Welcome back!

3 classes to go!

DISTRIBUTED SYSTEMS, NFS

Shivaram Venkataraman

CS 537, Spring 2020

ADMINISTRIVIA

Project 5: Due today! → 10pm → Slip days last project

AEFIS feedback ~ 30%.

Optional project → Project Extra Credit

~7 days Next Wed at 10pm → No slip days

→ Individual work

April 29

Discussion today: Optional project (short?)

↳ 30 mins or so

Final Exam → Cumulative

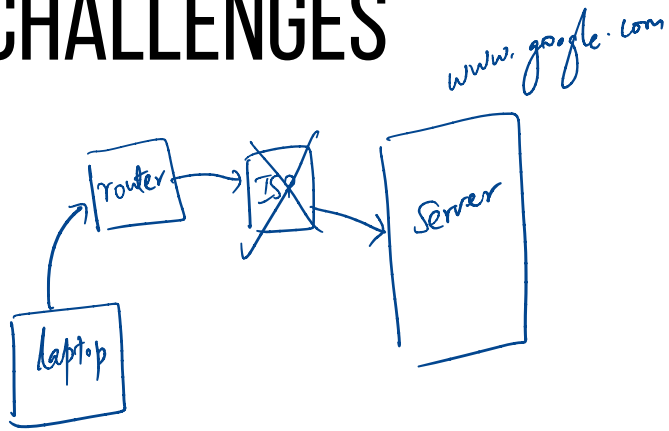
AGENDA / LEARNING OUTCOMES

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

DISTRIBUTED SYSTEMS: CHALLENGES



System failure: need to worry about partial failure

Communication failure: links unreliable

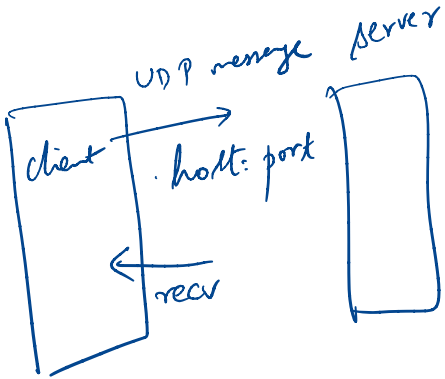
- bit errors
- packet loss
- node/link failure

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

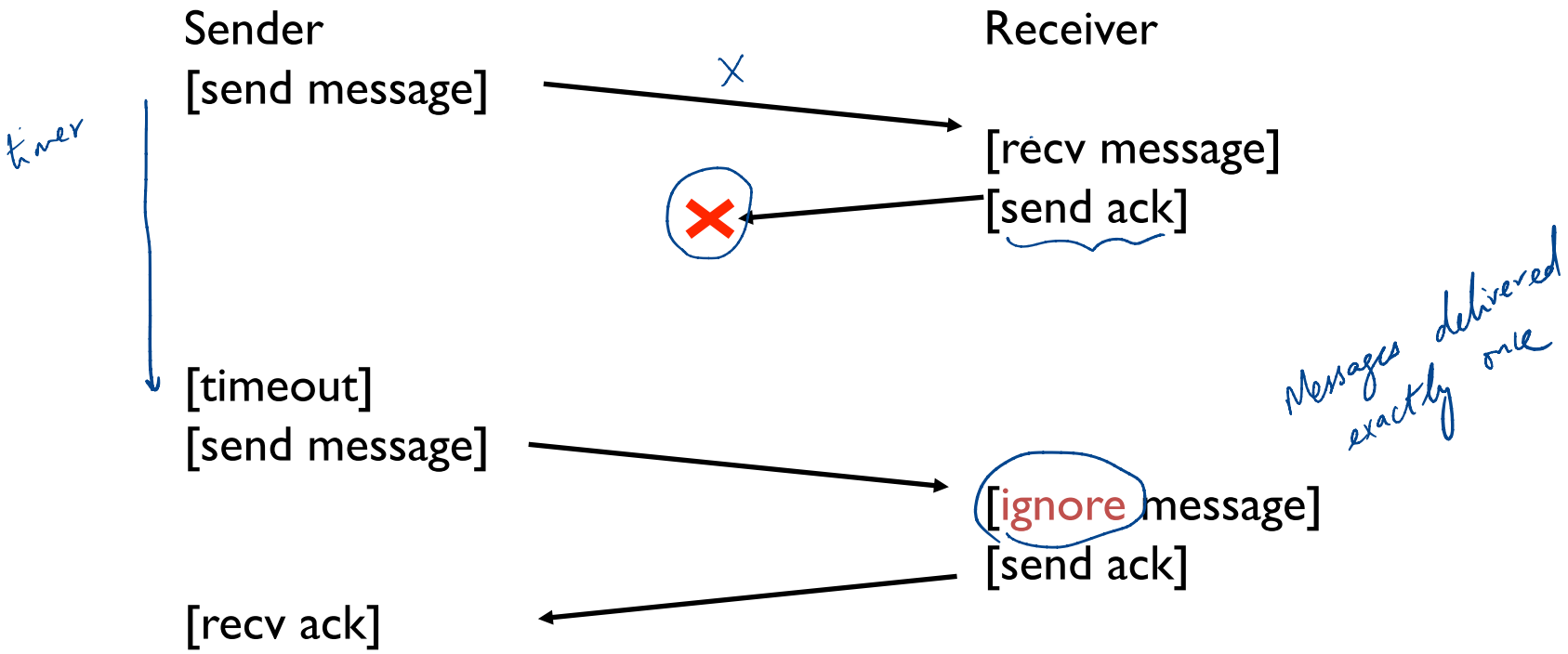


Provide minimal reliability features:

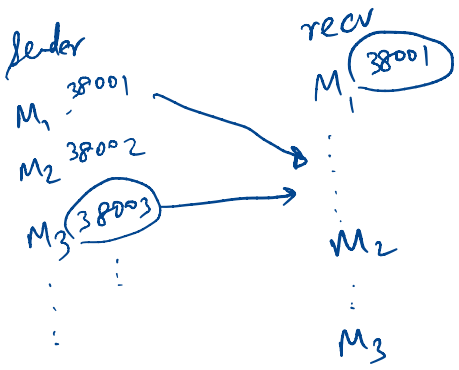
- messages may be lost
- messages may be reordered
- messages may be duplicated
- only protection: checksums to ensure data not corrupted

Reliable message delivery

TCP: ACKS, TIMEOUTS



TCP: SEQUENCE NUMBERS



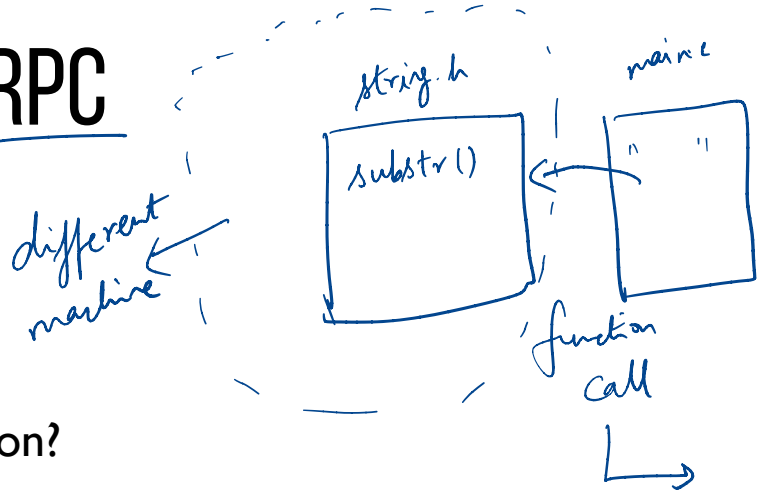
Sequence numbers

- 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, Msg K is already delivered, ignore it
- if K = N + 1, first time seeing this message
- if K > N + 1 ? buffer this message

RPC



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

What could be easier than calling a function?

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

simplifies application development

RPC

Client

Machine A

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

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

client wrapper

Exception

4

Same args return type

Server

Machine B

```
int foo(char *msg) {
        …
}

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

server wrapper

4 = foo(hello)

HTTP
↳ GET
↳ POST

msg ("hello")

msg (4)

RPC TOOLS

int foo(String {
send msg
x = recv return values.
return x;
}

more parallelism when sending RPCs

RPC packages help with two components

(1) Runtime library

- Thread pool
- Socket listeners call functions on server

work to actually send / recv network data

(2) Stub generation

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

foo.proto file

int foo(string)
bool bar(int)
⋮

generate

foo_server.h

compile

foo_client.h
foo_client.java

WRAPPER GENERATION

JSON →

Wrappers must do conversions:

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

foo("hello")

"hello"

"..... bytes"

parsing

Need uniform endianness (wrappers do this)

Conversion is called marshaling/unmarshaling, or serializing/deserializing

Object of type T

Serialize (object of type T) → bytes

commonly used term in Java

WRAPPER GENERATION: POINTERS

Distributed Shared memory

Not very common

0x005234

int len = foo((struct list * head));

length of linked list

Why are pointers problematic?

Address passed from client (not valid on server)

Ptr

Virtual memory in Client machine

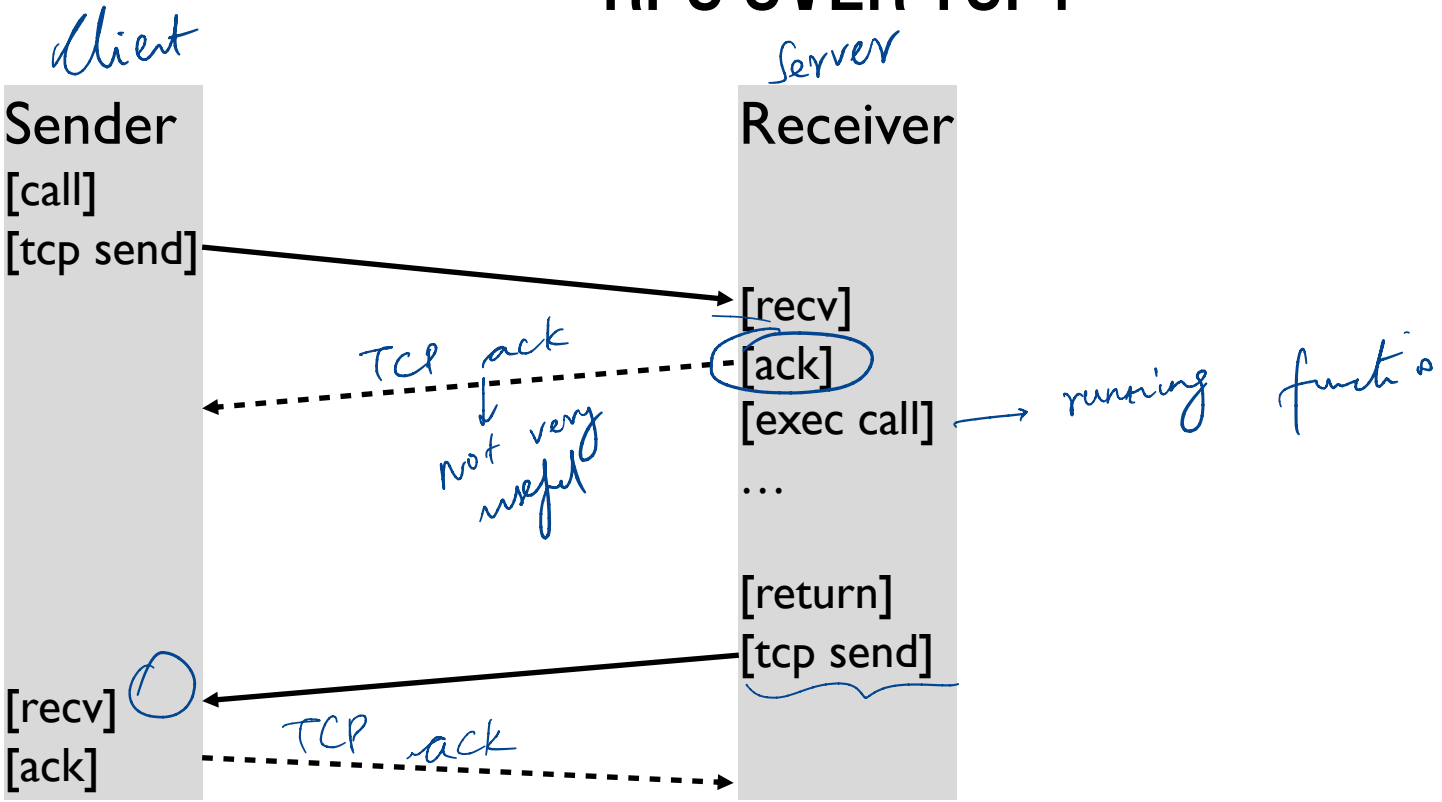
Solutions? Smart RPC package: follow pointers and copy data

Not done very often

end up copying a lot of data

RPC libraries expect you to pass concrete values.

RPC OVER TCP?

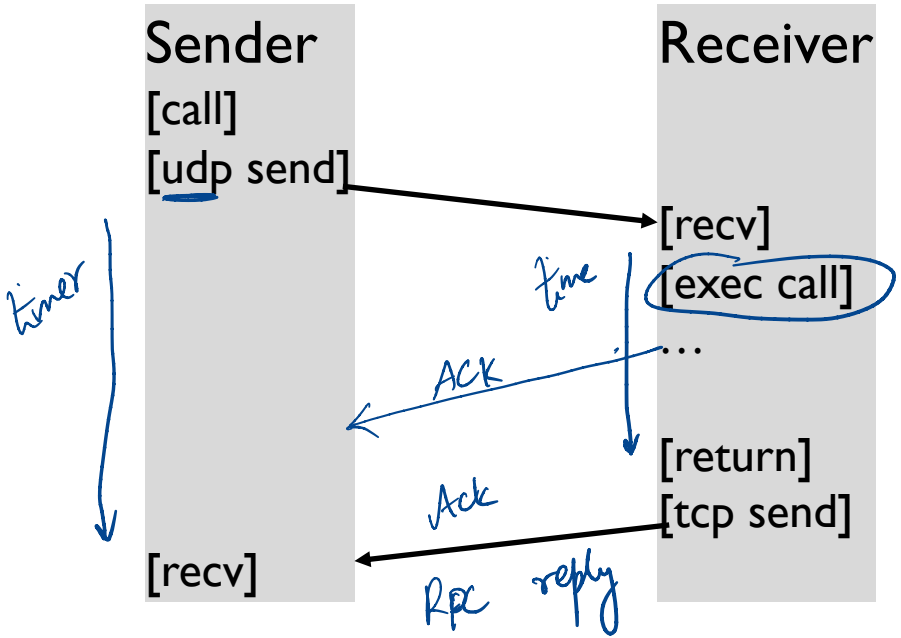

client
server
Sender
Receiver
[call]
[tcp send]
[recv]
[ack]
TCP ack
not very useful
[exec call]
running function
...
[return]
[tcp send]
[recv]
[ack]
TCP ack

RPC OVER UDP

Strategy: use function return as implicit ACK

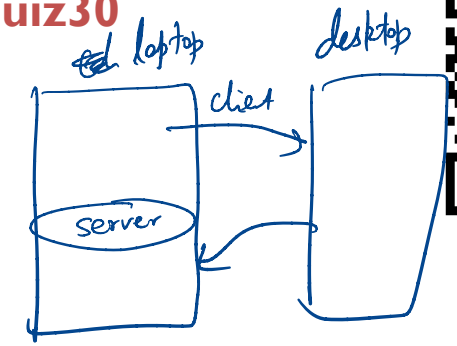
Piggybacking technique

What if function takes a long time?
then send a separate ACK



QUIZ 30

https://tinyurl.com/cs537-sp20-quiz30

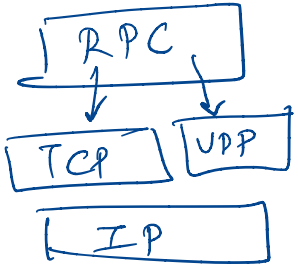


Who can initiate a Remote Procedure Call?

Client starts RPC to server

Which are the functions of client wrapper generator?

- Serializes function arguments
- Deserializes return values
- Returns from client stub to client



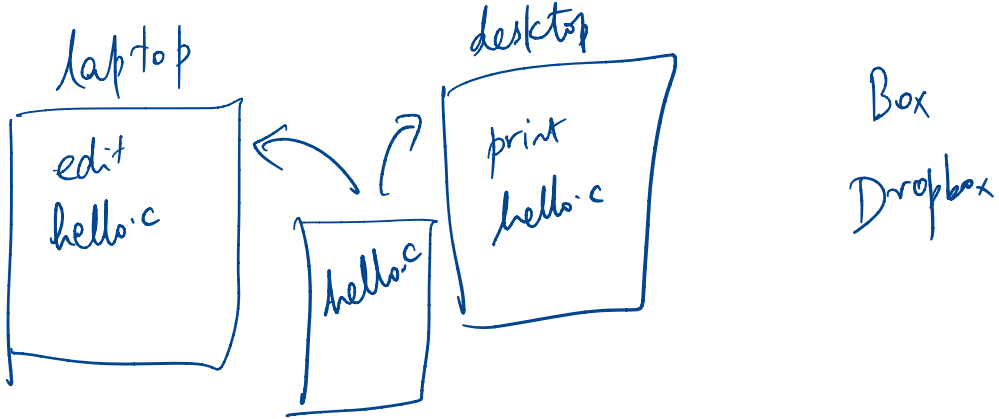
Which are the functions of server wrapper generator?

- Deserialize function arguments
- Execute function call
- Serialize return value

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 open, read, write, stat etc.

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

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

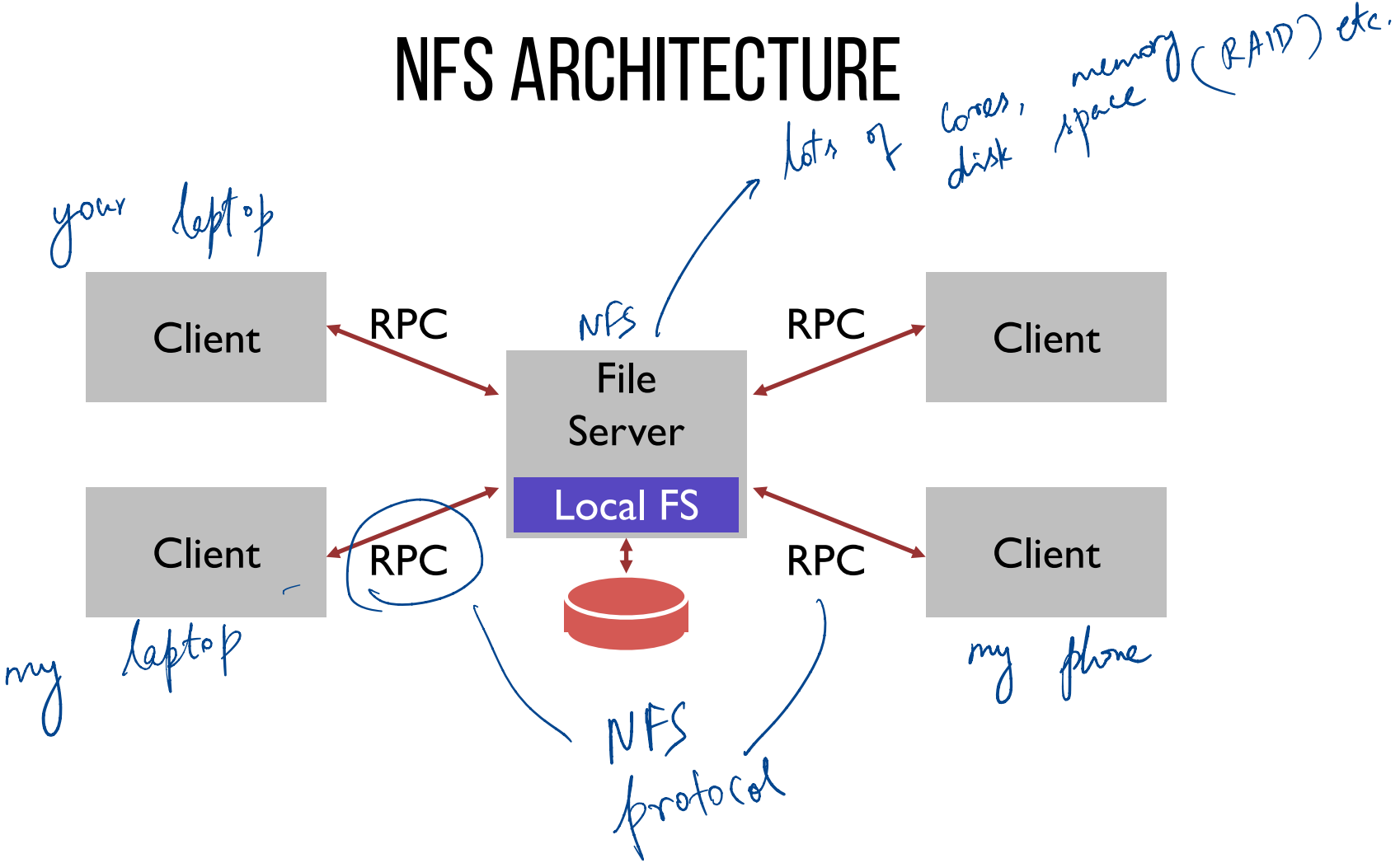
We're looking at NFSv2. NFSv4 has many changes

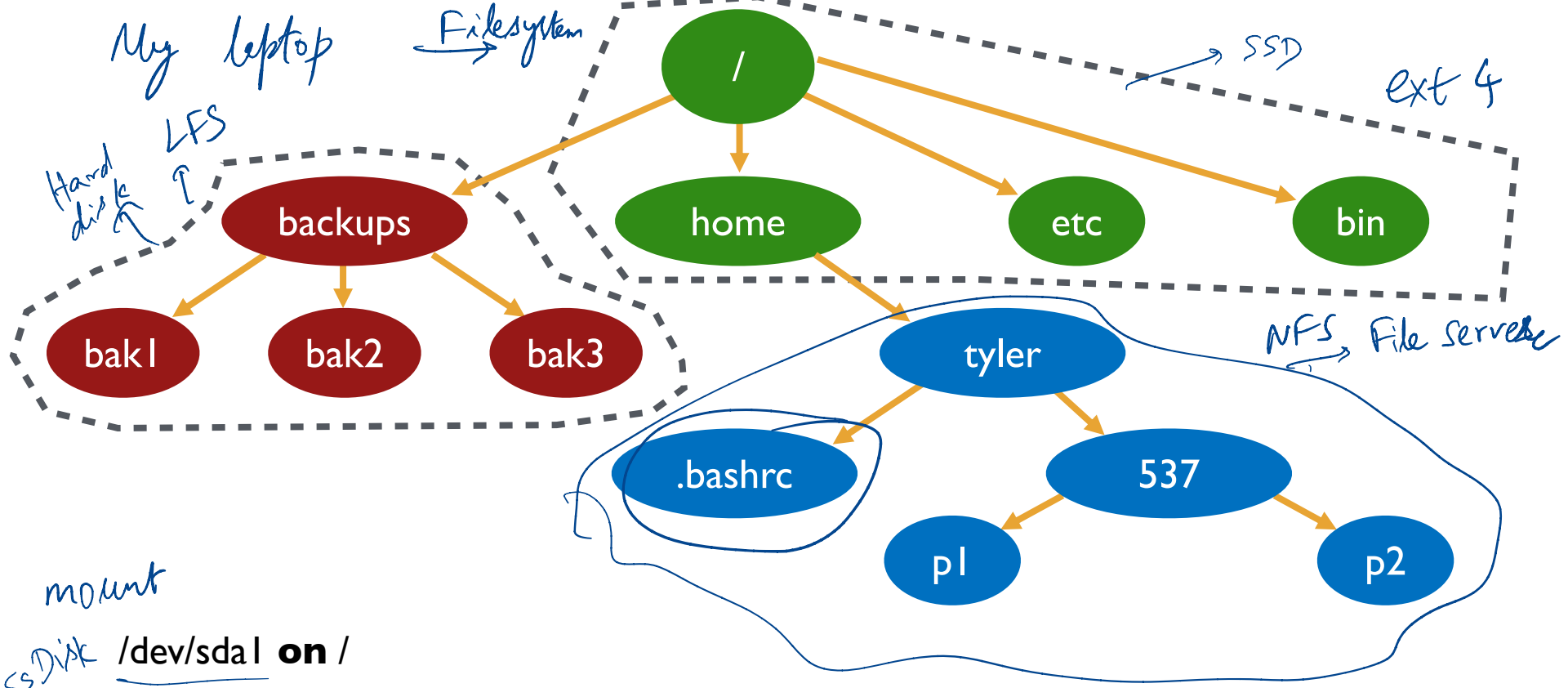
Why look at an older protocol? Simpler, focused goals

OVERVIEW

- Architecture
- Network API
- Write Buffering
- Cache

NFS ARCHITECTURE



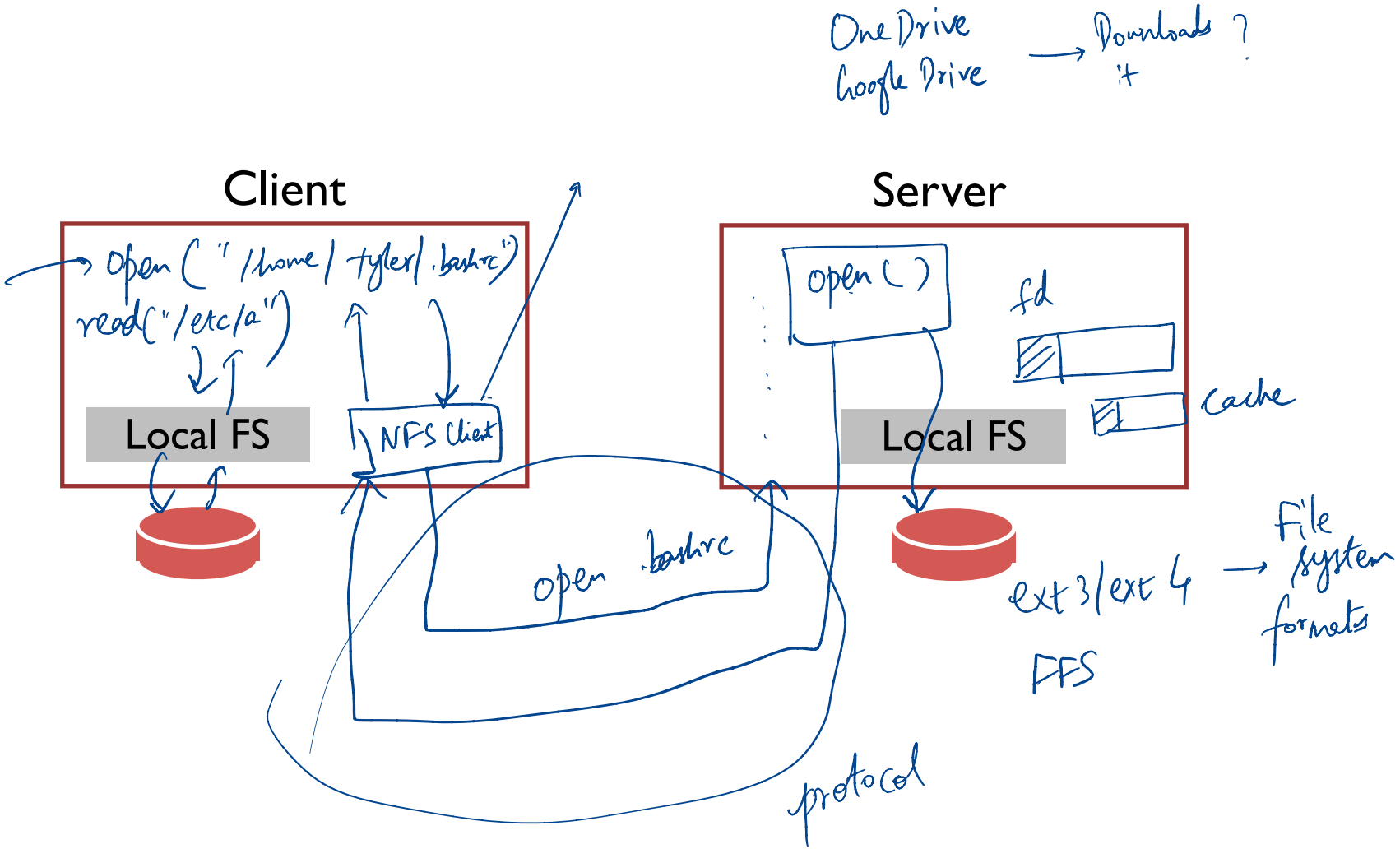


mount

SSDisk /dev/sda1 **on** /

HDD /dev/sdb1 **on** /backups

NFS **on** /home


One Drive
Google Drive
Downloads ?
Client
Server
open("/home/tyler/.bashrc")
read("/etc/a")
Local FS
NFS Client
open()
fd
cache
Local FS
open .bashrc
ext3/ext4
FFS
File system formats
protocol

OVERVIEW

~~Architecture~~

Network API

Write Buffering

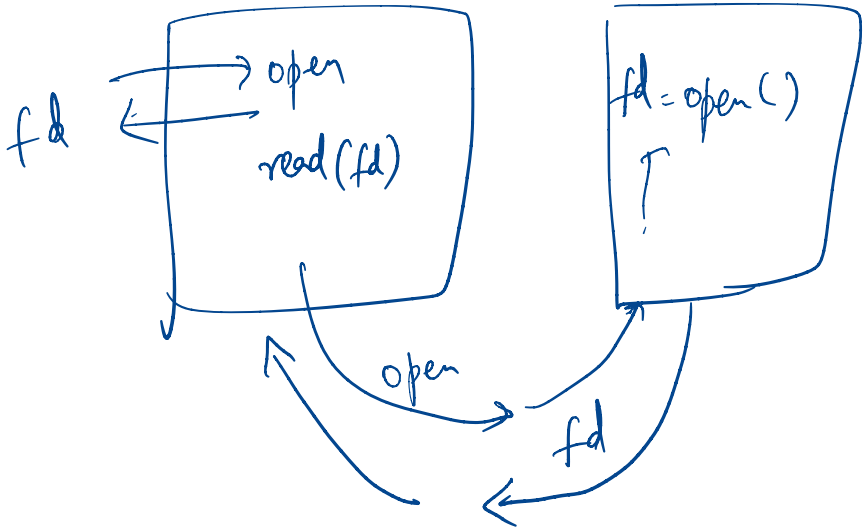
Cache

STRATEGY 1

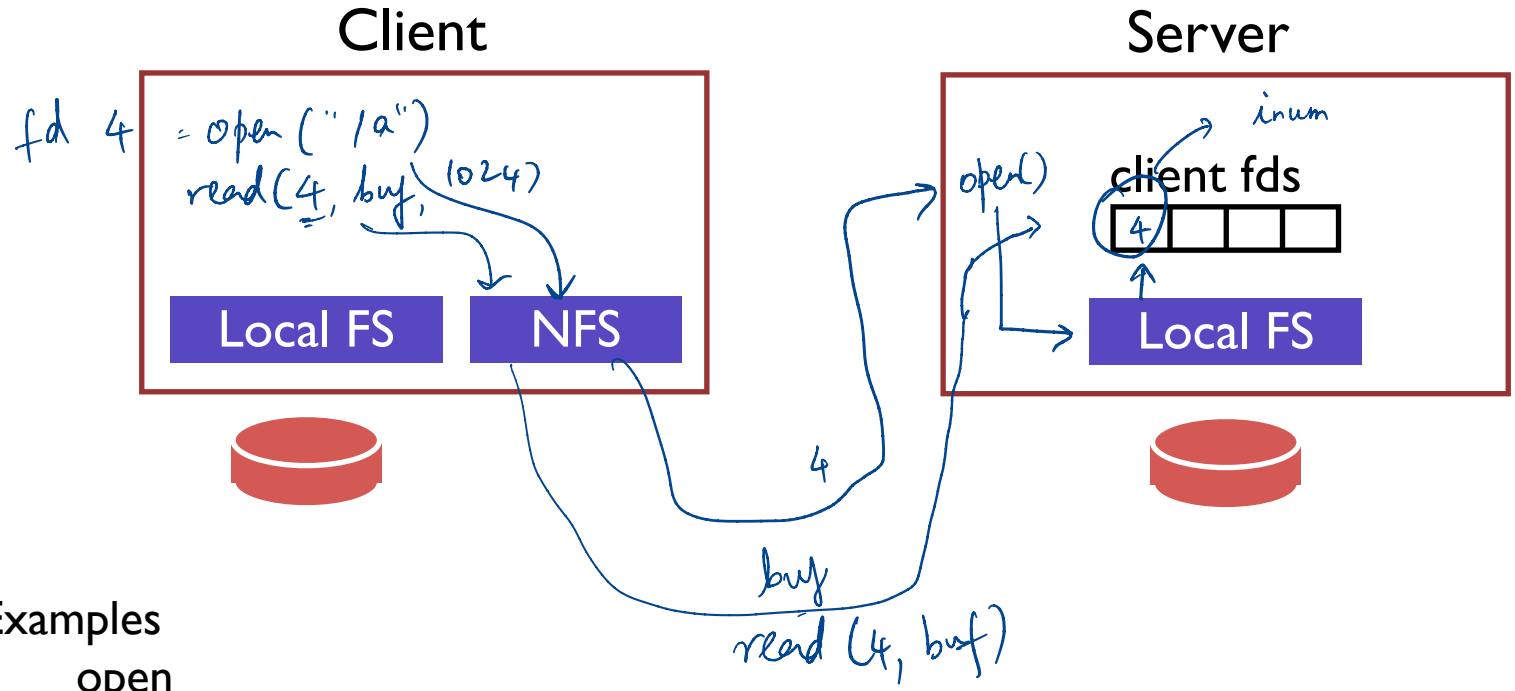
Attempt: Wrap regular UNIX system calls using RPC

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



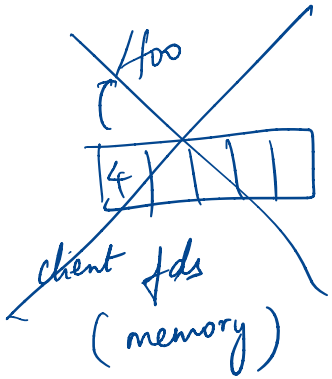
Examples

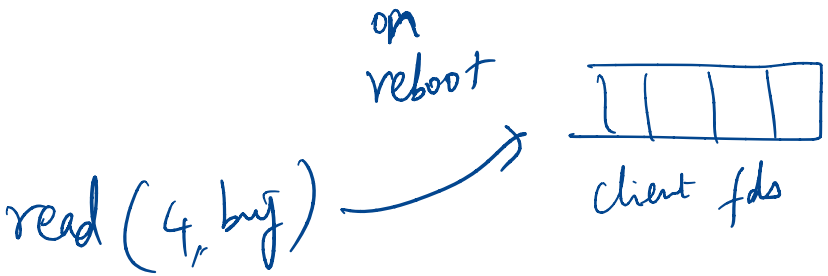
- open
- read

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 doesn't have to remember which FD is open by which client
- server still keeps other state, of course

↓

local FS

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(/home/tyler/.bashrc, buf, 100, 0)

sleep 10s

retry pread(

FD

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

Pros? Server can crash and reboot transparently

Cons? Too many path traversals

STRATEGY 3: FILE HANDLES

still part of the API

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

used to track inode reuse

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

Opaque to client (client should not interpret internals)

Which NFS volume the inode # is in

Avoid traversal cost

↳ Server side can extract inum for this file

```
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)
```

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	

NEXT STEPS

Next class: More NFS

P5 is due TODAY!

AEFIS feedback

Optional project discussion

D, E, F → disk partitions

mount

VFS layer

pread(fh, ...)

File handle

int fh = open("/foo")

server

scope of the NFS mount

file handle

/home

/backup

NFS server 1

NFS server 2

ssh → secure shell

Commands on remote files

Commands → ssh server | local FS