DISTRIBUTED SYSTEMS, NFS

Shivaram Venkataraman

CS 537, Spring 2023

ADMINISTRIVIA

Project 1 - Project 6 regrades – Last call!

Project 7 grades – this week, last regrade by Monday

Project 8 – final submissions by Thursday evening.

Midterm 3: May 8th

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

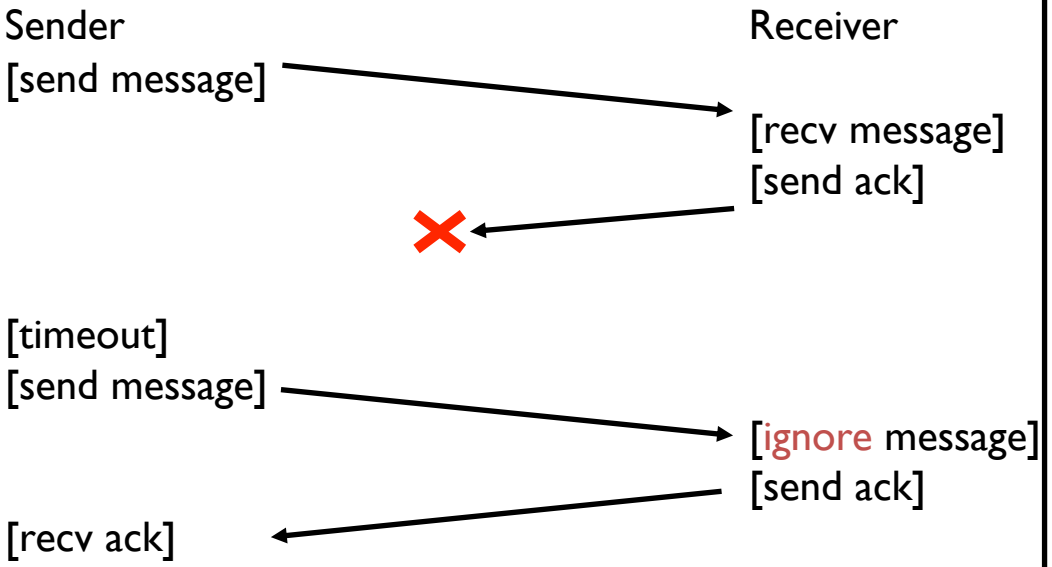
API:

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

Provide minimal reliability features:

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

TCP: ACKS, TIMEOUTS



Sequence numbers

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

Suppose message K is received.

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

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

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/deserializing

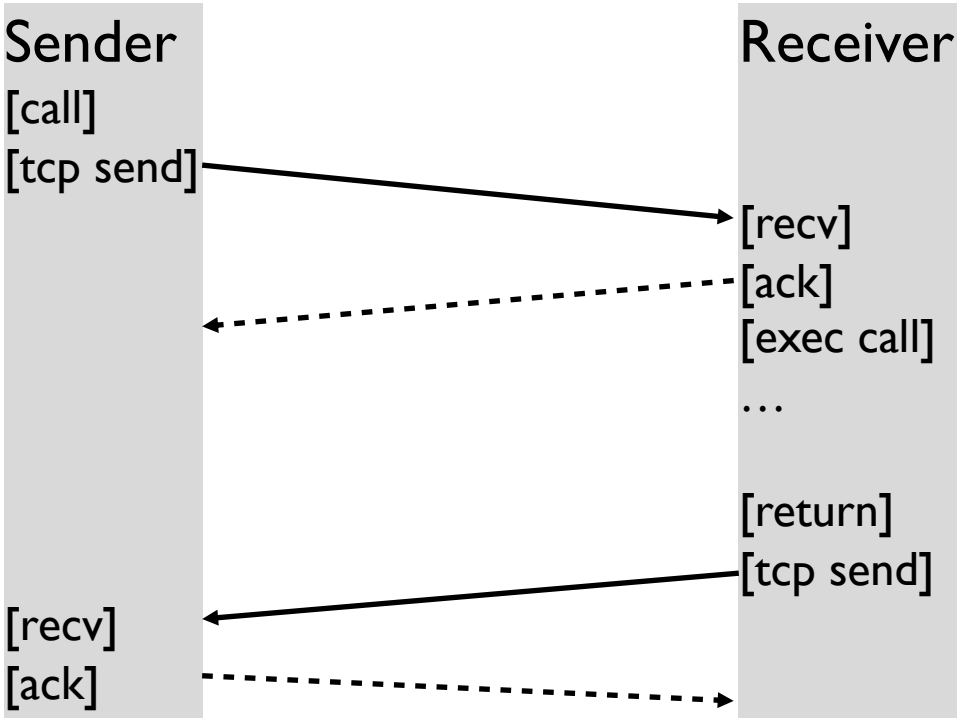
WRAPPER GENERATION: POINTERS

Why are pointers problematic?

Address passed from client not valid on server

Solutions? Smart RPC package: follow pointers and copy data

RPC OVER TCP?


Sender
[call]
[tcp send]
[recv]
[ack]
Receiver
[recv]
[ack]
[exec call]
…
[return]
[tcp send]

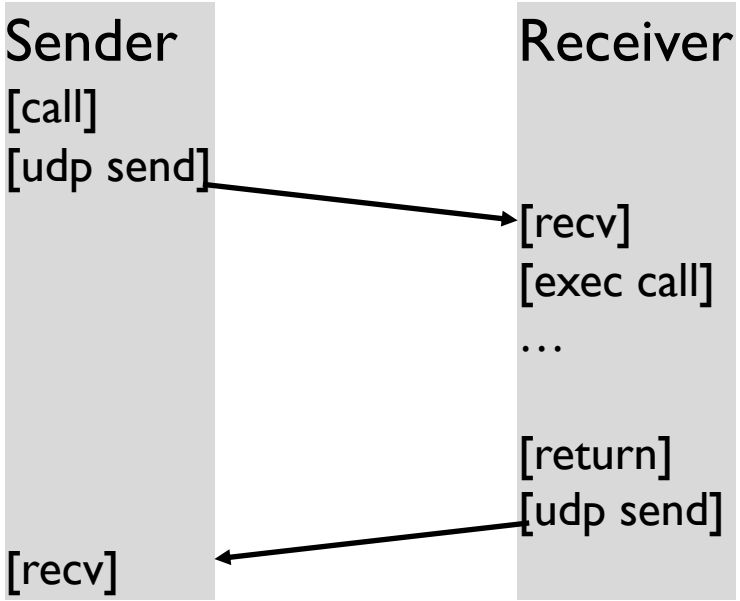
RPC OVER UDP

Strategy: use function return as implicit ACK

Piggybacking technique

What if function takes a long time?

then send a separate ACK



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

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

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

We're looking at NFSv2. NFSv4 has many changes

Why look at an older protocol? Simpler, focused goals

OVERVIEW

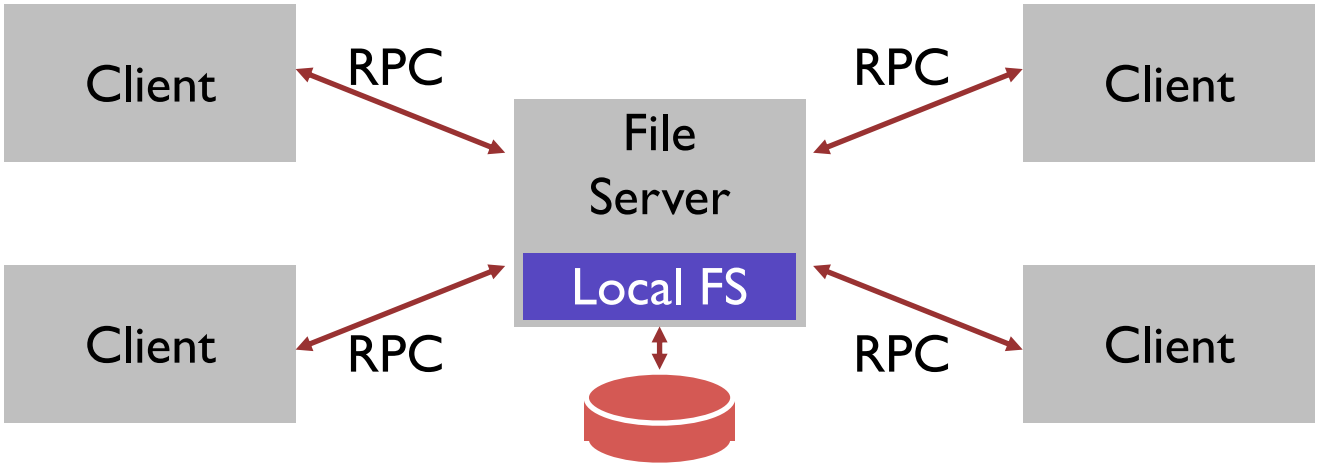
Architecture

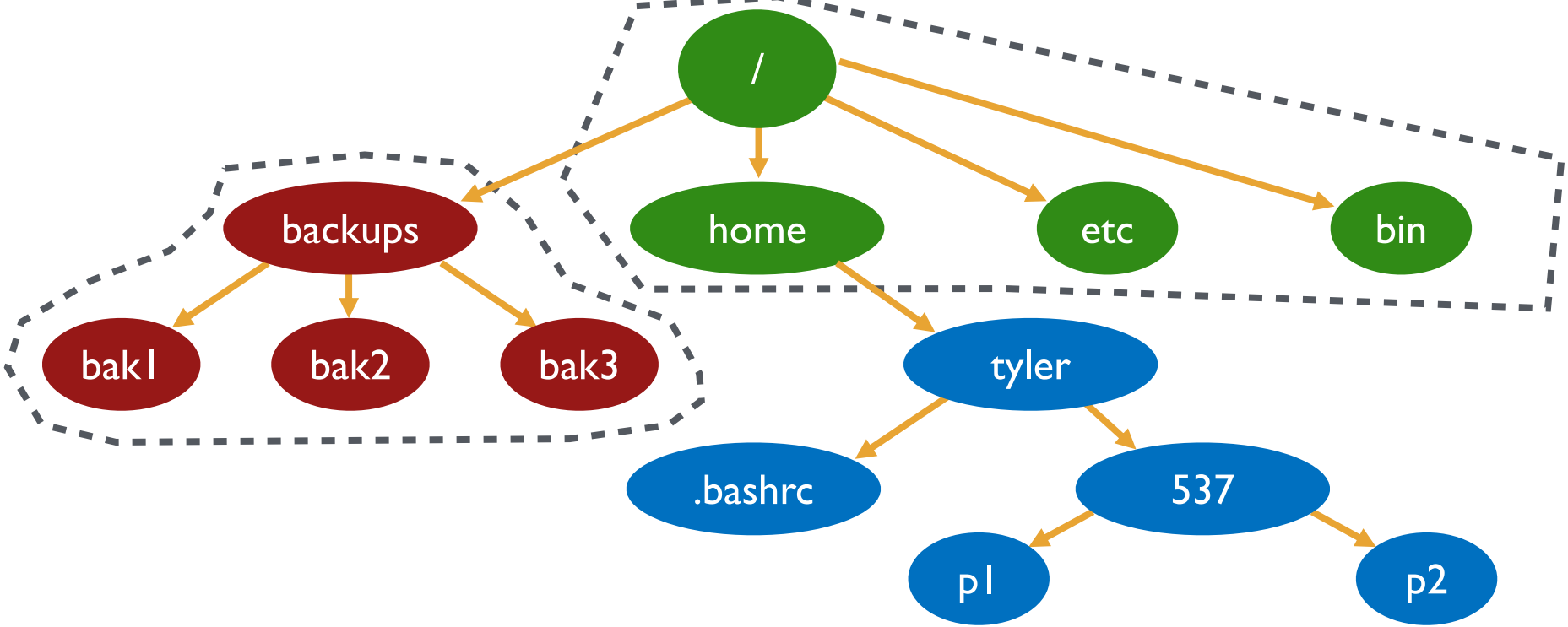
Network API

Write Buffering

Cache

NFS ARCHITECTURE

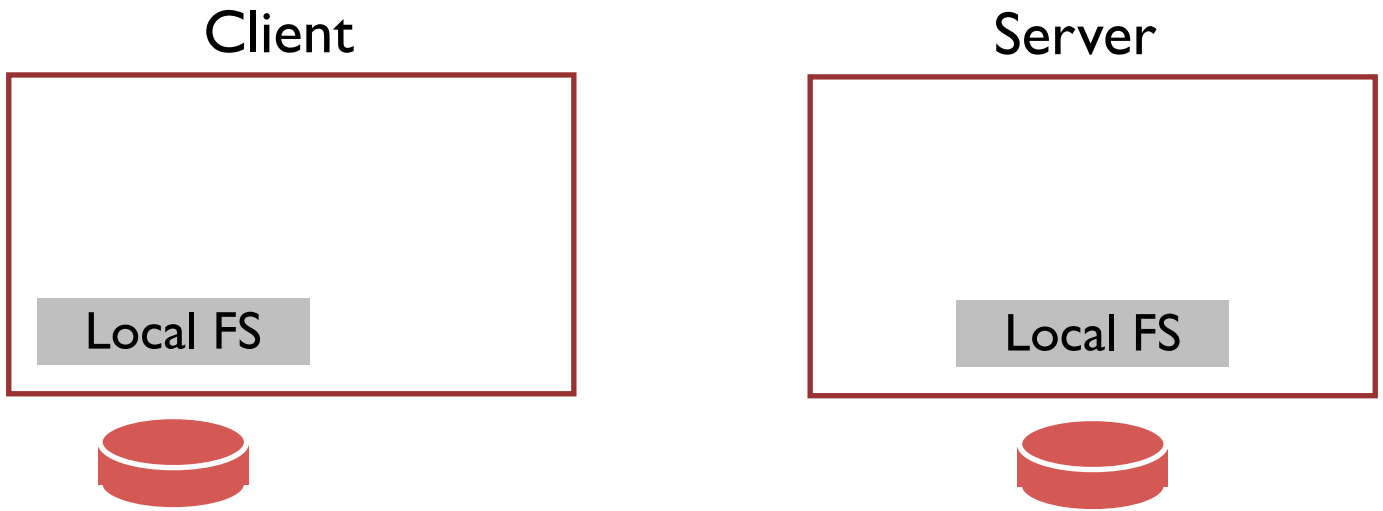




/dev/sda1 **on** /

/dev/sdb1 **on** /backups

NFS **on** /home


Client
Local FS
Server
Local FS

OVERVIEW

~~Architecture~~

Network API

Write Buffering

Cache

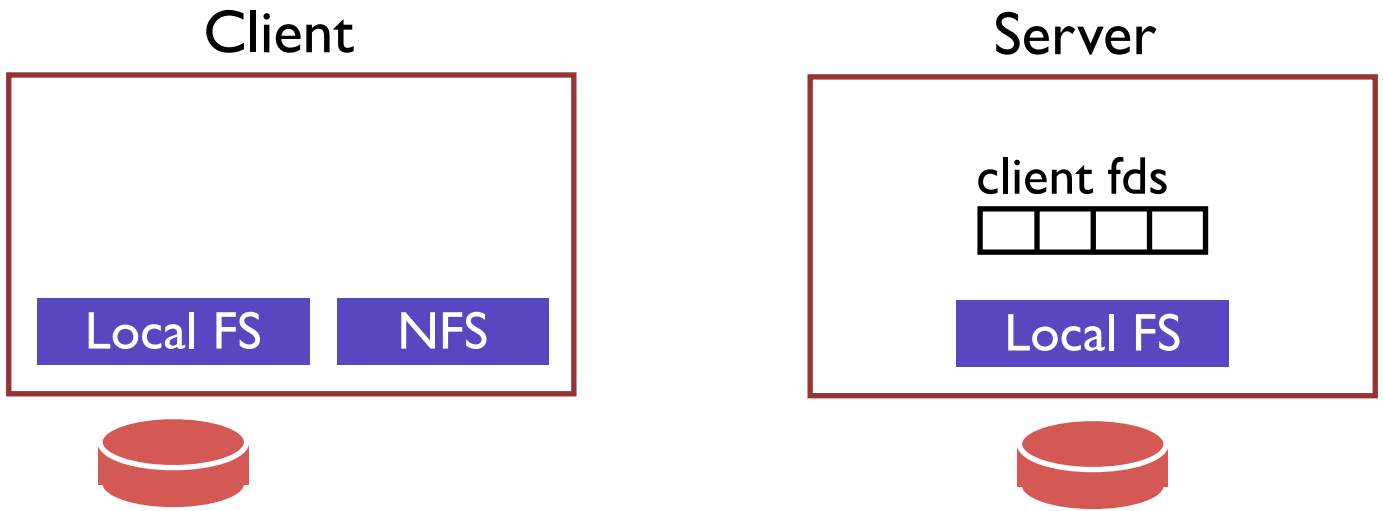
STRATEGY 1

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

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);

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

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

Pros?

Cons?

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)

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	

CAN NFS PROTOCOL INCLUDE APPEND?

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

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

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()

```
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

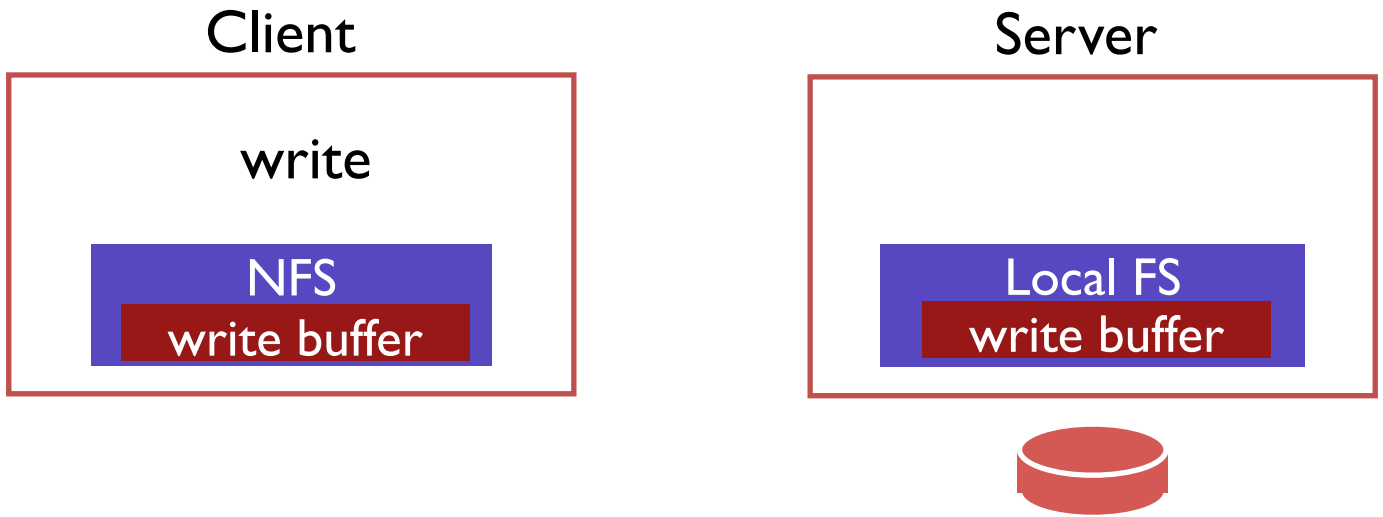
Not idempotent

- append

What about these?

- mkdir
- creat

WRITE BUFFERS

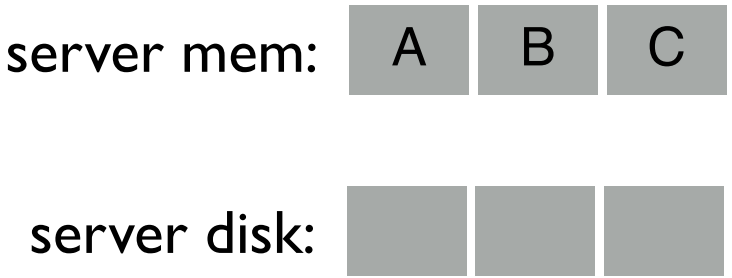


Server acknowledges write before write is pushed to disk;
What happens if server crashes?

SERVER WRITE BUFFER LOST

client:

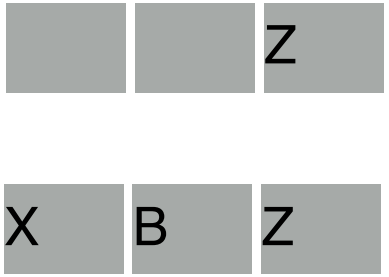
```
write A to 0
write B to 1
write C to 2
```



server acknowledges write before write is pushed to disk

SERVER WRITE BUFFER LOST

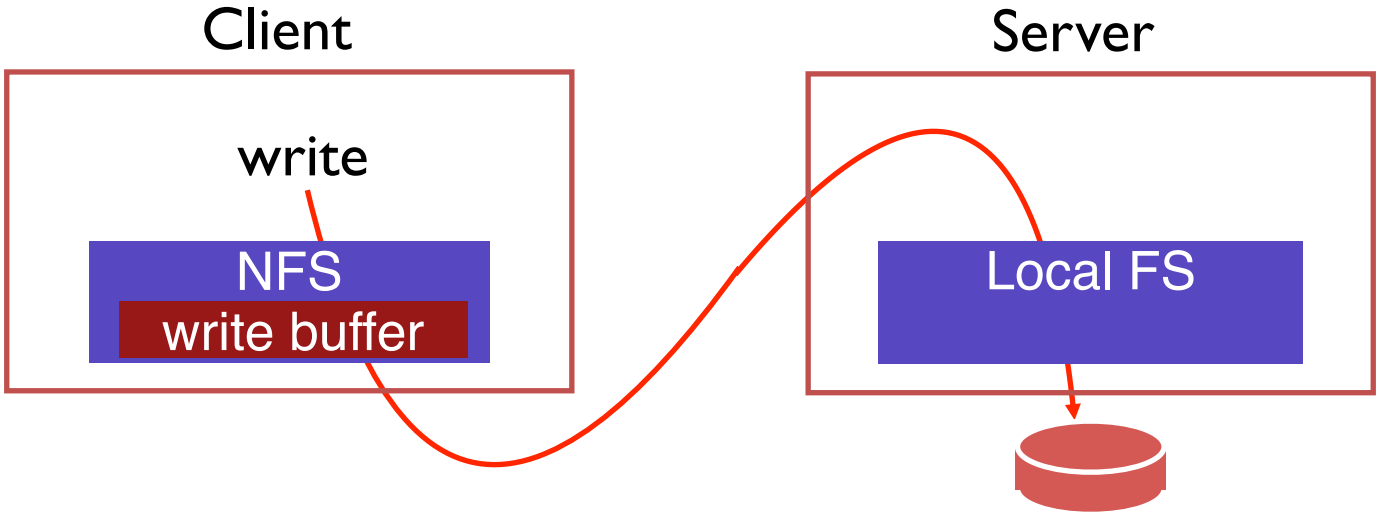
```
Client:

 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)

NEXT STEPS

Next class: Wrap up NFS, Summary