PERSISTENCE: DISTRIBUTED FILE SYSTEMS (NFS + AFS)

Andrea Arpaci-Dusseau

CS 537, Fall 2019

ADMINISTRIVIA

Project 7: xv6 File systems: Improvements + Checker
- Specification Quiz – Worth Project Points
  - 7a due yesterday, 7b due Today
- Can still request project partner if needed…

Final Exam
- Friday, December 13th 7:25-9:25 pm
- Two Rooms: Last name A-H in SOC SCI 5206, I-Z in Humanities 3650
- Slightly cumulative (some T/F from Virtualization and Concurrency – 25%)

Exam Review
- Next Tuesday: You ask questions to cover by Monday at 5:00pm
- Next Wednesday discussions

AGENDA / LEARNING OUTCOMES

What is the **NFS stateless protocol**?

- What are **idempotent** operations and why are they useful?
- What state is tracked on NFS clients?

What is the **AFS protocol**?

- Why is AFS more **scalable** with more intuitive consistency model?

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, running massively parallel map-reduce

WHY GO DISTRIBUTED?

More computing power

- throughput
- latency

More storage capacity

Fault tolerance

Data sharing

NEW CHALLENGES

System failure: need to worry about partial failure

Communication failure: network links unreliable

- bit errors
- packet loss
- link failure

Individual **nodes (machines)** crash and recover

- Some of our focus today

DISTRIBUTED FILE SYSTEMS

Local FS (FFS, ext3/4, LFS):
Processes on same machine access shared files

Network FS (NFS, AFS):
Processes on different machines access shared files in same way

Many clients with single server…

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

- Scale with number of clients?

NFS: NETWORK FILE SYSTEM

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

Many companies have implemented NFS since 1980s:
Oracle/Sun, NetApp, EMC, IBM

We're looking at NFSv2
- NFSv4 has many changes

Why look at an older protocol?
- Simpler, focused goals (simplified crash recovery, stateless)
- To compare and contrast NFS with AFS

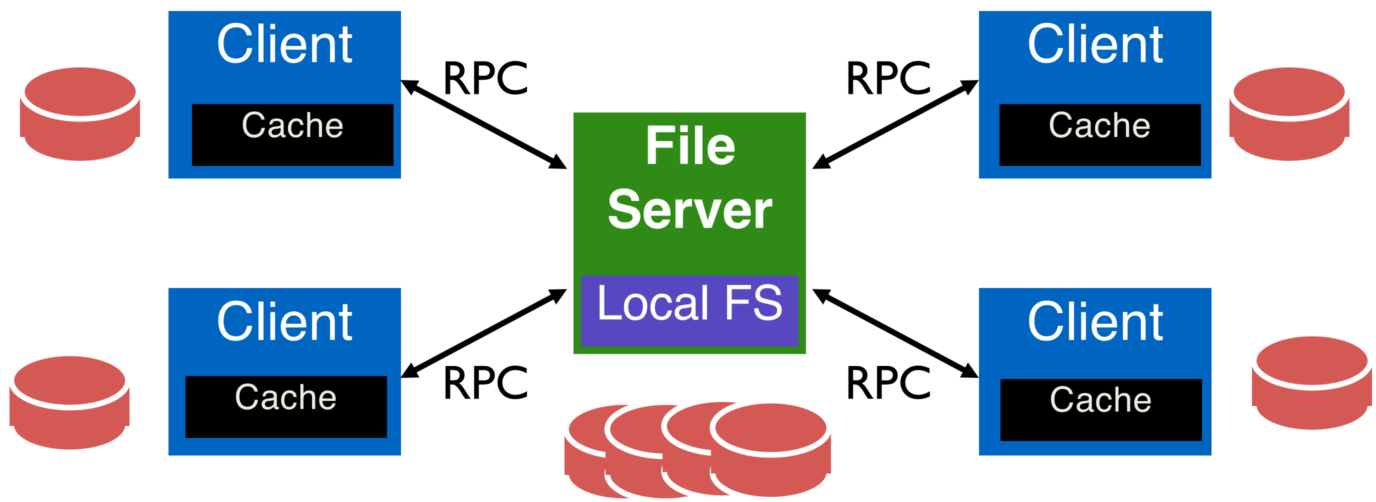
NFS OVERVIEW

Architecture

Network API

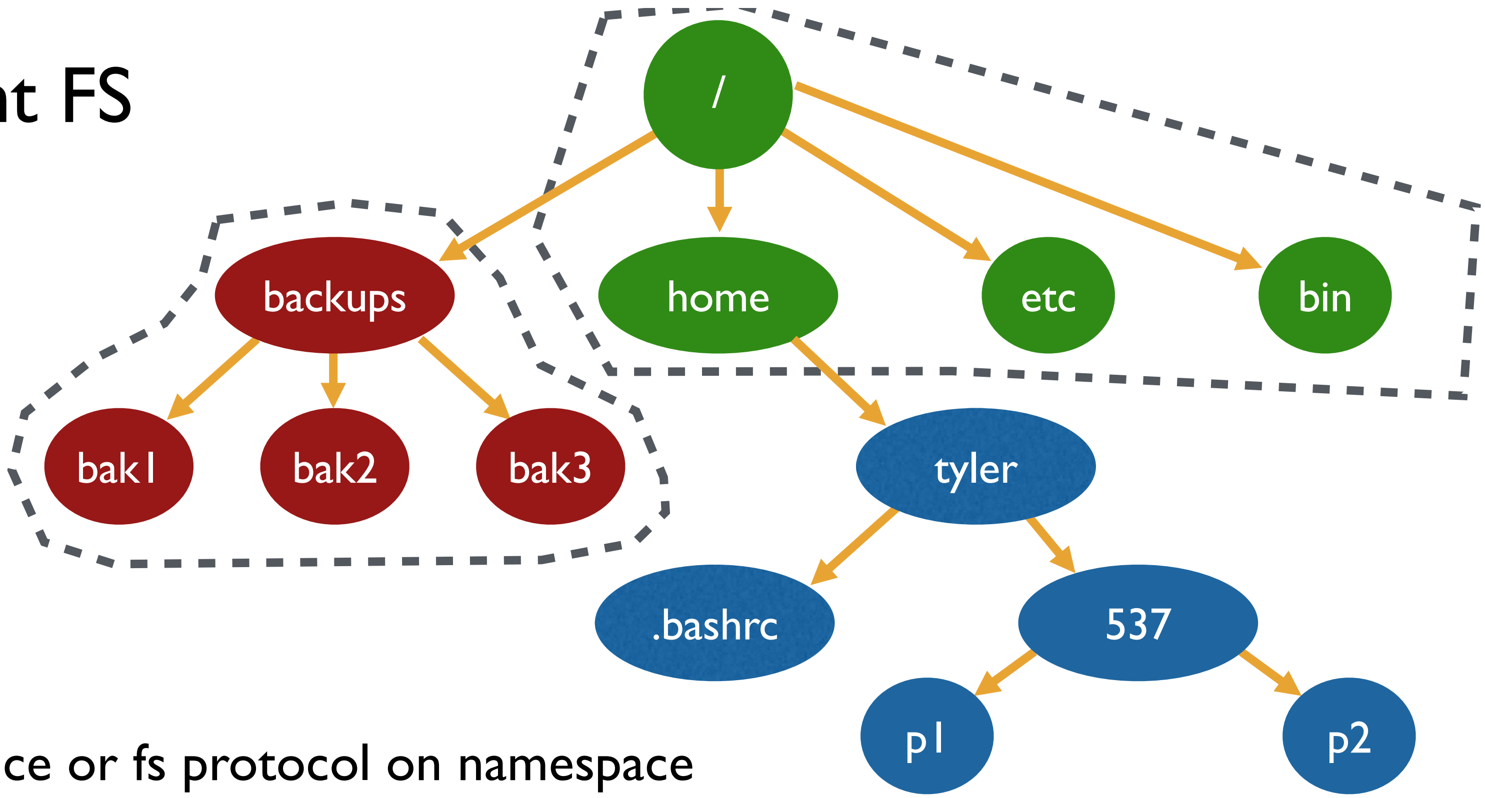
Caching

NFS ARCHITECTURE



RPC: Remote Procedure Call
Cache individual blocks of NFS files

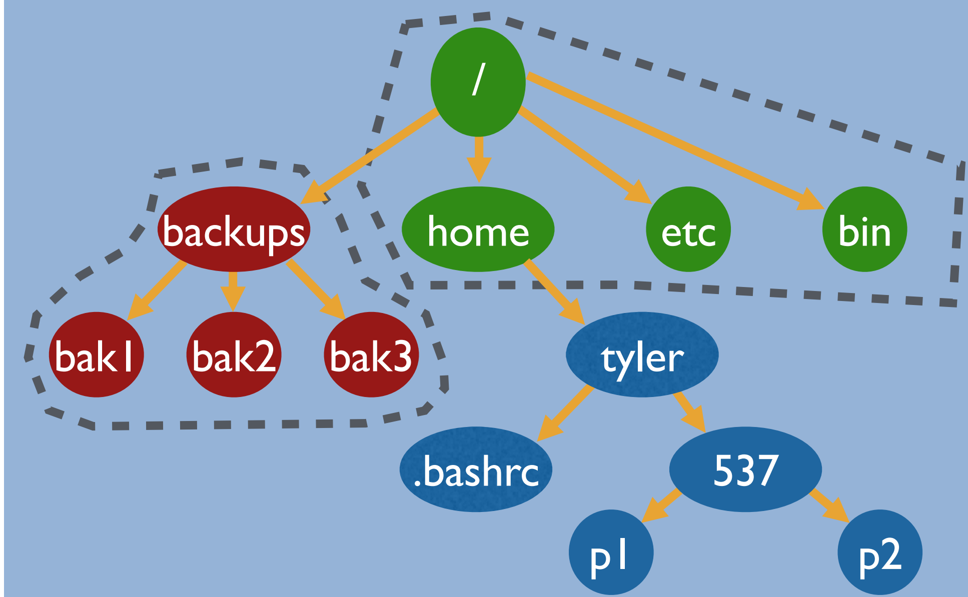
Client FS



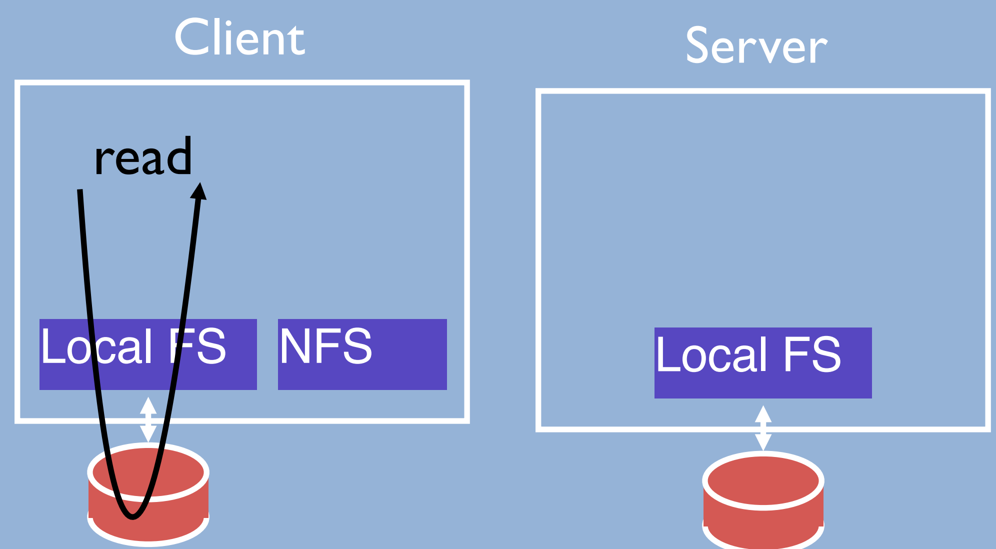
Mount: device or fs protocol on namespace

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

GENERAL STRATEGY: EXPORT FS

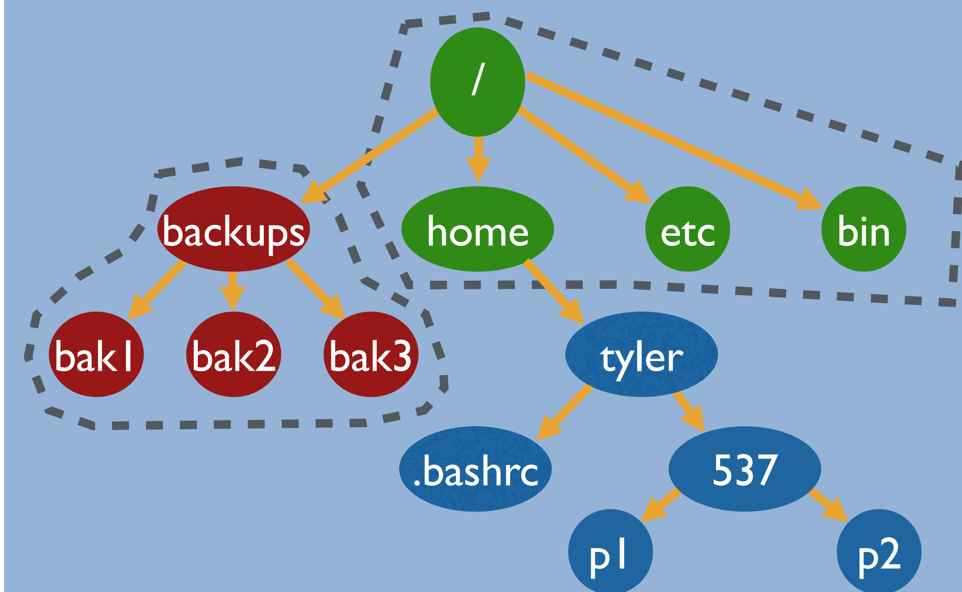


Where will read to /backups/bak1 go?

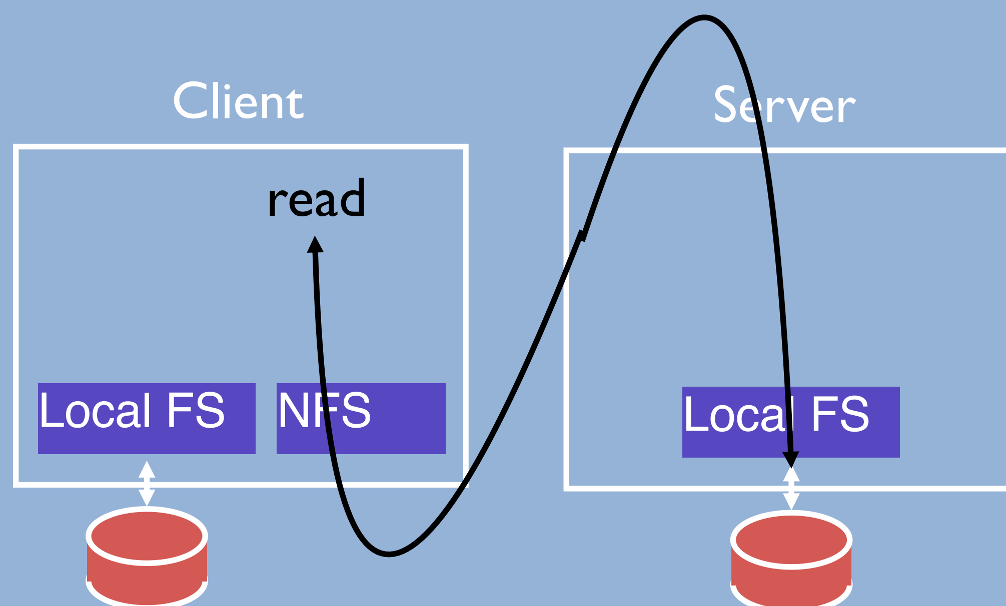


/dev/sda1 **on** /
/dev/sdb1 **on** /backups
NFS **on** /home/tyler

GENERAL STRATEGY: EXPORT FS



Where will read to /home/tyler/.bashrc go?



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

OVERVIEW

~~Architecture~~

Network API:
How do clients communicate with NFS server?

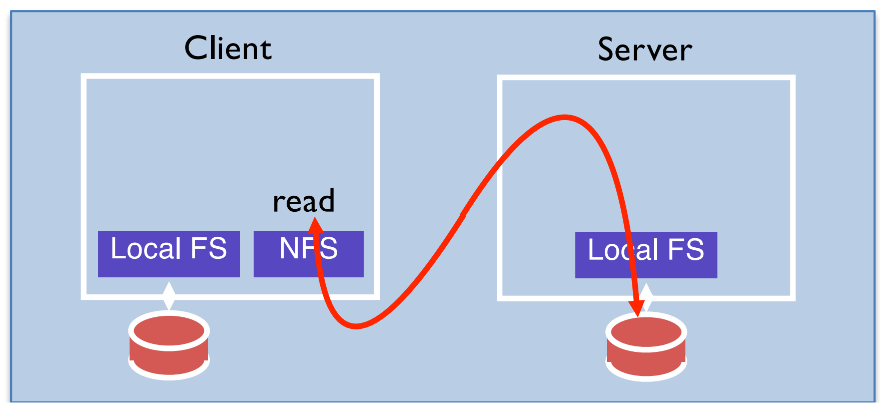
Caching

API STRATEGY 1

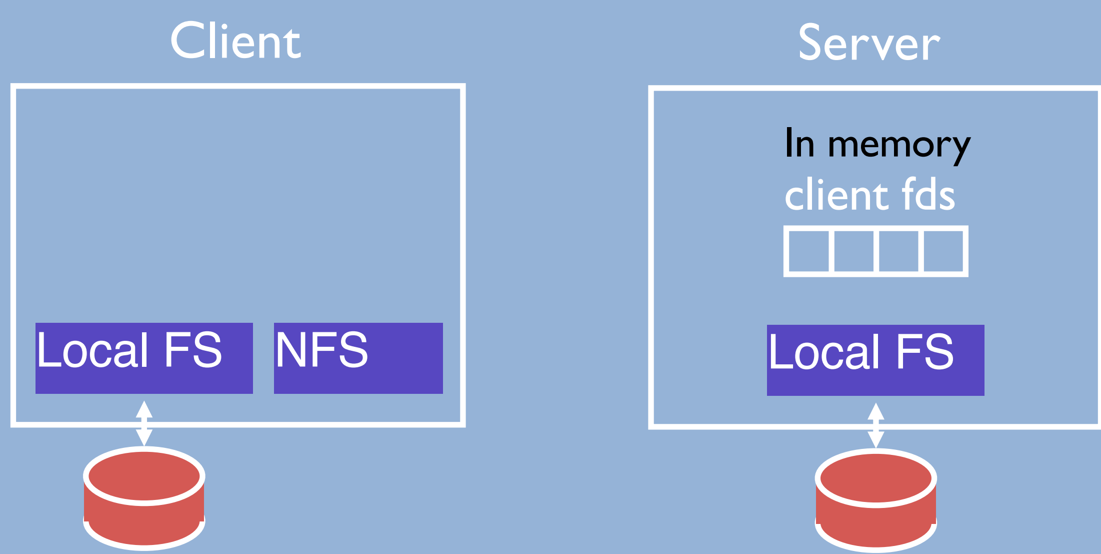
Attempt: Wrap regular UNIX system calls using RPC (Remote Procedure Call)

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

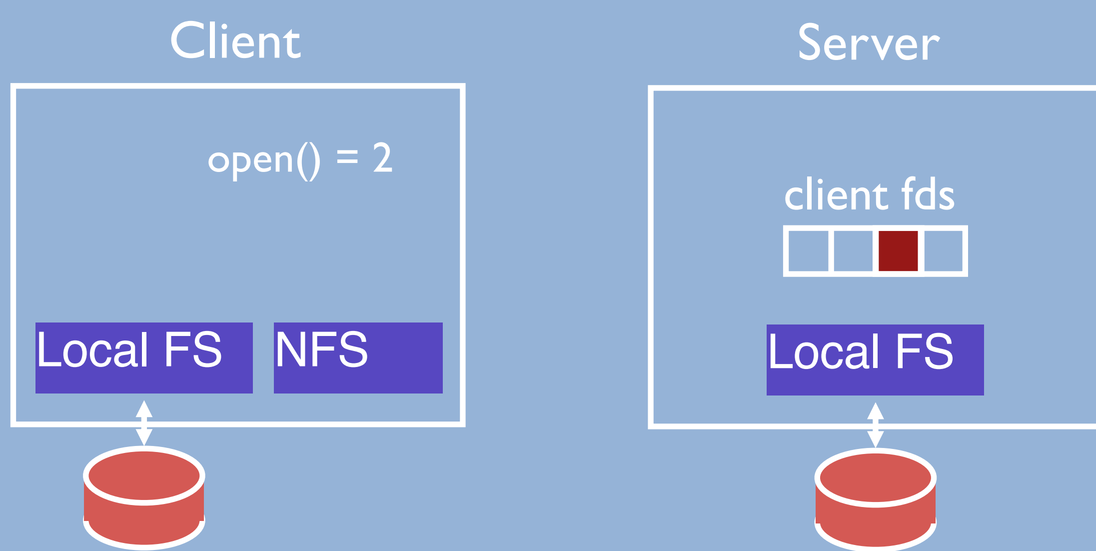
- read(fd) on client calls read(fd) on server
- read(fd) on server returns data back to client



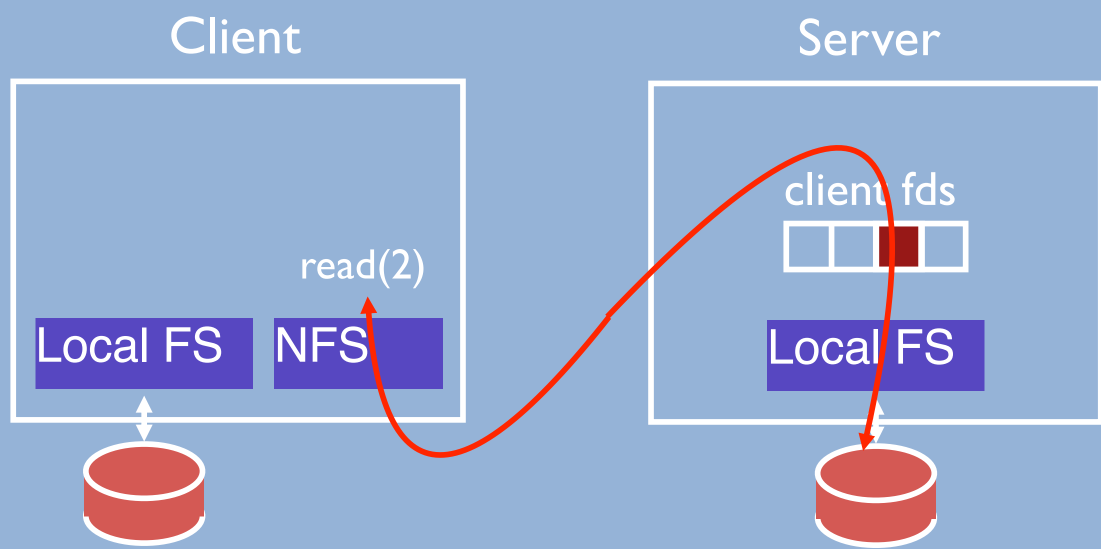
FILE DESCRIPTORS



FILE DESCRIPTORS



FILE DESCRIPTORS



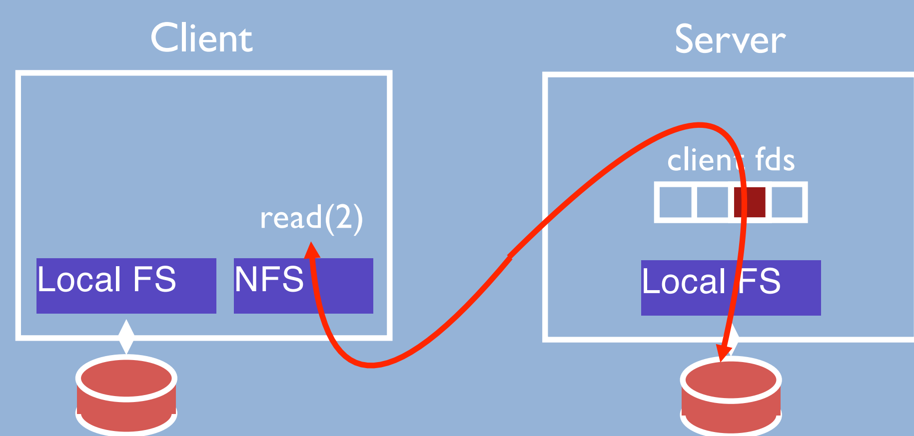
Remember: What is fd tracking?

STRATEGY 1 PROBLEMS

What about server crashes? (and reboots)

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);      <-- Server crash!
...
read(fd, buf, MAX);
```

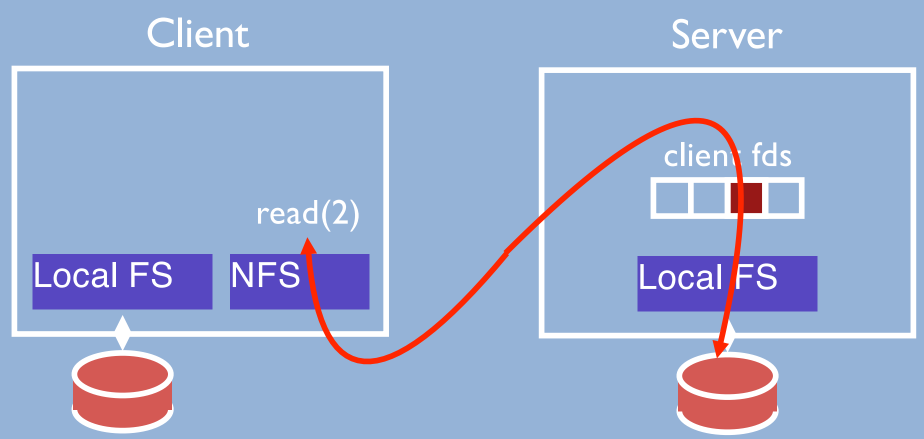
Goal: behave like slow read



POTENTIAL SOLUTIONS

1. Run some crash recovery protocol when server reboots
   – Complex

2. Persist fds on server disk
   – Slow for disks
   – How long to keep fds? What if client crashes? misbehaves?



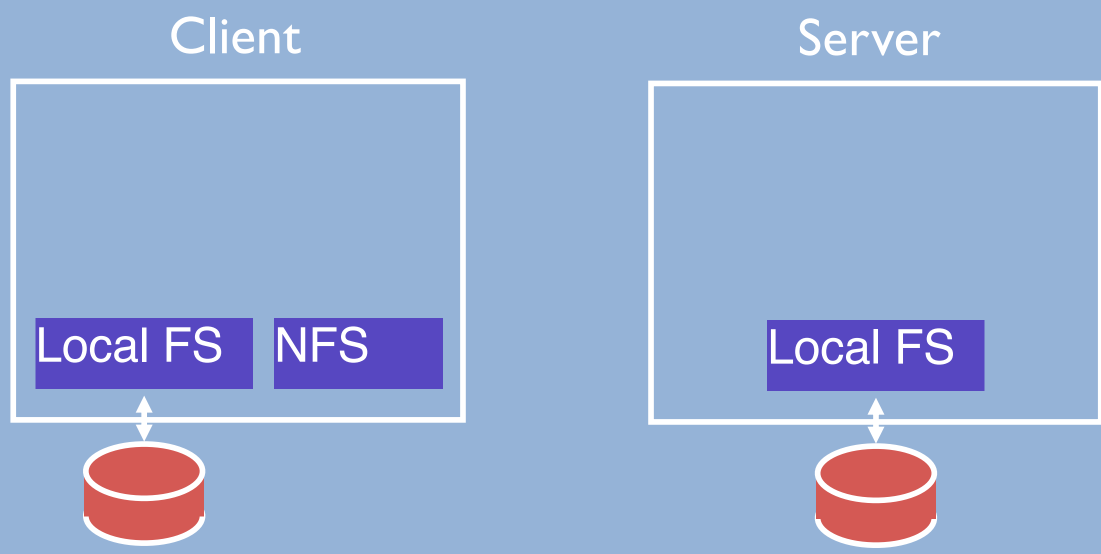
API STRATEGY 2: PUT ALL INFO IN REQUESTS

Every request from client completely describes desired operation

Use "stateless" protocol!

- server maintains no state about clients
- server can still keep other state just as hints (cached copies)
- can crash and reboot with no correctness problems (just performance)
- Main idea of NFSv2

ELIMINATE FILE DESCRIPTORS



STRATEGY 2: PUT ALL INFO IN REQUESTS

Use "stateless" protocol!

- server maintains no state about clients

Need API change. Get rid of fds; One possibility:

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

Specify path and offset in each message
Server need not remember anything from clients

Pros?

Server can crash and reboot transparently to clients

Cons?

API STRATEGY 3: INODE REQUESTS

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

With some new interfaces on server, this is pretty good! Any correctness problems?

API 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)

One of the fields in an inode is generation #,
incremented each time inode is allocated to new file/directory

CAN NFS PROTOCOL INCLUDE APPEND?

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

Problem with append()?

RPC often has "at-least-once" semantics (may call procedure on server multiple times)
(implementing "exactly once" requires state on server, which we are trying to avoid)

If RPC library replays messages, what happens when append() is retried on server?

Could wrongly append() multiple times if server crashes and reboots

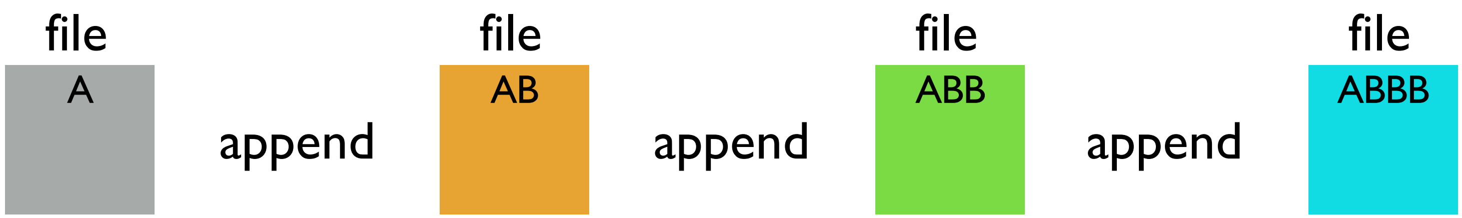
IDEMPOTENT OPERATIONS

Solution:
Design API so no harm if execute 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

API STRATEGY 4: FILE HANDLES

Do not include append() in protocol

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

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

Can applications call append????

FINAL API STRATEGY 5: CLIENT LOGIC

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

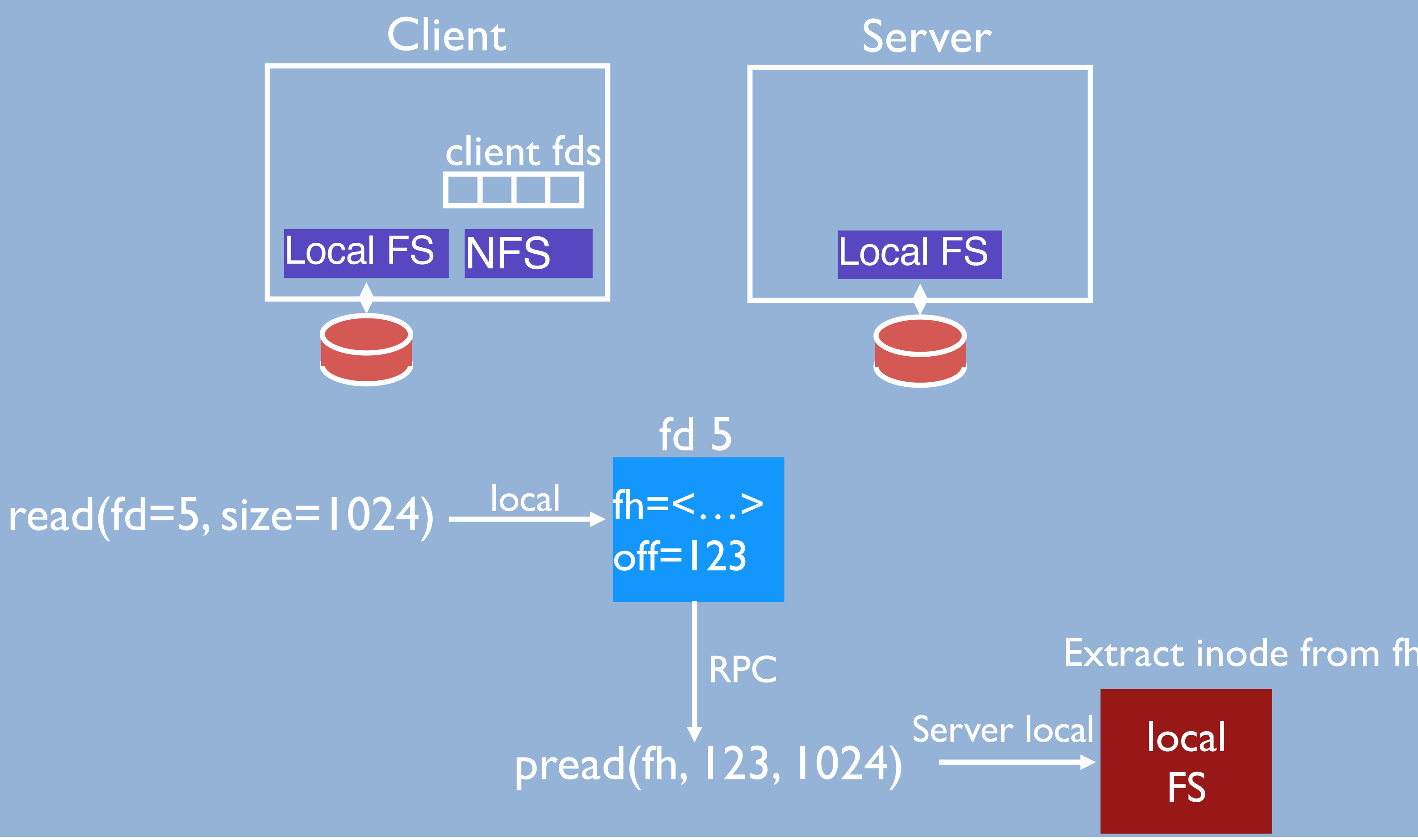
Clients maintain their own file descriptors

Client open() creates a local fd object

Local fd object contains:

- file handle (returned by server)
- current offset (maintained by client)

FINAL API STRATEGY 5: CLIENT LOGIC



NFS OVERVIEW

~~Architecture~~

~~Network API~~

Cache

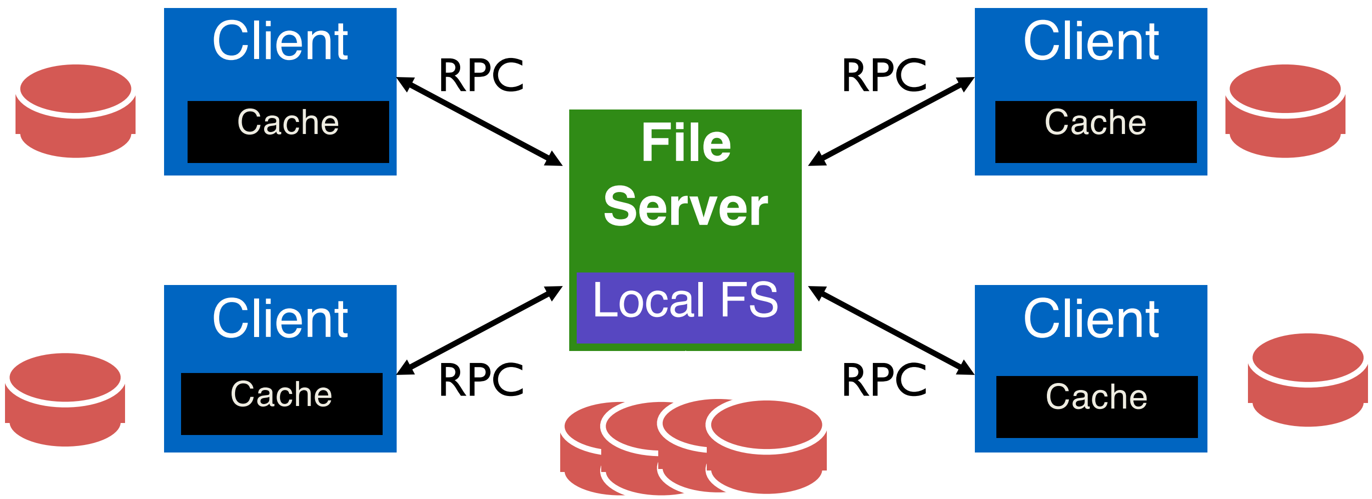
CACHE CONSISTENCY

NFS can cache data in three places:

- server memory
- client disk
- client memory

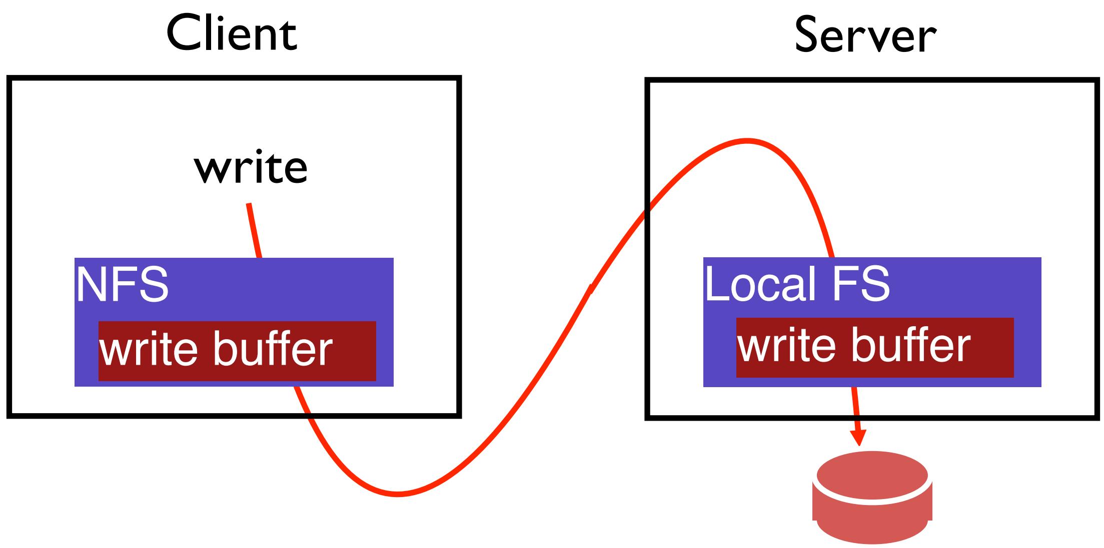
How to make sure all versions are in sync?

NFS ARCHITECTURE



RPC: Remote Procedure Call
Cache individual blocks of NFS files

CACHE PROBLEM 1: SERVER MEMORY



NSF Server often buffers writes to improve performance;
Server might acknowledge write before write is pushed to disk

What happens if server crashes?

SERVER MEMORY – LOST ON CRASH

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

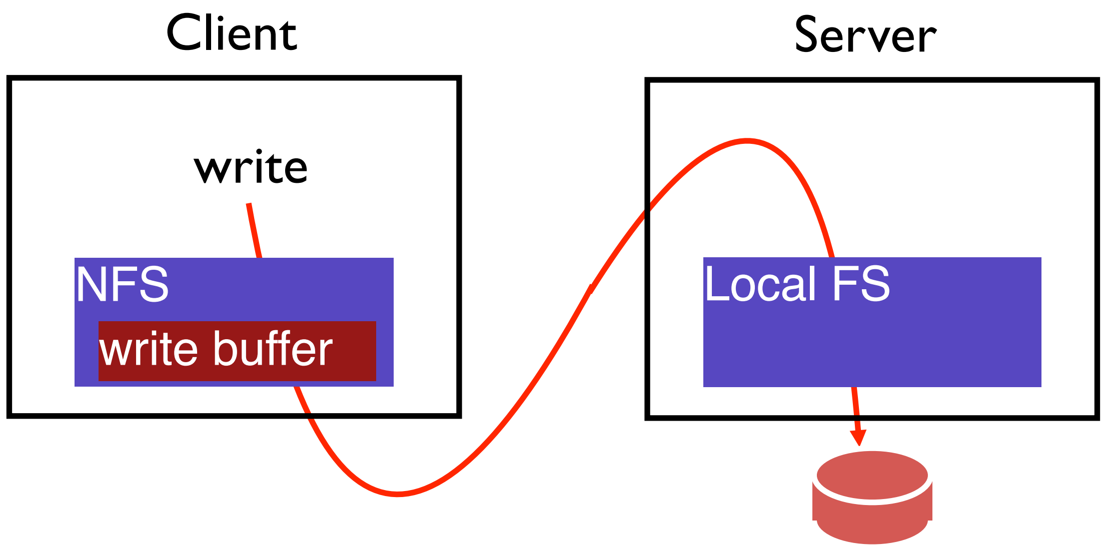
	0	1	2
server mem:			
server disk:	X	B	Z

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

What could have happened?

Solutions????

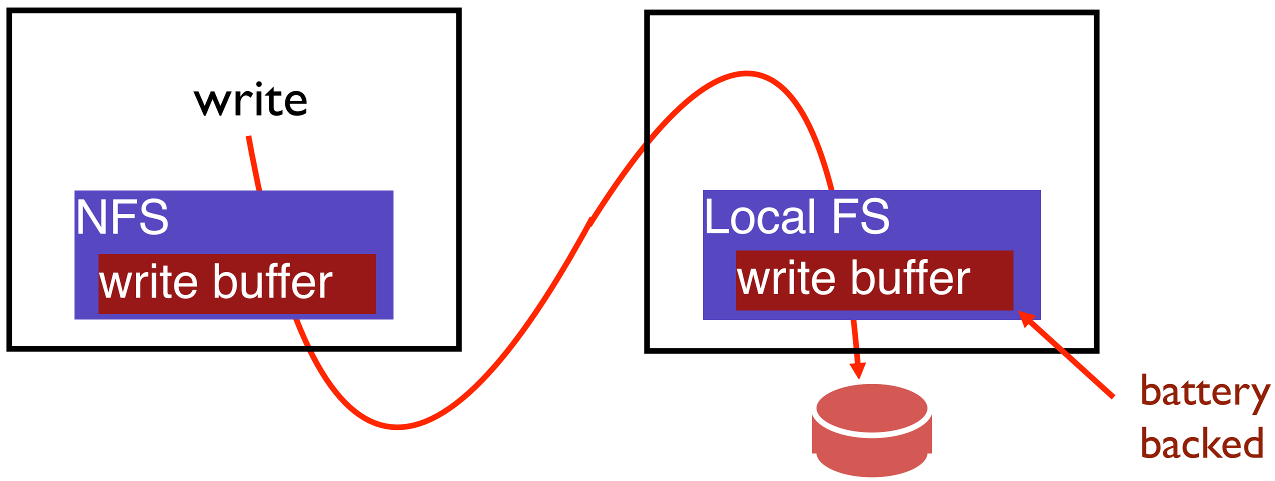
SERVER WRITE BUFFERS



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

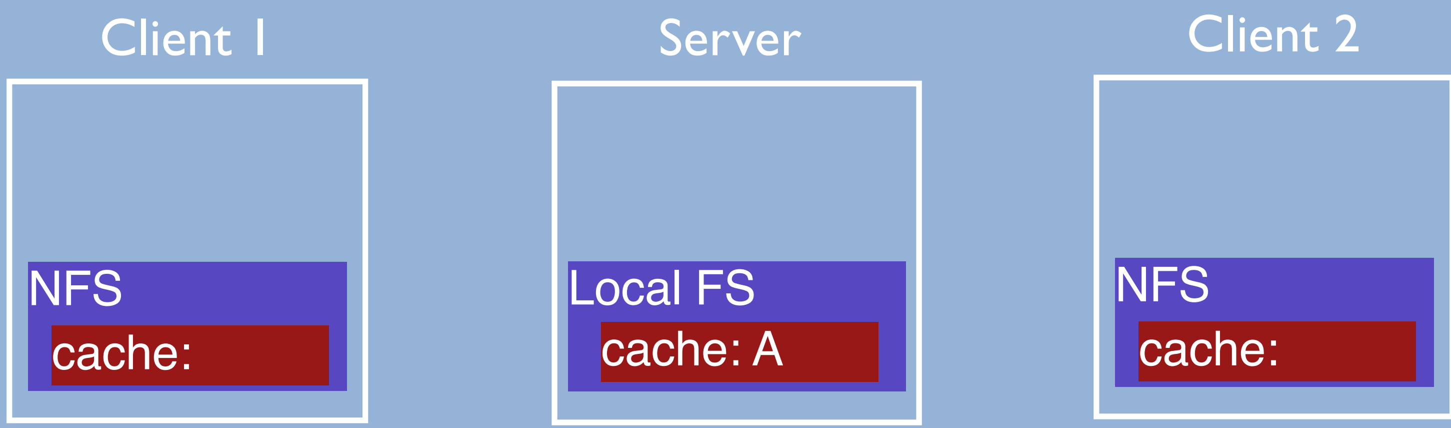
Problem: Slow!

SERVER WRITE BUFFERS



2. Use persistent write buffer (more expensive)

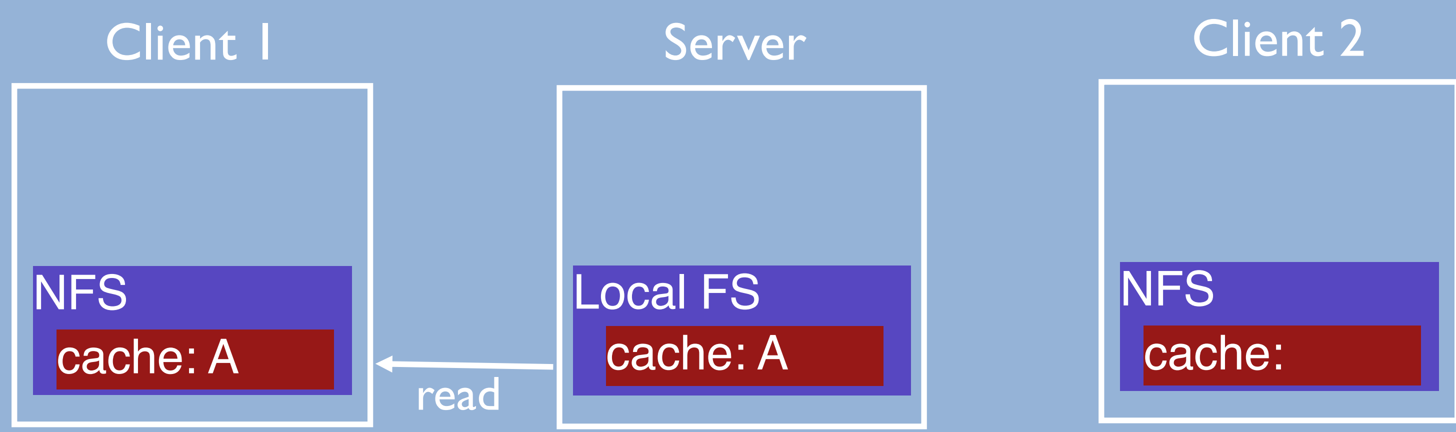
CACHE PROBLEM 2 + 3: DISTRIBUTED CACHE



Clients must cache some data

- Too slow to always contact server
- Server would become severe bottleneck

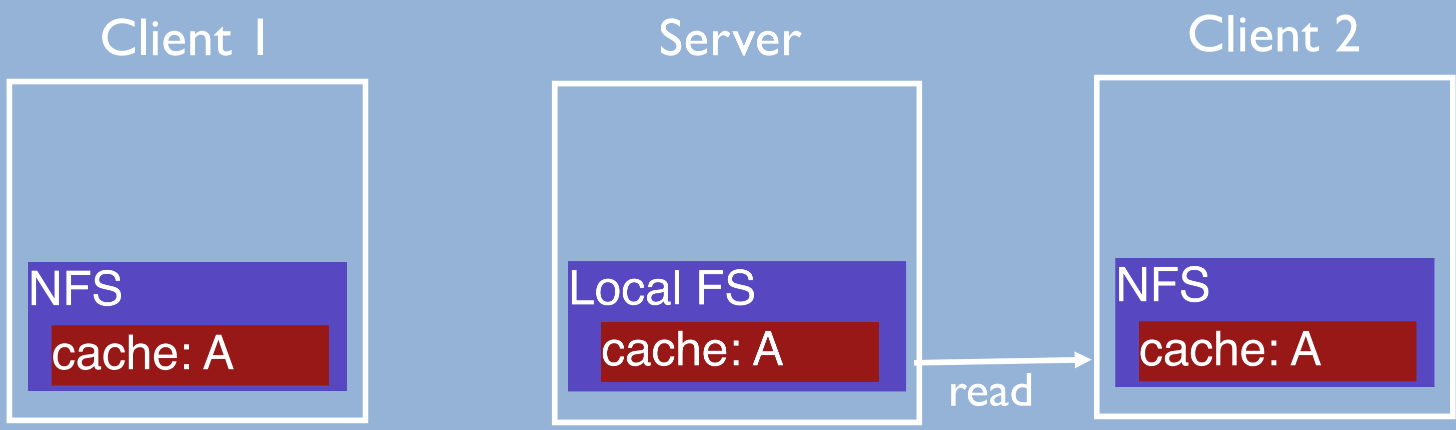
CACHE



Clients must cache some data

- Too slow to always contact server
- Server would become severe bottleneck

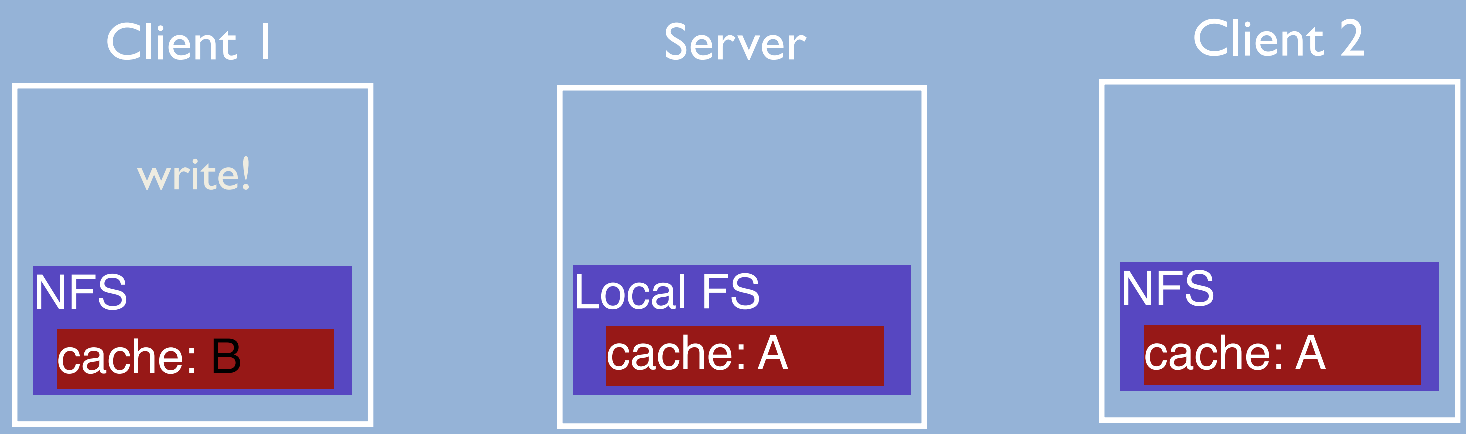
CACHE



Clients must cache some data

- Too slow to always contact server
- Server would become severe bottleneck

CACHE PROBLEM 2: UPDATE VISIBILITY

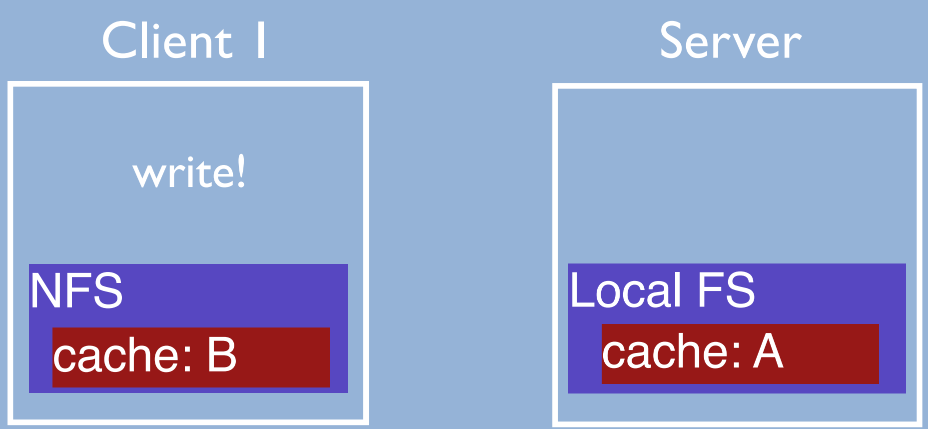


"Update Visibility" problem: server doesn't have latest version

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

Sees old version (different semantics than local FS)

SOLUTION TO 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 UPDATE VISIBILITY

Possibilities

- After every write (too slow)
- Periodically after some interval (odd semantics)

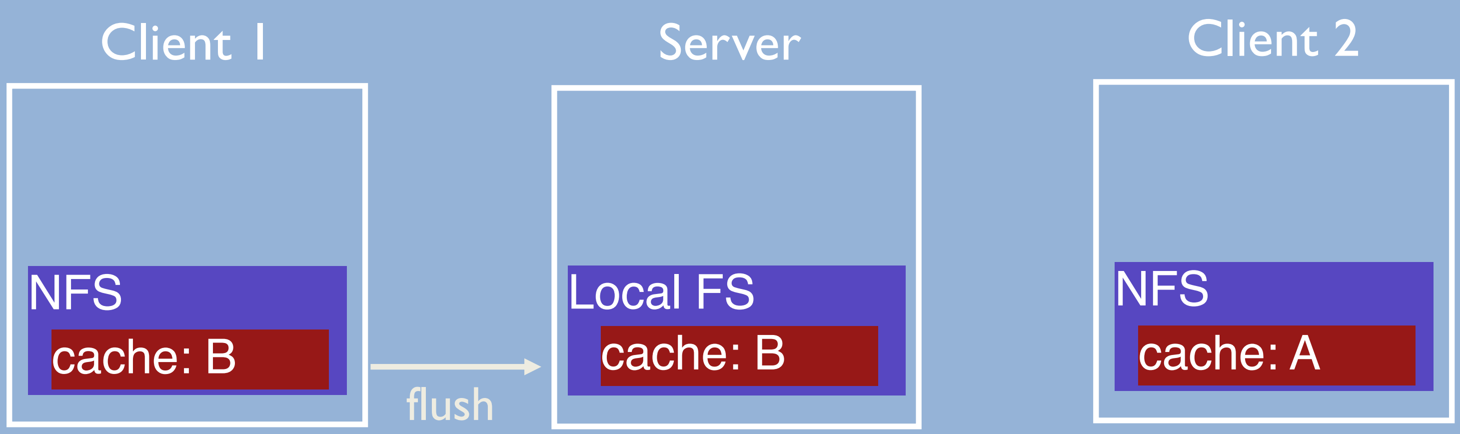
NFS solution: Flush blocks

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

Problems not solved by NFS:

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

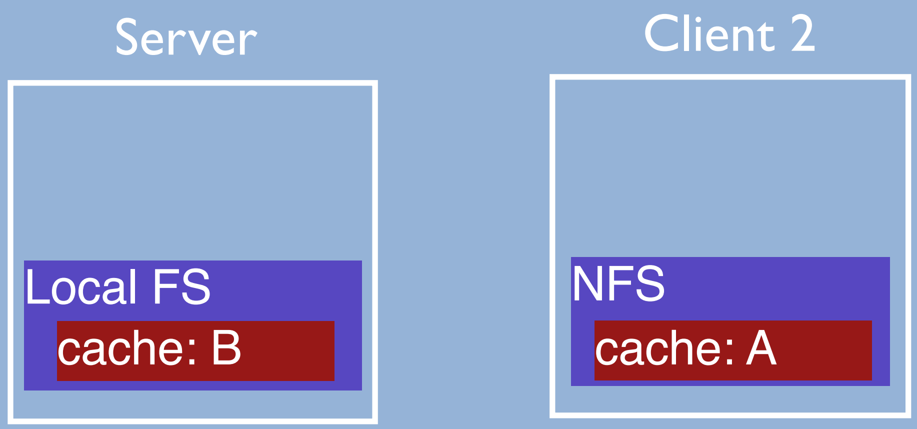
CACHE PROBLEM 3: STALE CACHE



"Stale Cache" problem: Client 2 doesn't have latest version from server

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

SOLUTION TO STALE CACHE



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

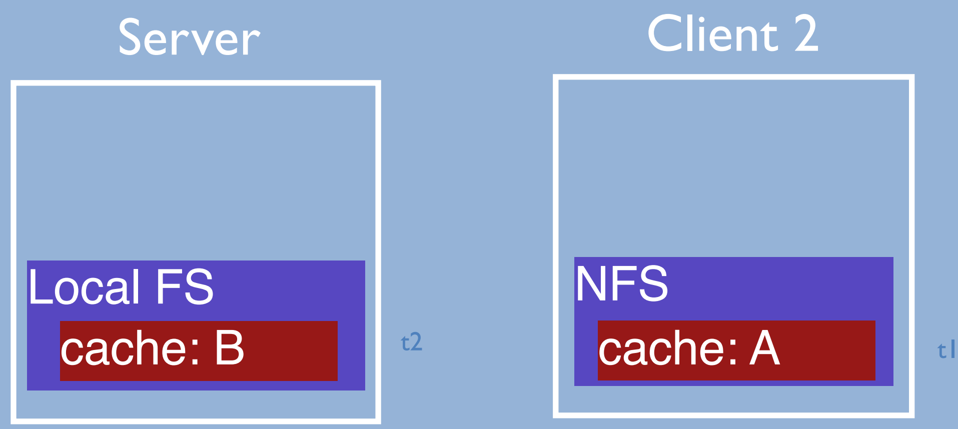
One possible solution:

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

NFS stateless solution:

- Clients recheck if cached copy is current before using data (recheck faster than getting data)

SOLUTION TO STALE CACHE

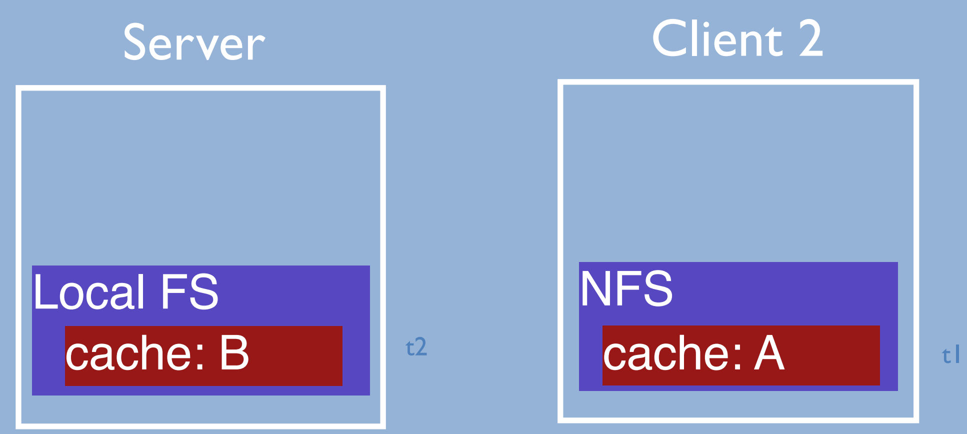


Client cache records time when **data block** was fetched (t1)

Before using data block, client sends file STAT request to server

- get's last modified timestamp for this **file** (t2) (not block…)
- compare to cache timestamp
- if file changed since block fetch timestamp (t2 > t1), then refetch data block

MEASURE THEN BUILD



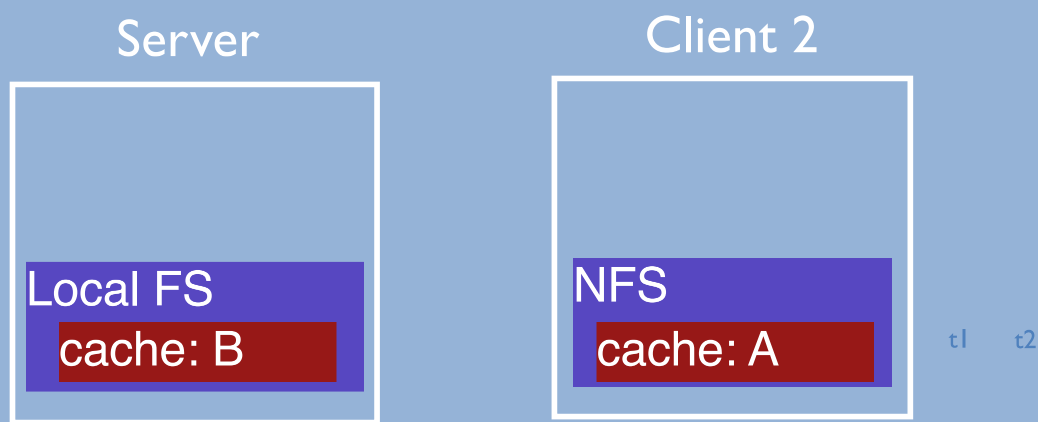
NFS developers found server overloaded – limits number of clients

Found `stat` accounted for 90% of server requests

Why?

Because clients frequently recheck cache

REDUCING STAT CALLS



Partial Solution: client caches result of `stat` (attribute cache)

What is result?

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

What is the result?

NFS SUMMARY

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

- **stateless**: servers don't remember clients or open files
- **idempotent**: repeating operations gives same results

Caching and write buffering is hard 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

Andrew File System: Carnegie Mellon University in 1980s

More reasonable semantics for concurrent file access

Improved scalability (many clients per server)

Willing to sacrifice simplicity and statelessness

AFS WHOLE-FILE CACHING

Approach
- Measurements show most files are read in entirety
- Upon open, AFS client fetches whole file, storing in local memory or disk
- Upon close, client flushes file to server (if file was written)

Convenient and intuitive semantics:
- Use same version of file entire time between open and close

Performance advantages:
- AFS needs to do work only for **open/close**
- **Reads/writes** are completely local

AFS CACHE CONSISTENCY

1. Update visibility:
   How are updates sent to the server?

2. Stale cache:
   How are other caches kept in sync with server?

AFS UPDATE VISIBILITY

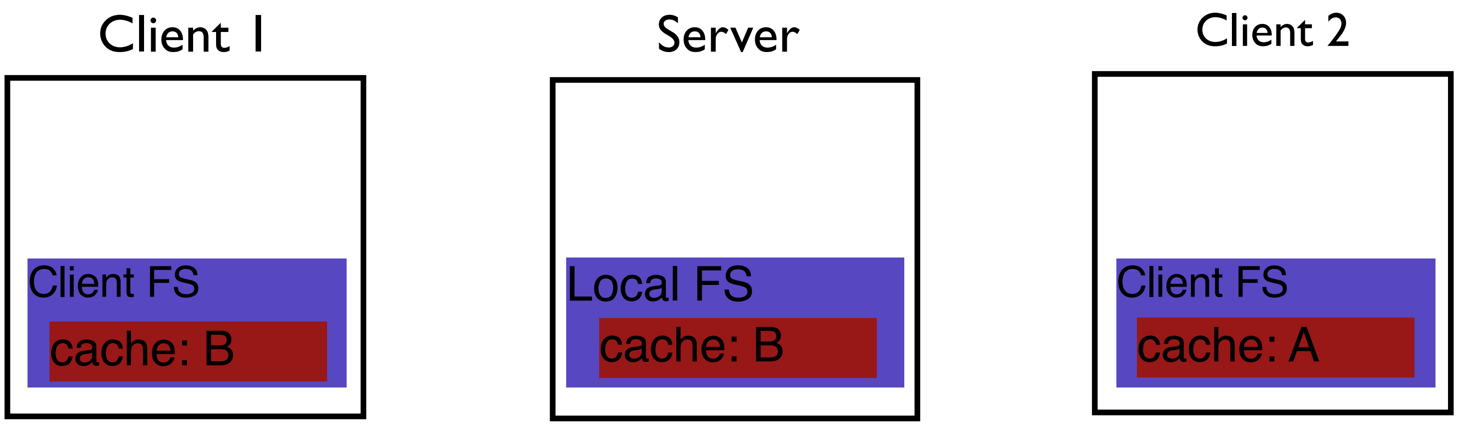
AFS solution:

- Like NFS, also flush on close
- Buffer **whole files** on local disk;
  update file on server atomically

Concurrent writes?

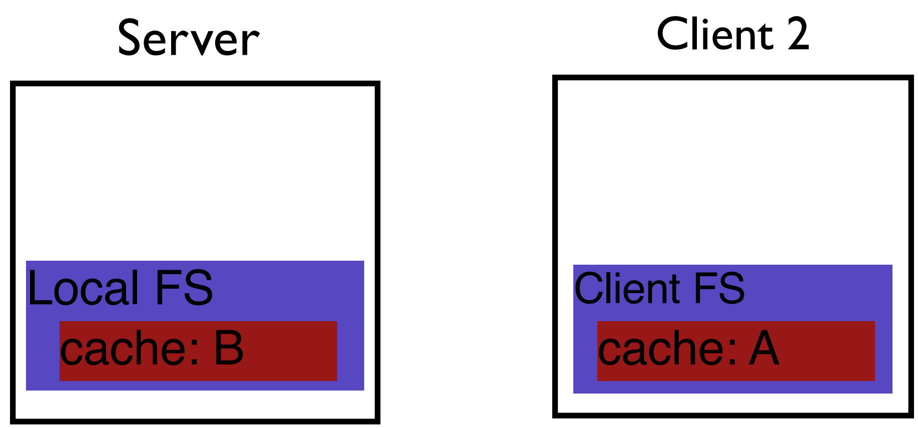
- **Last writer** (i.e., last file closer) wins
- Never get data mixed from multiple versions on server (unlike NFS)

AFS STALE CACHE PROBLEM



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

AFS: NO STALE CACHE



AFS solution: Server tells clients when data is overwritten

- Server must remember which clients have this file open right now
- Server is no longer stateless!

When clients cache data (on open), ask for "callback" from server if changes

- Clients can use data (during this open) without checking all the time

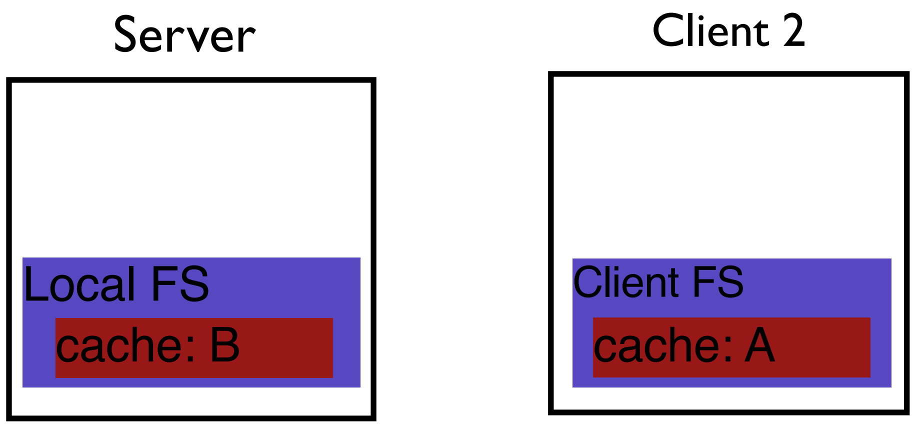
Clients only verifies callback when open() file (not every read); might not refetch on next open()

- Operate on same version of file from open to close

AFS CALLBACKS: DEALING WITH STATE

1. What if client crashes?
2. What if server runs out of memory?
3. What if server crashes?

DETAIL 1: CLIENT CRASH



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

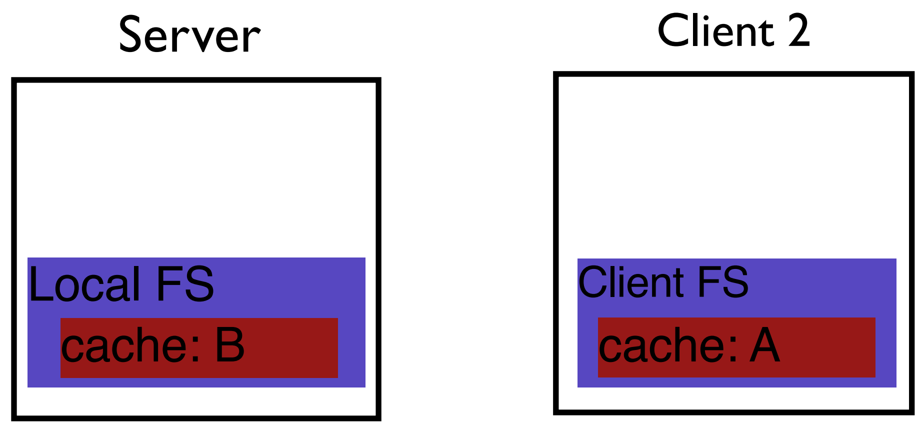
Concern? may have missed notification that cached copy changed

Option 1: evict everything from cache

Option 2: ???

recheck entries before using

DETAIL 2: 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

DETAIL (?) 3: 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…

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)

CACHE CONSISTENCY COMPARISON

- When will clients see changes?

- NFS
  - Individual reads: 3 seconds after other client closes file

- AFS
  - Whole file: Next time open file after other client closes file

NFS VS AFS 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?
What data will be sent? What will each client see?

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		getattr
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 - kickout		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		getattr
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 call back
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 call backs
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	