Welcome to the penultimate lecture!

DISTRIBUTED SYSTEMS, NFS

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

Project 1 - Project 6 regrades – Last call! → Wednesday → email the TA and cc me

Project 7 grades – this week, last regrade by Monday

Project 8 – final submissions by Thursday evening. →

Midterm 3: May 8th Monday

↳ Piazza → Videos, Old Exam

Venue

Quiz → Next day or two.

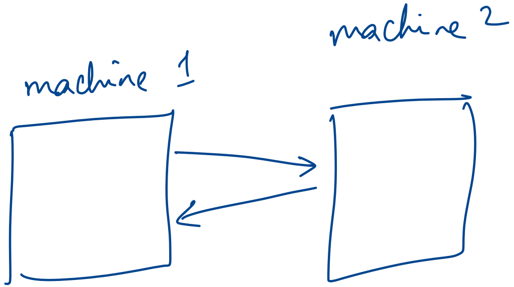
AGENDA / LEARNING OUTCOMES

What are some basic building blocks for systems that span across machines?

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

RECAP

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

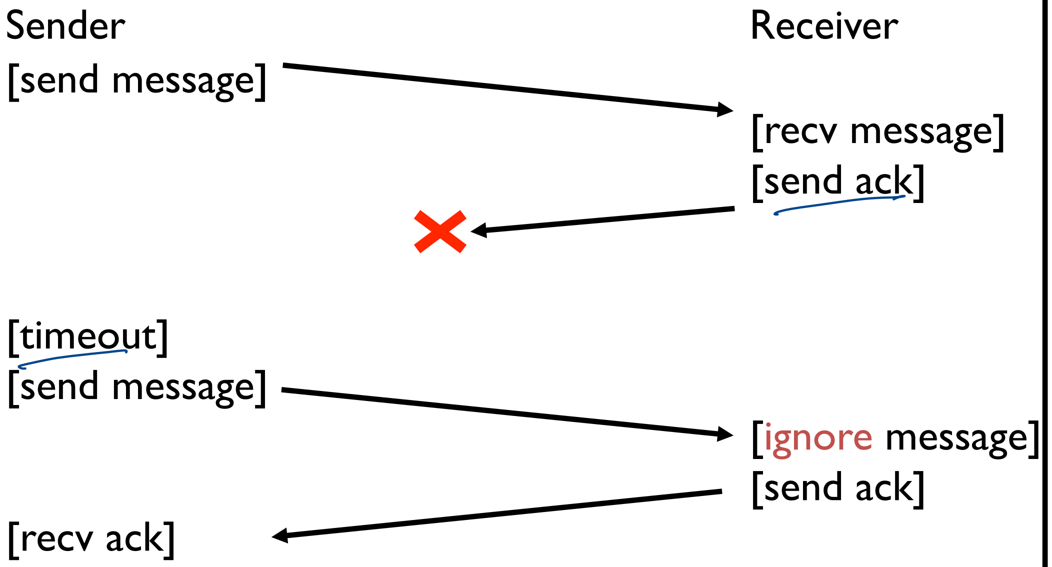
UDP:

Provide minimal reliability features: Best-effort delivery

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

TCP: ACKS, TIMEOUTS

Reliable delivery:



Sender

[send message]

Receiver

[recv message]

[send ack]

[timeout]

[send message]

[ignore message]

[send ack]

[recv ack]

Ordering , No duplicate messages

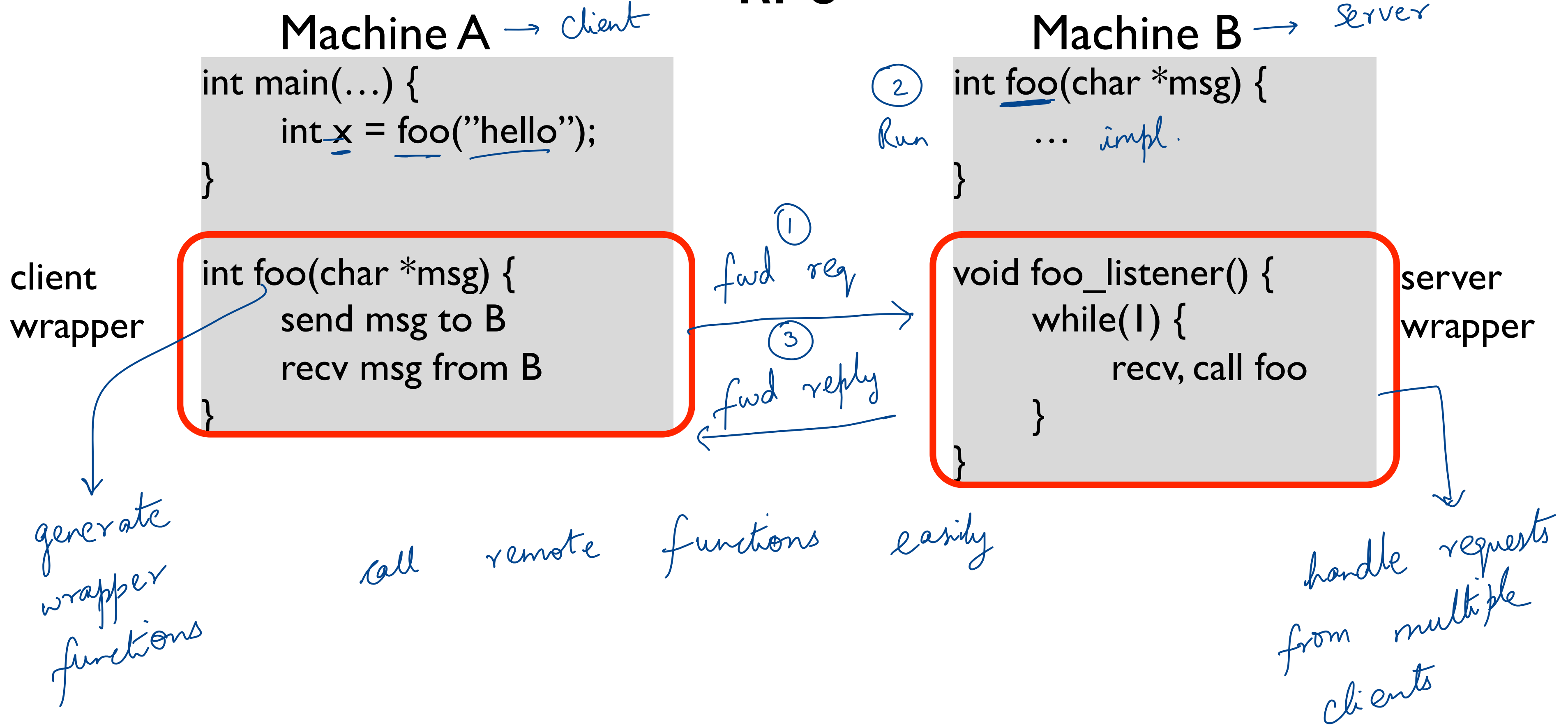
Sequence numbers

- senders gives each message an increasing unique seq number
- receiver knows it has seen all messages before N → last seq number

Suppose message K is received. delivered

- if K <= N, ignore it → duplicate
- if K = N + 1, first time seeing this message
- if K > N + 1, buffer and then deliver later

RPC → Remote procedure



Machine A → client

```
int main(...) {
    int x = foo("hello");
}

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

client wrapper

Machine B → server

```
int foo(char *msg) {
    ... impl.
}

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

server wrapper

generate wrapper functions

call remote functions easily

handle requests from multiple clients

WRAPPER GENERATION

Wrappers must do conversions:

- client arguments to message (stream of bytes)
- message to server arguments
- convert server return value to message (bytes)
- convert message to client return value

"hello"

↓

*0x 72 ....

↓

"hello"

client side

server side

Need uniform endianness (wrappers do this)

Conversion is called marshaling/unmarshaling, or serializing/deserializing

WRAPPER GENERATION: POINTERS

Why are pointers problematic?

Address passed from client not valid on server

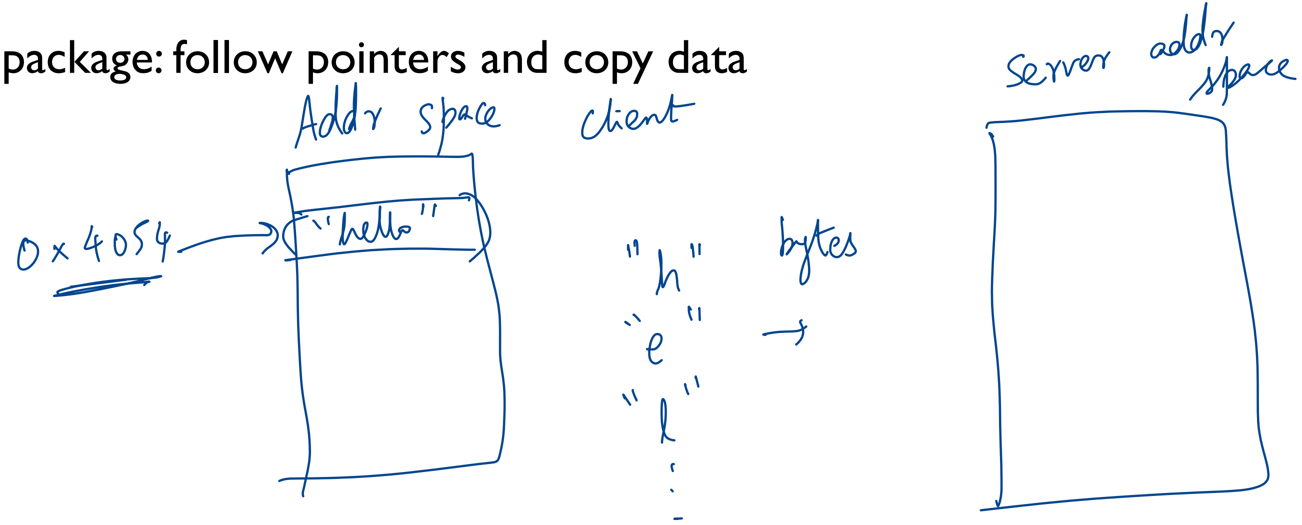
Solutions? Smart RPC package: follow pointers and copy data

wrapper function

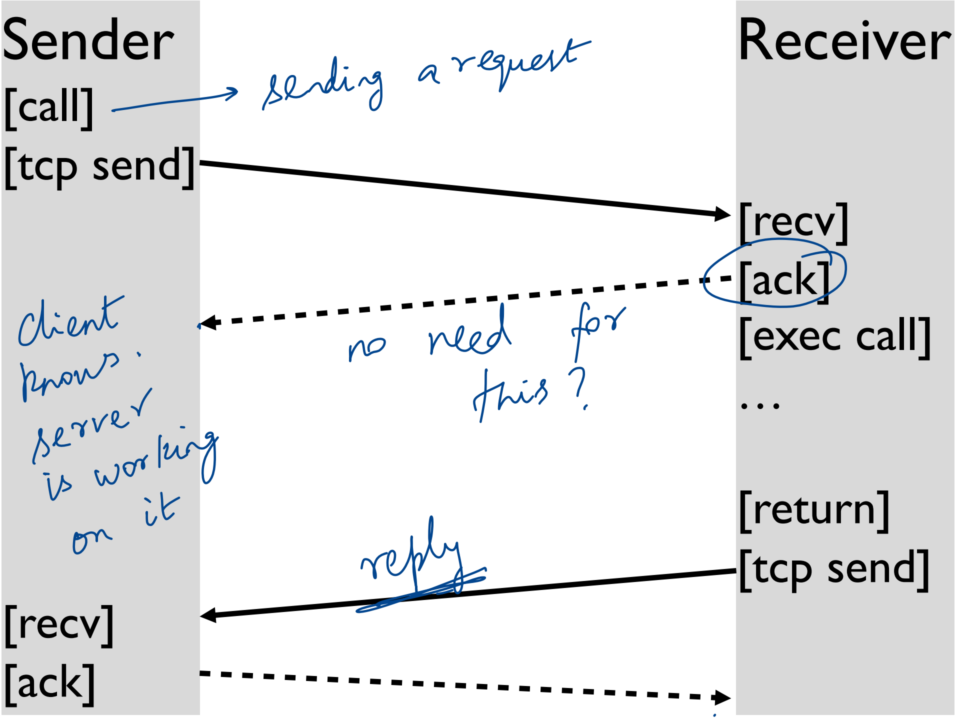
```
foo ( char * x ) {
    x → ptr to char
}
```

Argument is large:

→ Slow!



RPC OVER TCP?



Sender

[call] → sending a request

[tcp send]

client knows server is working on it

[recv]

[ack]

Receiver

[recv]

[ack]

no need for this ?

[exec call]

…

[return]

[tcp send]

reply

do work based on function

4 messages sent on network

↳ 1 RPC

1. Short RPC

→ Ack TCP is not very useful

2. Long RPC → Ack could be useful

RPC OVER UDP

Strategy: use function return as implicit ACK

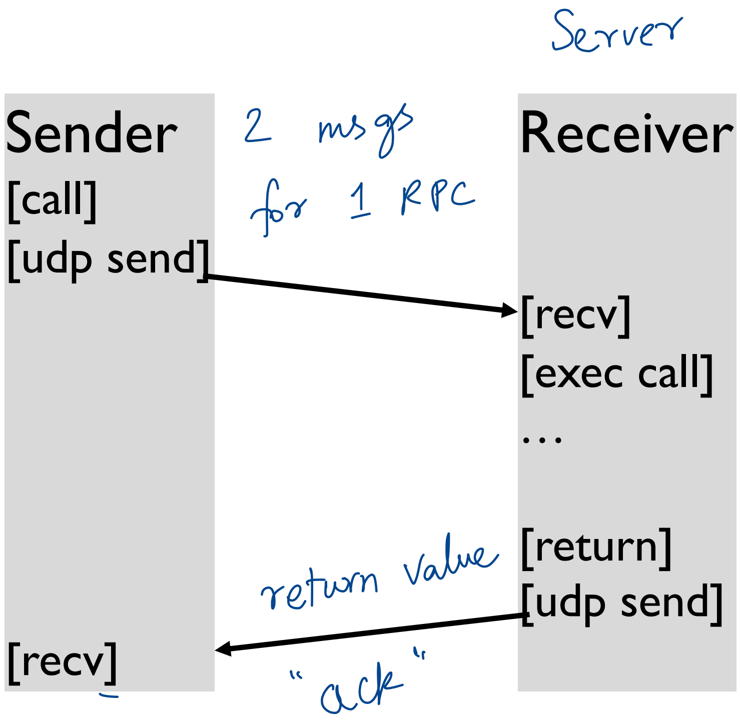
"Piggybacking" technique

What if function takes a long time?

then send a separate ACK

Protobuf → C++, Java . . . .

Thrift



Server

Sender

[call]

[udp send]

[recv]

Receiver

[recv]

[exec call]

…

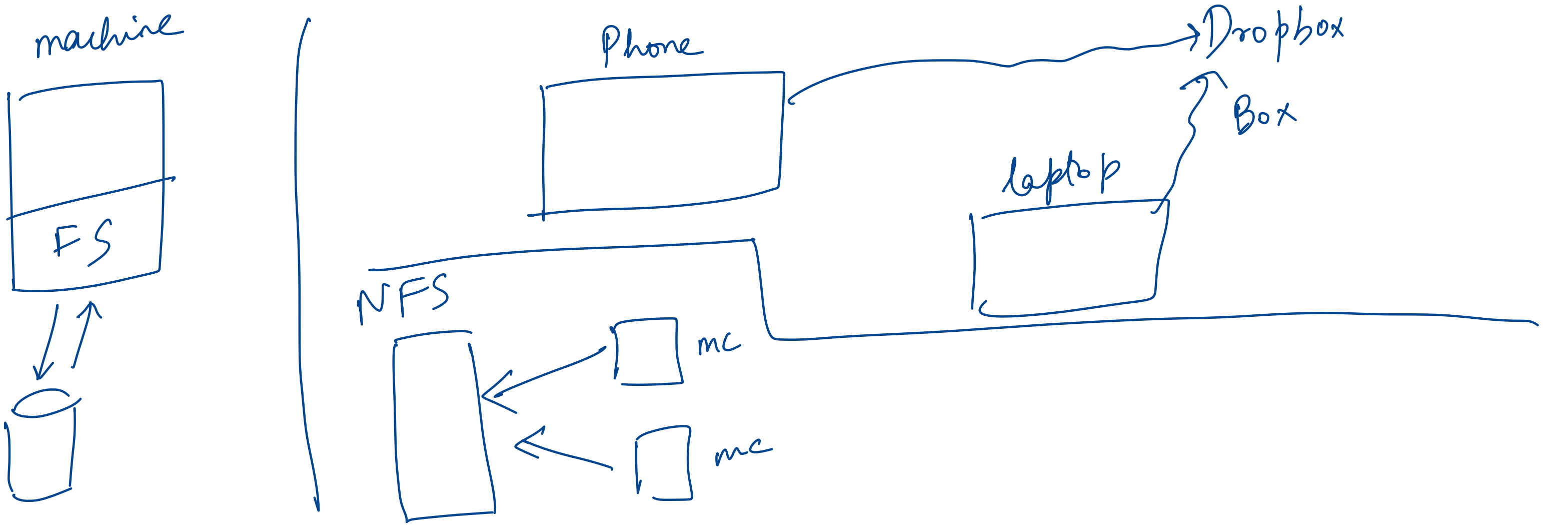
[return]

[udp send]

DISTRIBUTED FILE SYSTEMS

Local FS: processes on same machine access shared files

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

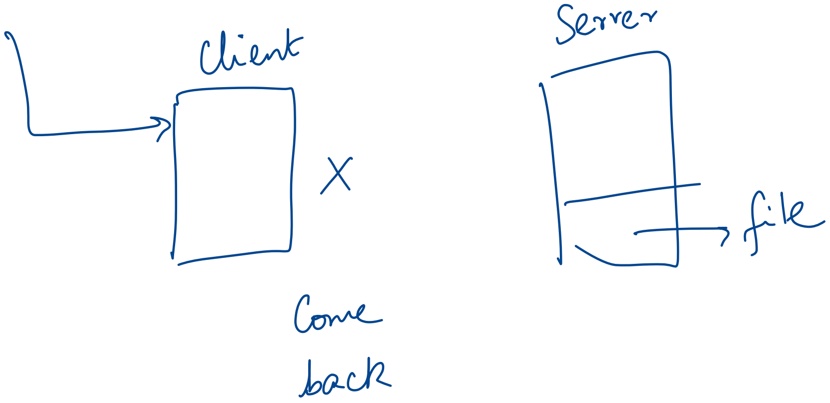


GOALS FOR DISTRIBUTED FILE SYSTEMS

- Transparent access → Don't need to modify user applications
  - can't tell accesses are over the network
  - normal UNIX semantics → open read writes . . .

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

Reasonable performance?



NETWORK FILE SYSTEM: NFS

NFS: more of a protocol than a particular file system

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

build storage servers

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

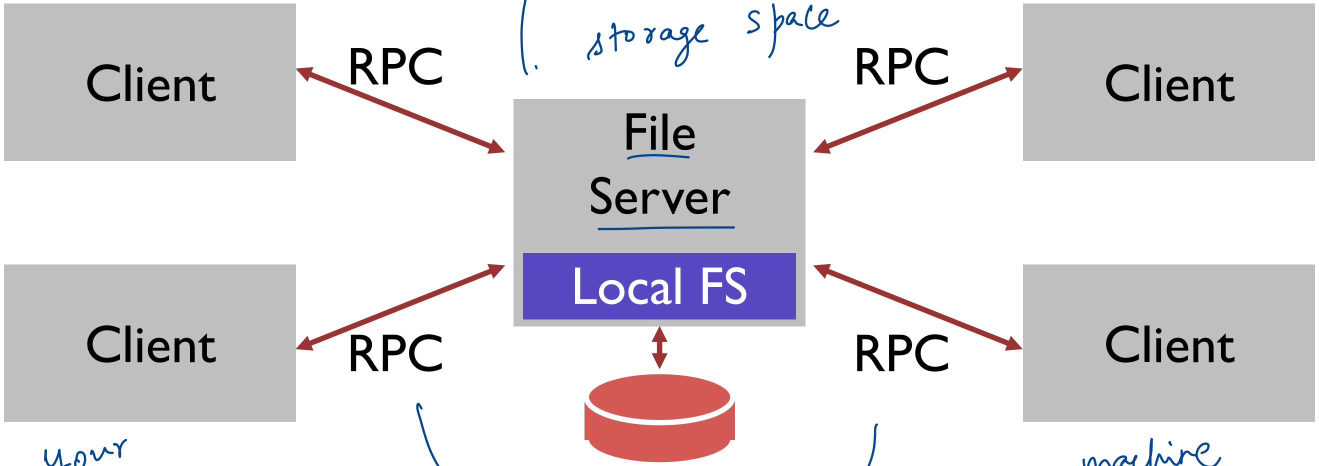
Specific concerns

NFS ARCHITECTURE

Thin client

lots of cores

lots of disks
storage space



Client

RPC

Client

RPC

File Server

Local FS

Client

RPC

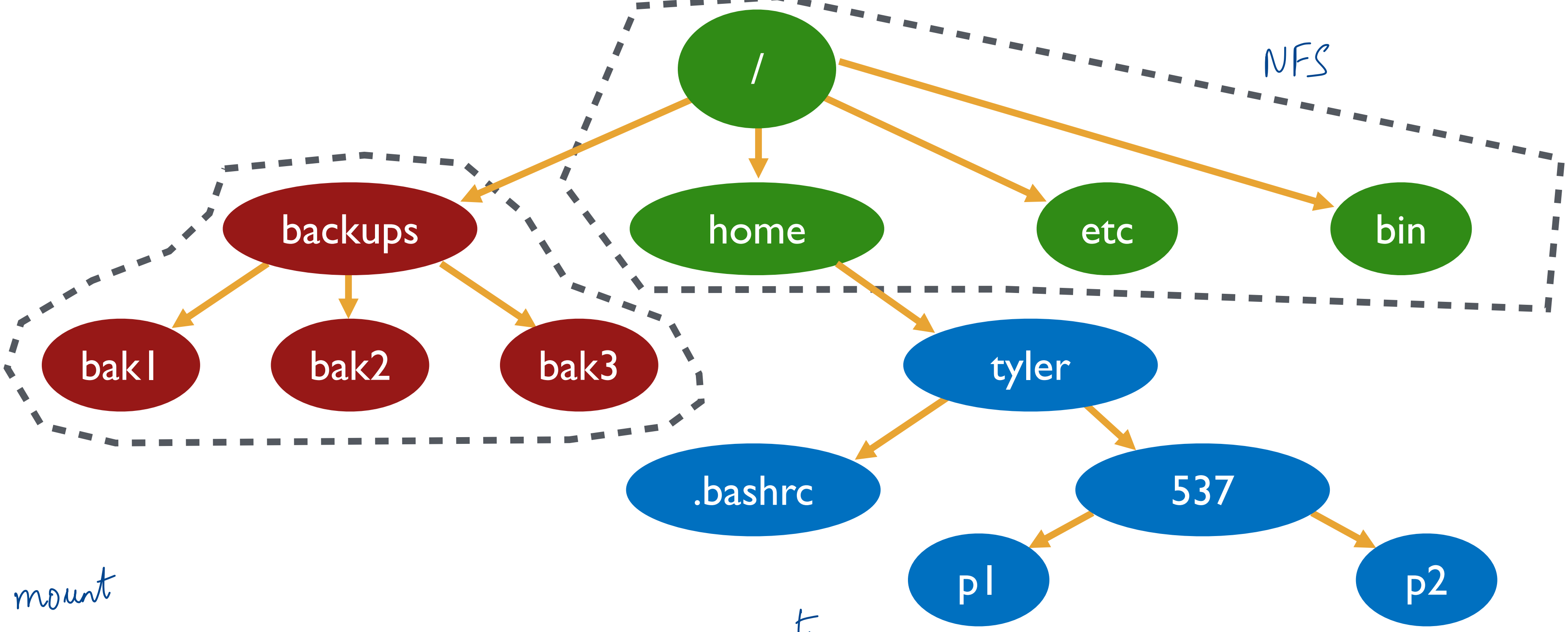
Client

RPC

your laptop

NFS protocol

machine



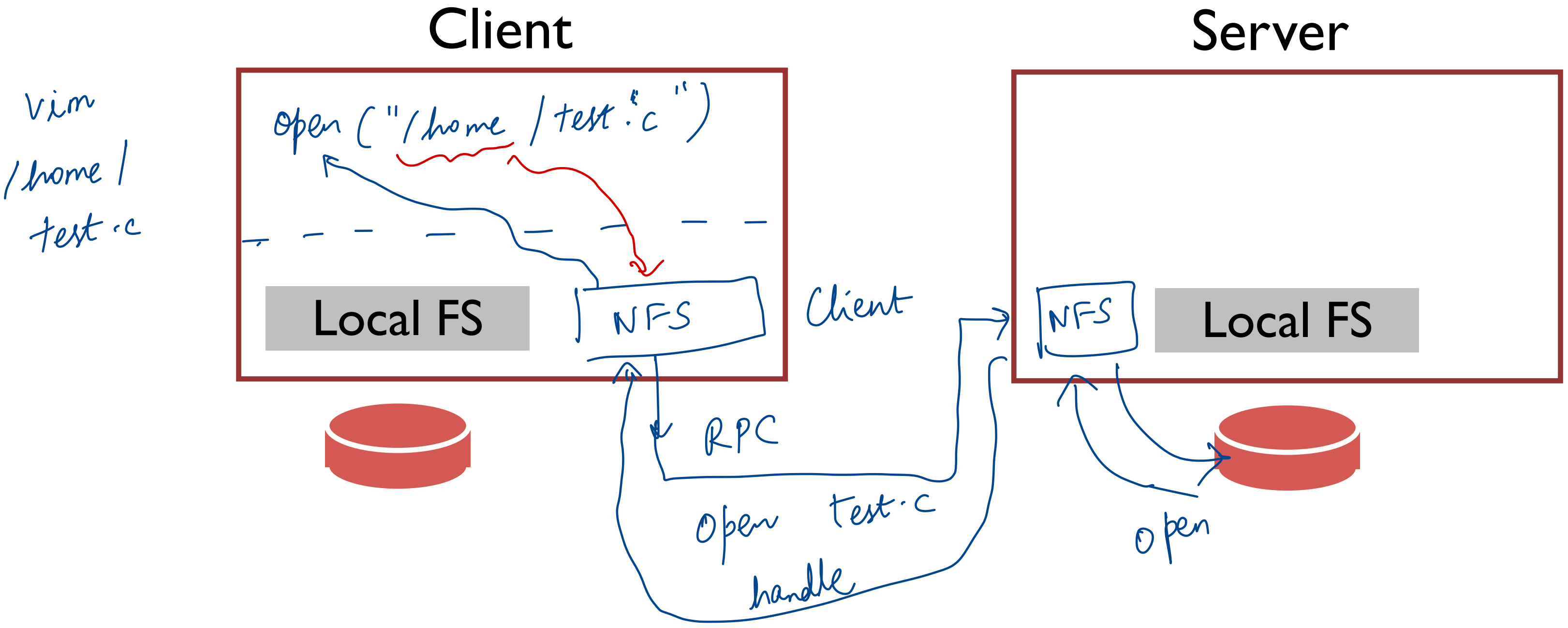
mount

→/dev/sda1 **on** /

/dev/sdb1 **on** /backups

NFS **on** /home

diff parts of directory tree are in diff FS



Client

Server

Local FS

Local FS

OVERVIEW

~~Architecture~~

Network API

Write Buffering

Cache

STRATEGY 1

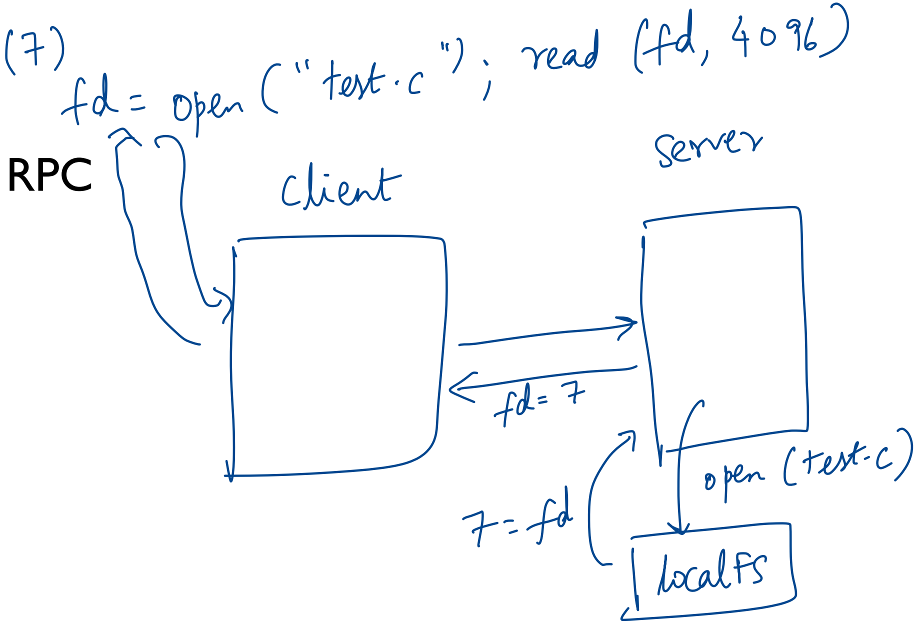
Attempt: Wrap regular UNIX system calls using RPC

open() on client calls open() on server

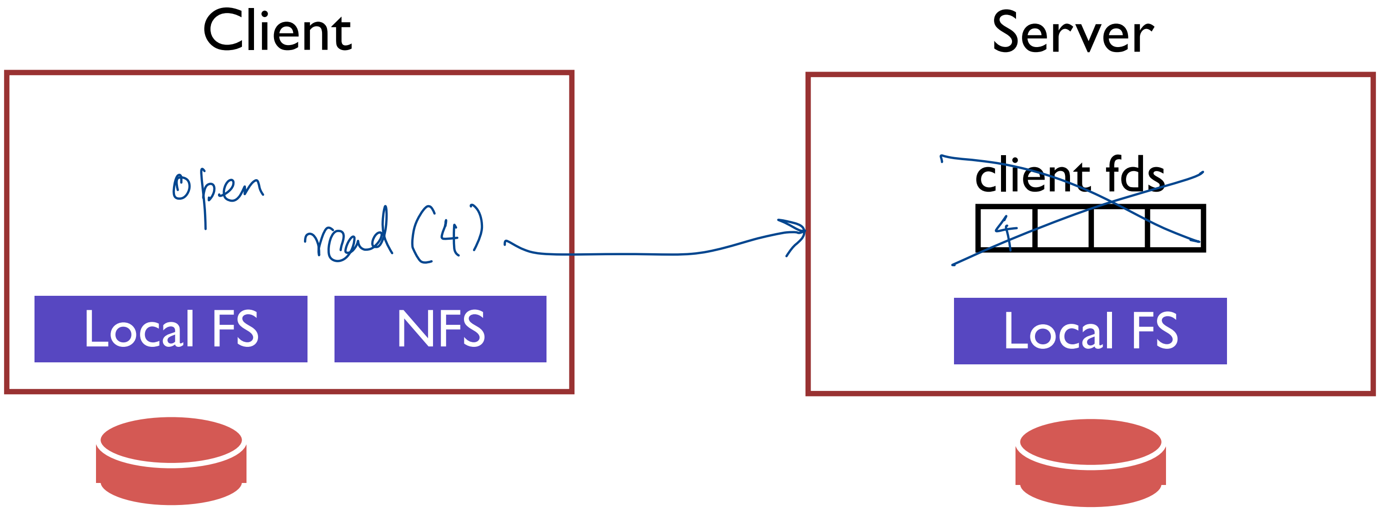
open() on server returns fd back to client

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

read(fd) on server returns data back to client



FILE DESCRIPTORS



Client

open

read (4)

Local FS

NFS

Server

~~client fds~~

~~4~~			

Local FS

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);
```

4096

Server crash! and comes back after 30s

Fail

retry read, succeed.

POTENTIAL SOLUTIONS

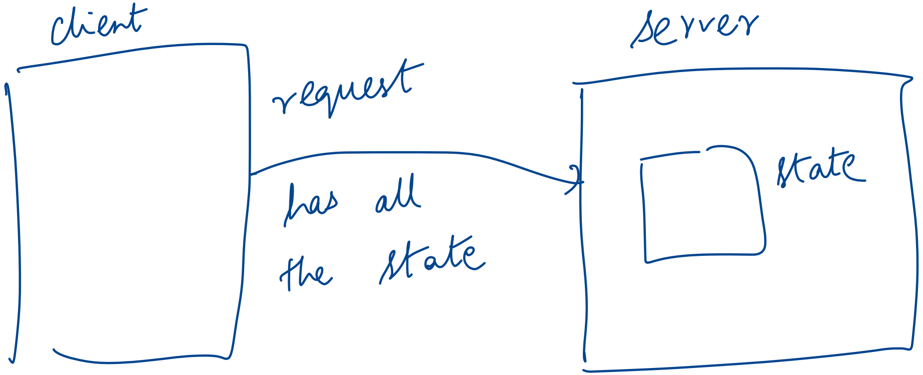
1. Run some crash recovery protocol upon reboot
   - Complex → large number of client

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



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); → server can execute this without looking up any state

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

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

Pros? → Server crash / recover is clean

Cons? → Perform traversal each time

STRATEGY 3: FILE HANDLES

part of the API

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

specify offset every time

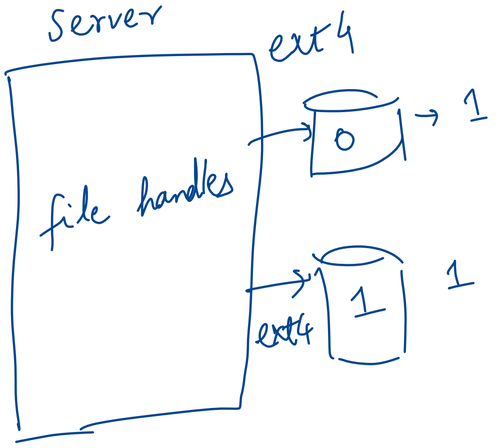
File Handle = <volume ID, inode #, **generation #**>

Opaque to client (client should not interpret internals)

which disk

file inside local FS

re-using inodes



/a/b/c/d/e/foo

↳ FH is better than path

Client **Server**

fd = open("/foo", ...);

Send LOOKUP (rootdir FH, "foo") → RPC

Receive LOOKUP request

look for "foo" in root dir → traversal

return foo's FH + attributes

"0x 5023"

reply

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

vim ← file descriptor ← 7

NFS client

FH	FD
0x5023	7

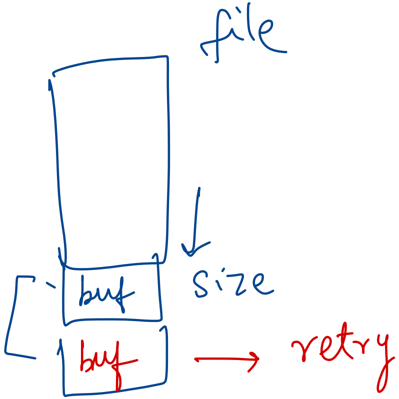
CAN NFS PROTOCOL INCLUDE APPEND?

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

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

API is diff

① Multiple clients → append()
→ append + read

② Failure

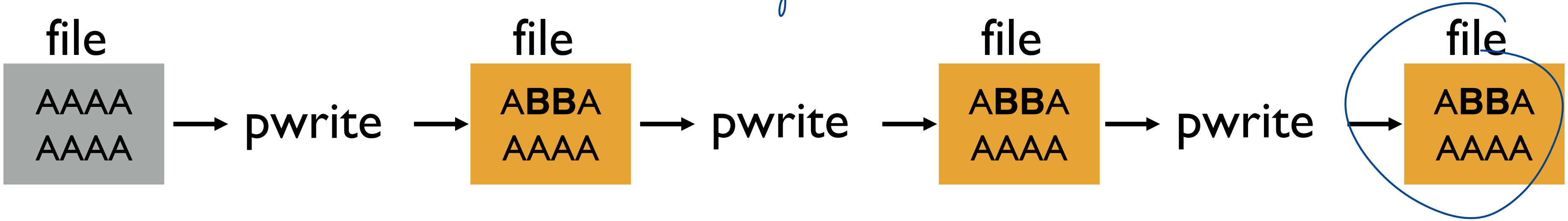


PWRITE VS APPEND

always same output

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

retry pwrite operation



file

AAAA
AAAA

→ pwrite →

file

ABBA
AAAA

→ pwrite →

file

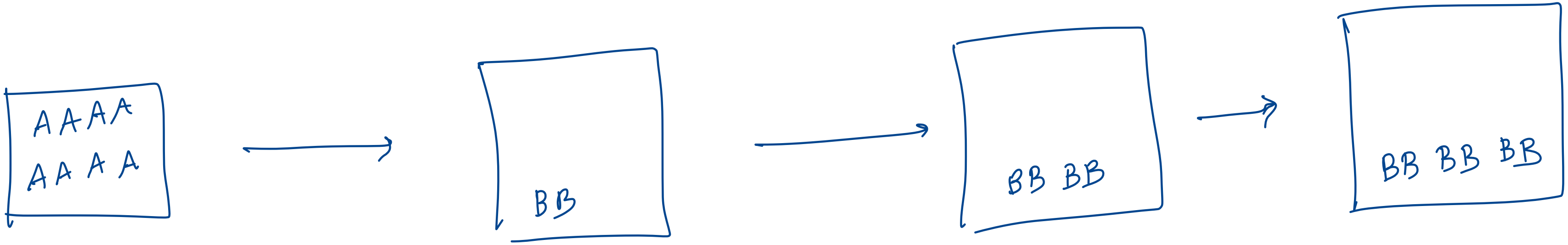
ABBA
AAAA

→ pwrite →

file

ABBA
AAAA

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() as many times

append is not idempotent

pread / pwrite are idempotent

```
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

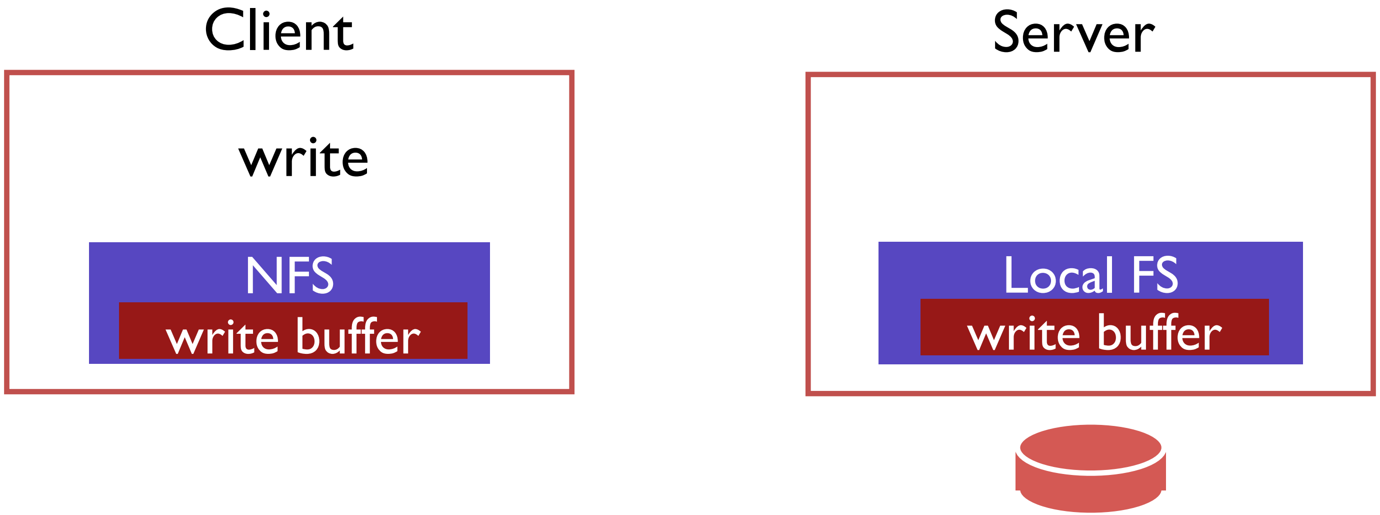
- append →

What about these?

- mkdir → mkdir ("/foo"); mkdir ("/foo")
- creat

↳ fail. /foo already exists

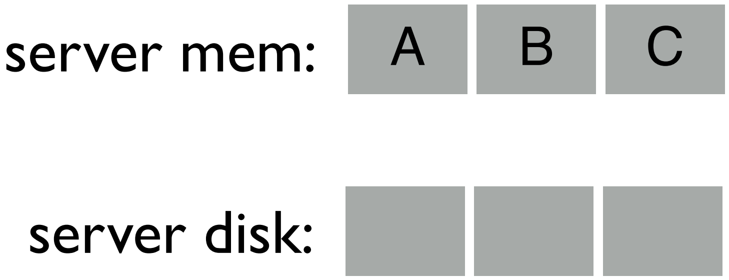
WRITE BUFFERS



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 C

server disk:

server acknowledges write before write is pushed to disk

SERVER WRITE BUFFER LOST

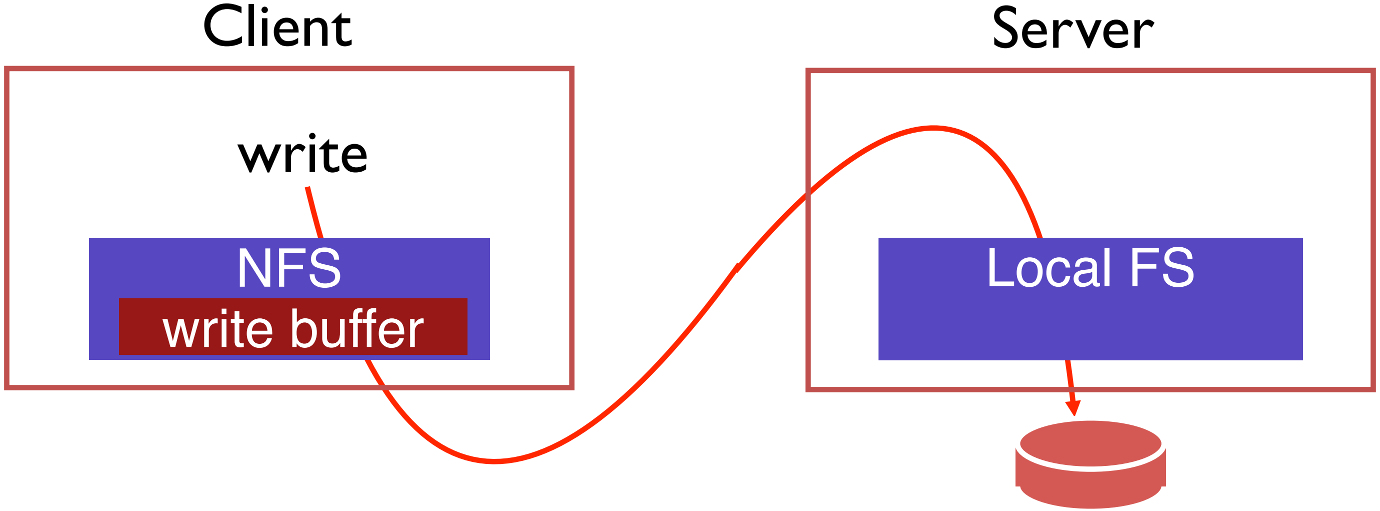
```
Client:

 write A to 0

 write B to 1

 write C to 2

 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



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

Use persistent write buffer (more expensive)

NEXT STEPS

Next class: Wrap up NFS, Summary