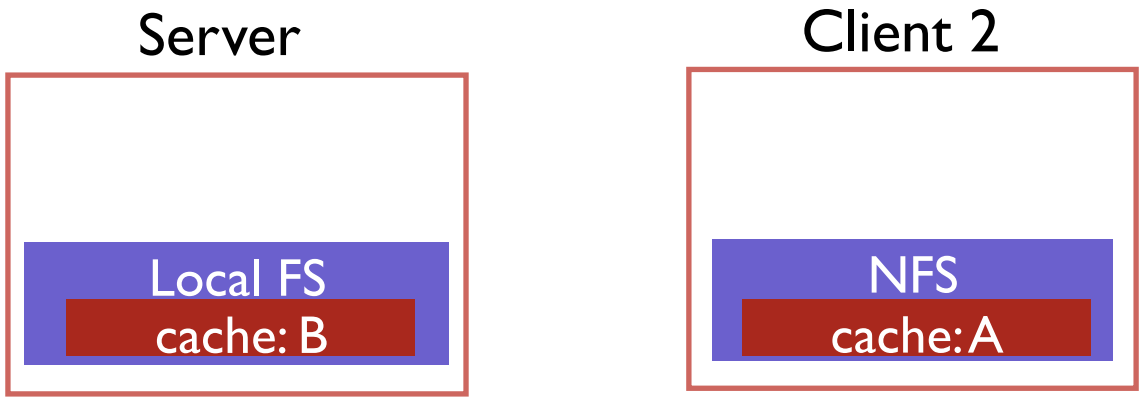
When client buffers a write, how can server (and other clients) see update?

Client flushes cache entry to server

When should client perform flush?

NFS solution: flush on fd close

PROBLEM 2: STALE CACHE

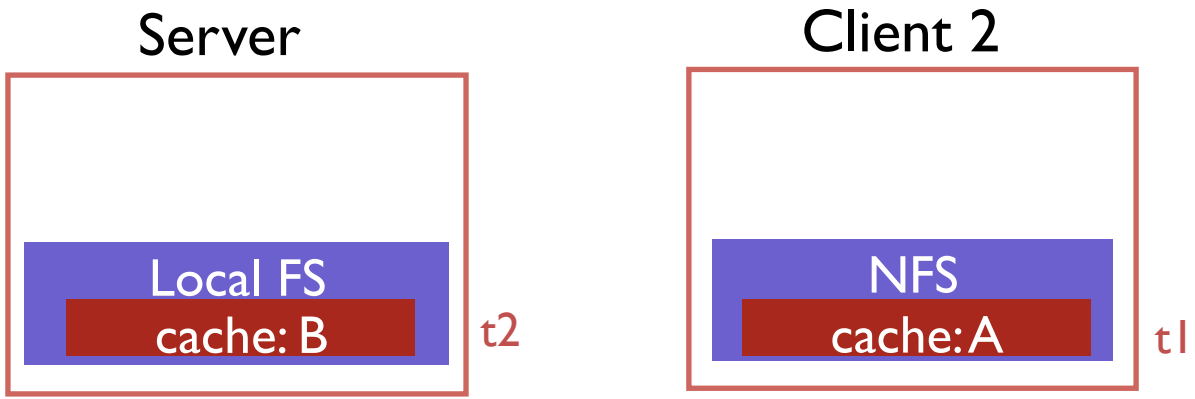


Problem: Client 2 has stale copy of data; how can it get the latest?

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)

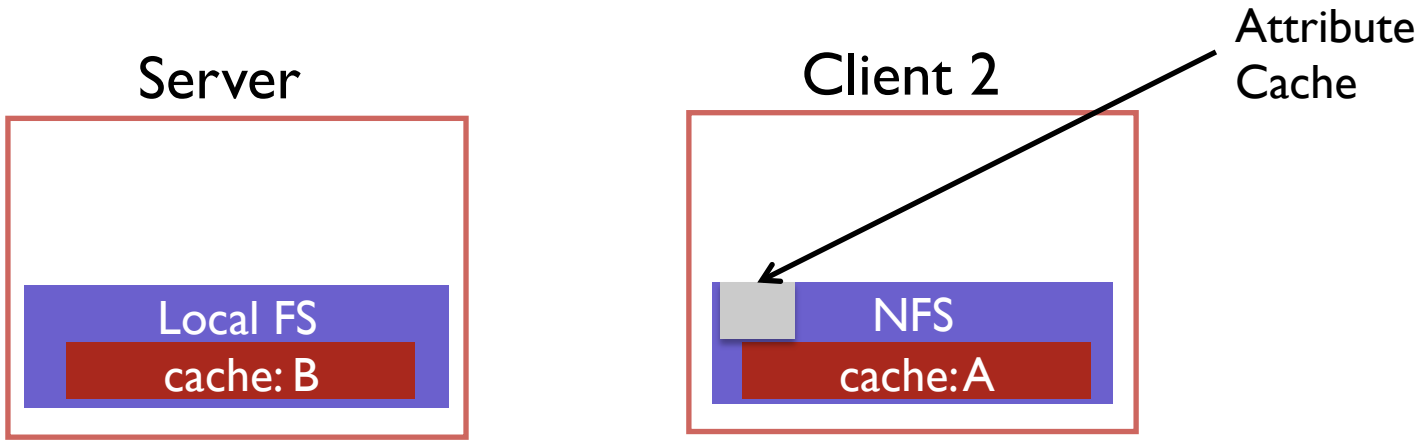
MEASURE THEN BUILD

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

Partial Solution:

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

What is the consequence?

NFS SUMMARY

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

- stateless: servers don't remember clients
- idempotent: doing things twice never hurts

Caching and write buffering is harder, especially with crashes

Problems:

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

NEXT STEPS

Next class: AFS, Wrap up, Review

No discussion this week!