

ANNOUNCEMENTS

P4: Graded – Will resolve all Project grading issues this week

P5: File Systems

- Test scripts available
- Due **Due:** Wednesday 12/14 by 9 pm.
- **Free Extension Due Date:** Friday 12/16 by 9pm.
 - Extension means absolutely nothing for any reason after that!
- Fill out form if would like a new project partner

Final Exam: **Saturday 12/17 at 10:05 am**

- **Fill out exam form if academic conflicts**

Advanced Topics: Distributed File Systems (NFS, AFS, GFS)

Read as we go along: Chapter 47 and 48

UNIVERSITY of WISCONSIN-MADISON
Computer Sciences Department

CS 537
Introduction to Operating Systems

Andrea C. Arpaci-Dusseau
Remzi H. Arpaci-Dusseau

ADVANCED TOPICS: DISTRIBUTED SYSTEMS AND NFS

Questions answered in this lecture:

What is **challenging** about distributed systems?

What is the **NFS stateless protocol**?

What is **RPC**?

How can a **reliable messaging protocol** be built on unreliable layers?

What are **idempotent** operations and why are they useful?

What state is tracked on NFS clients?

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

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

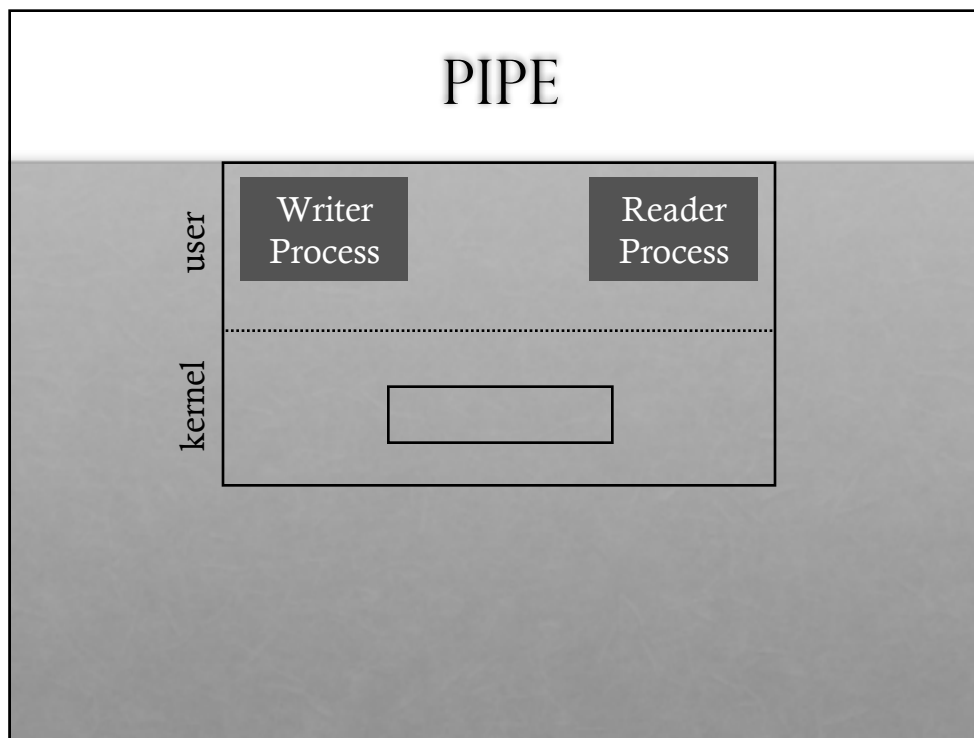
- bit errors
- packet loss
- link failure

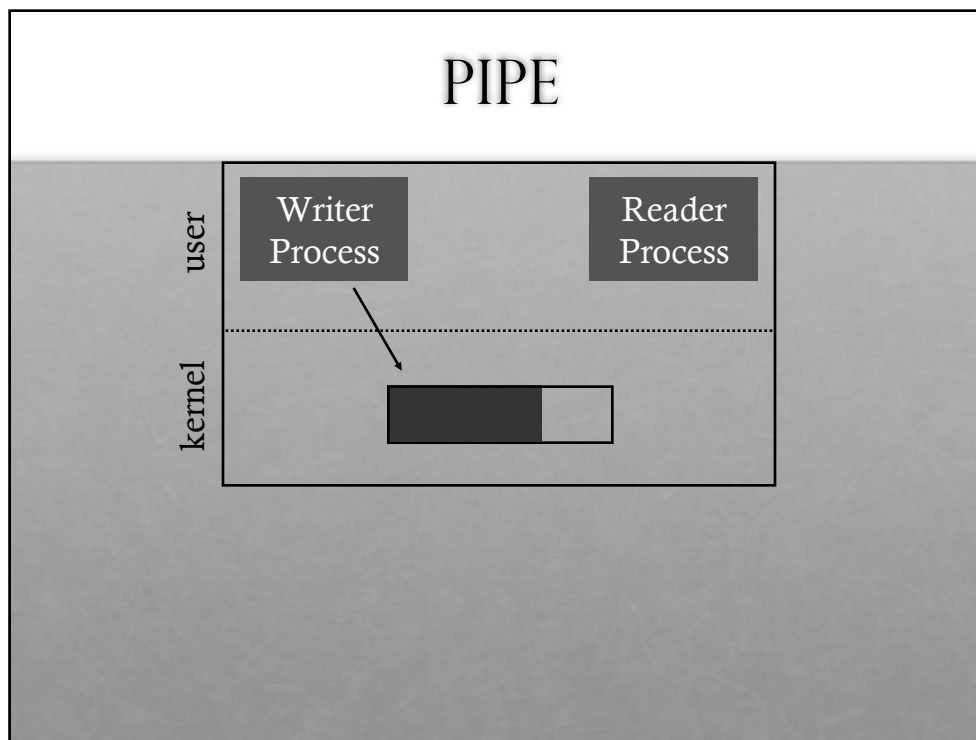
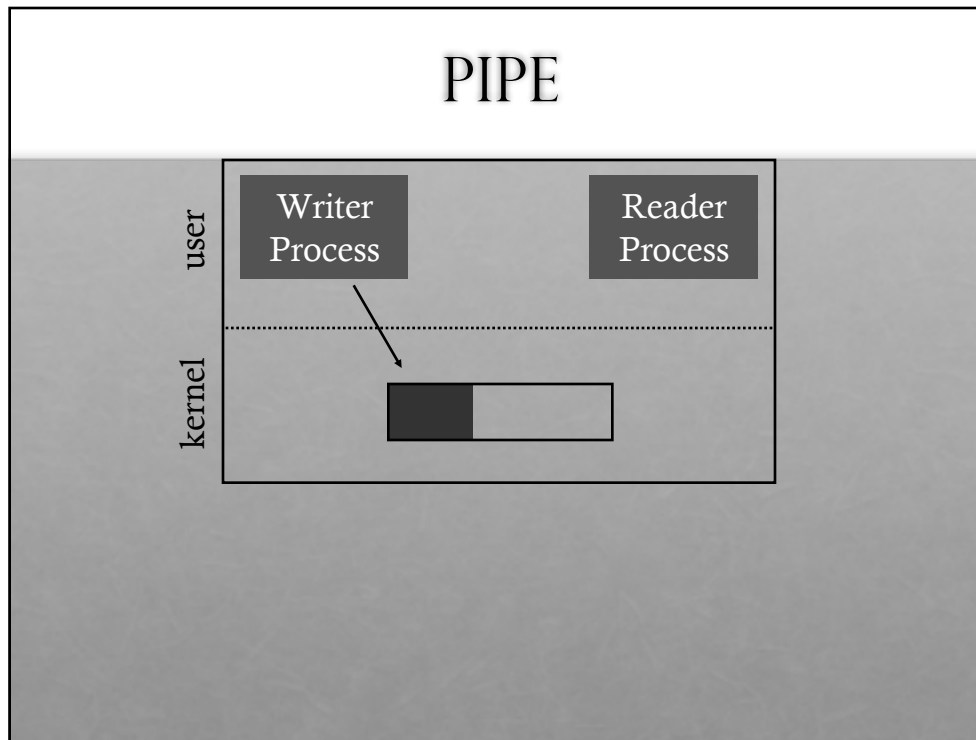
Individual nodes crash and recover

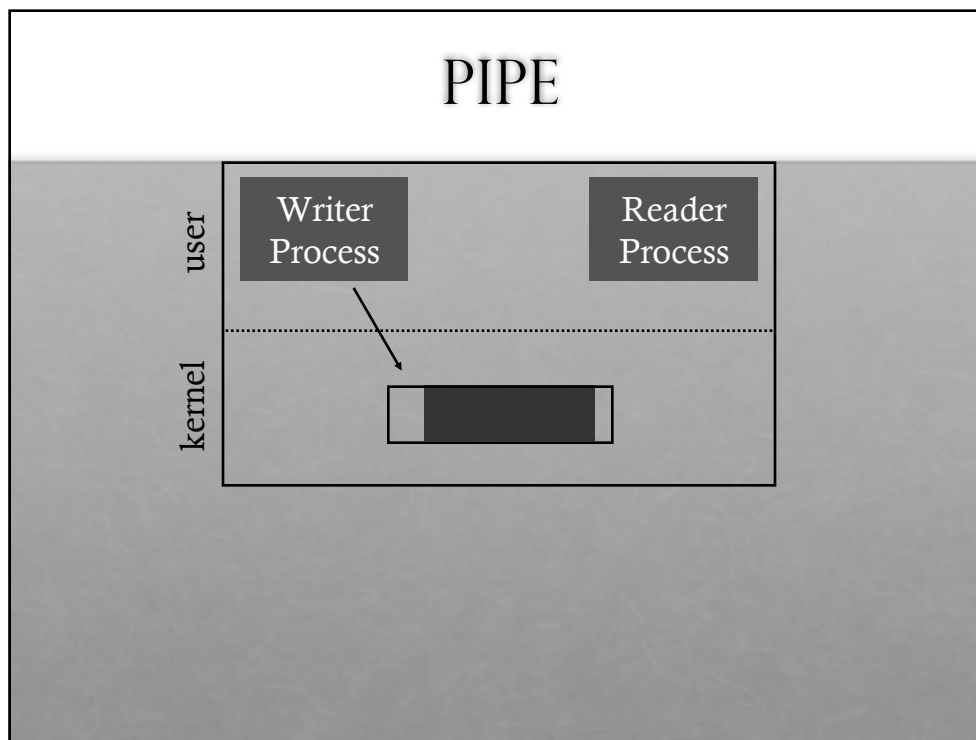
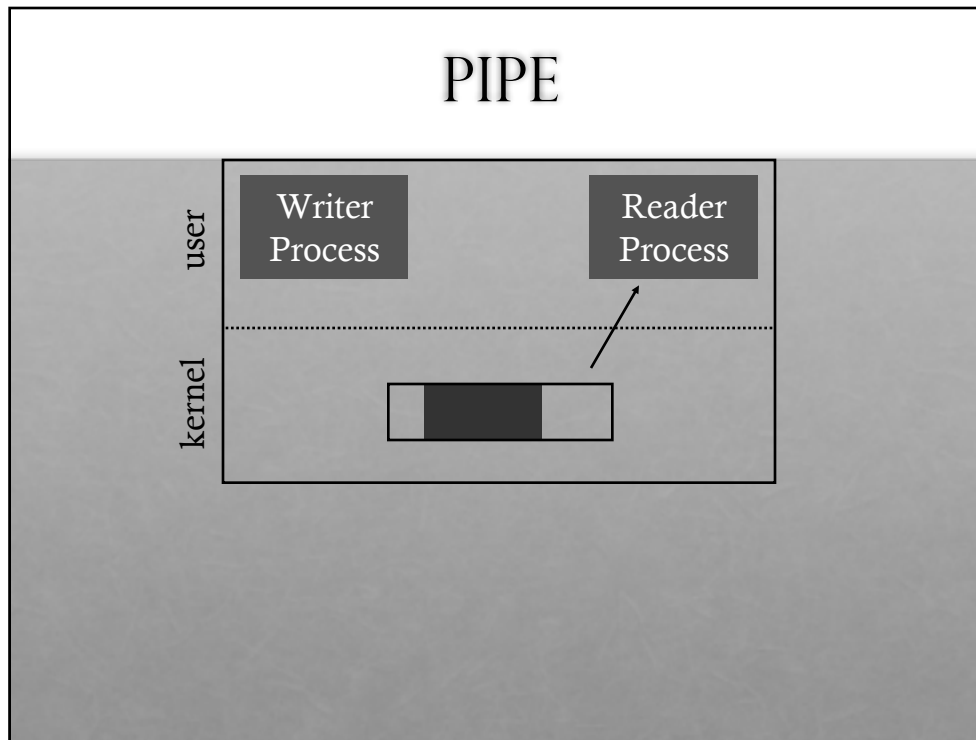
Motivation example:

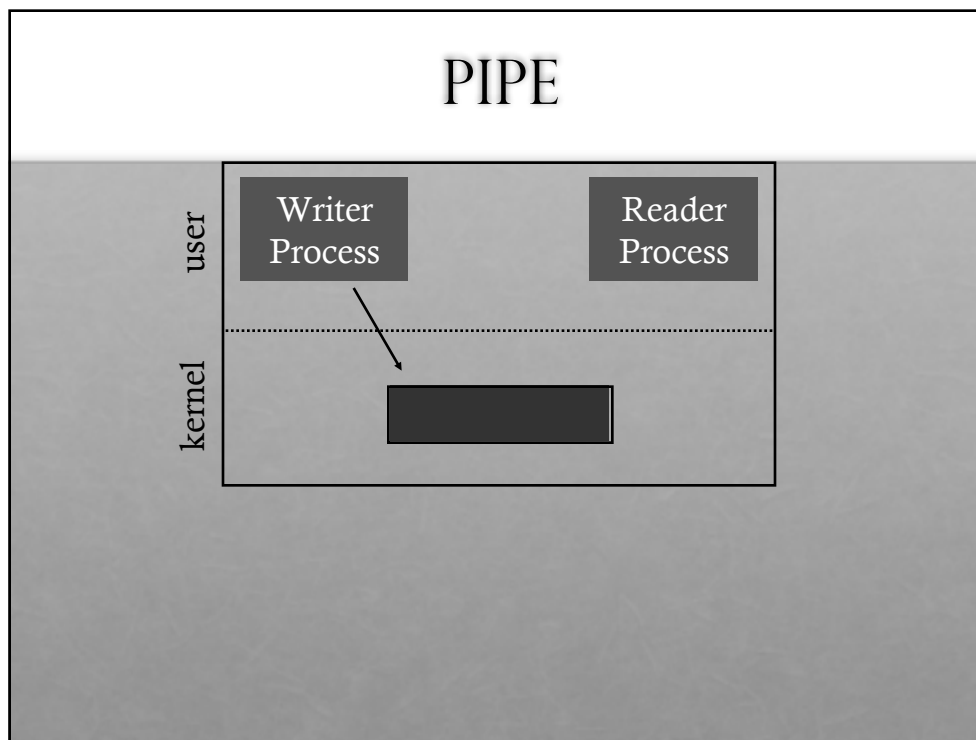
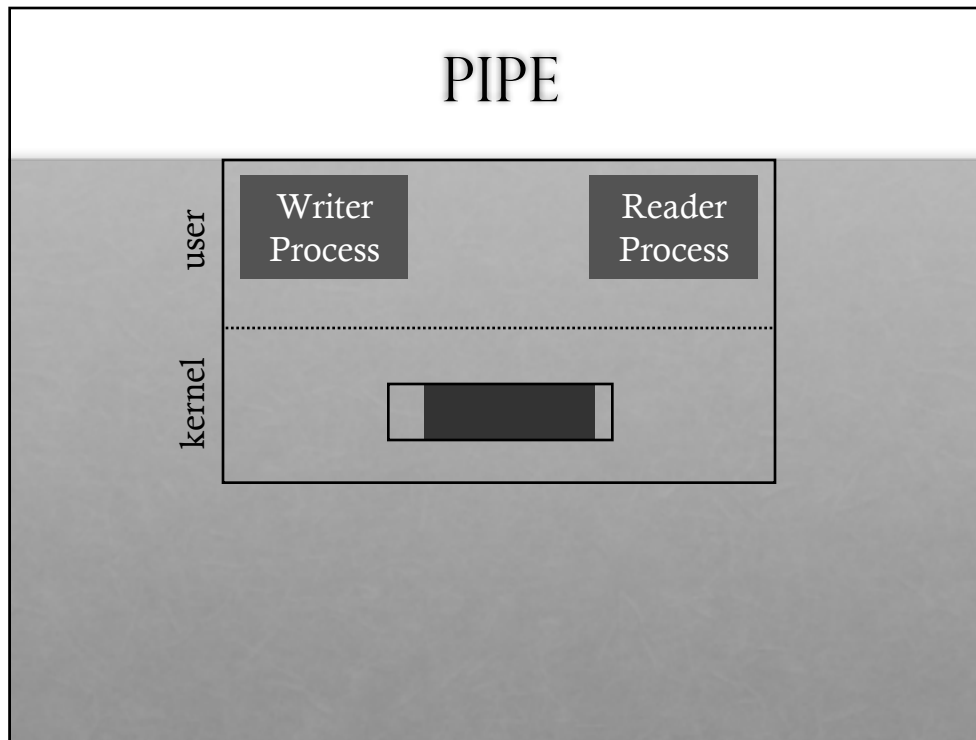
Why are network sockets less reliable than pipes?

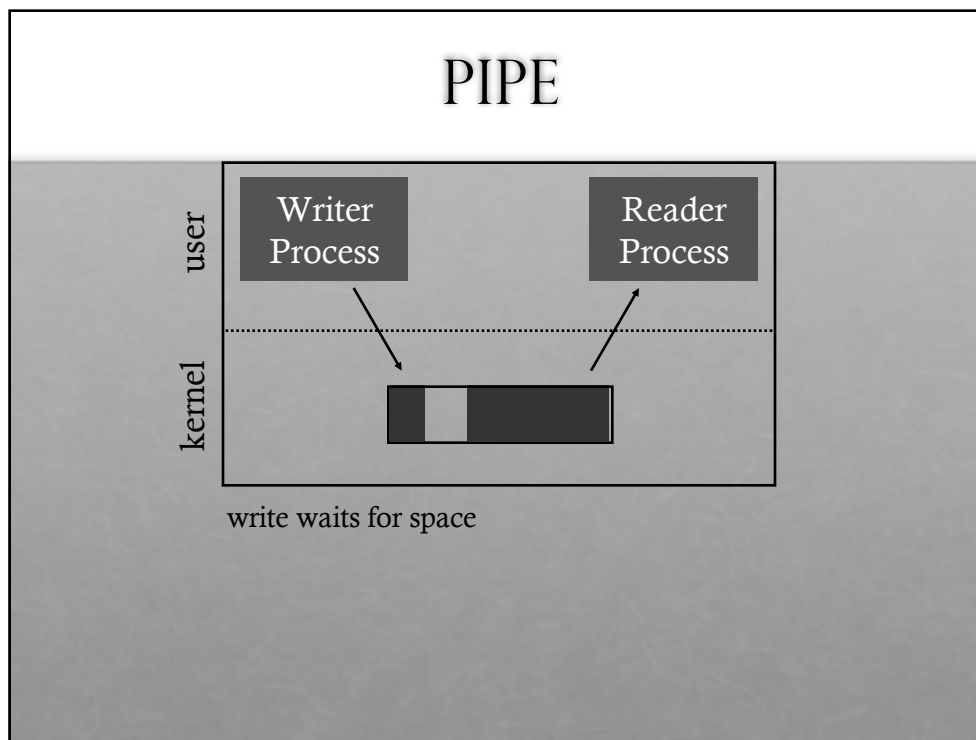
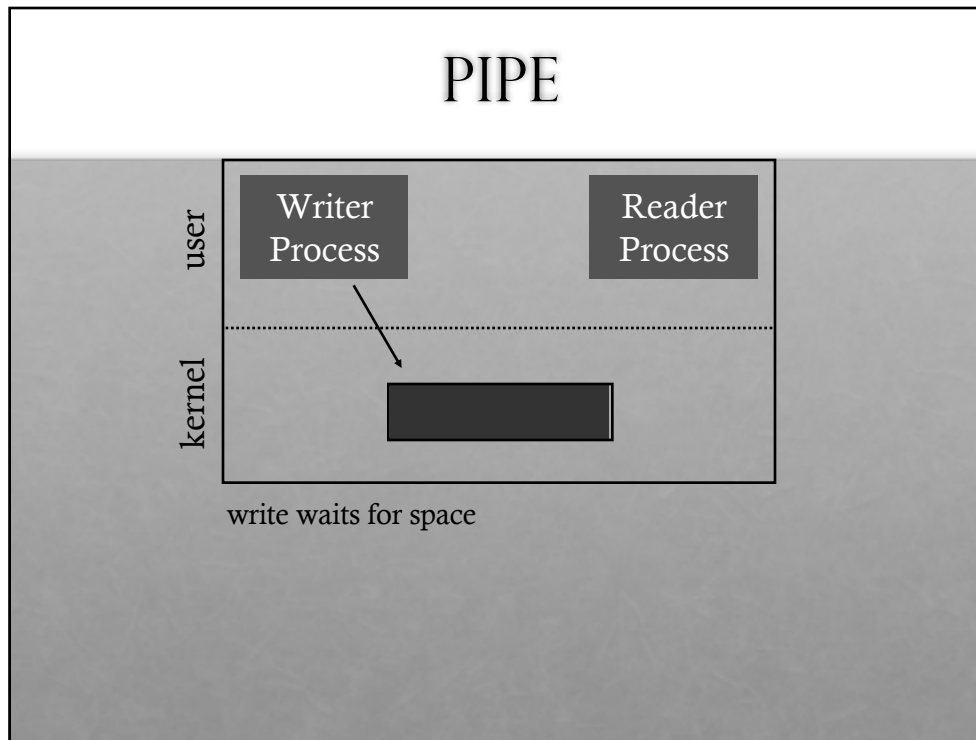
PIPE

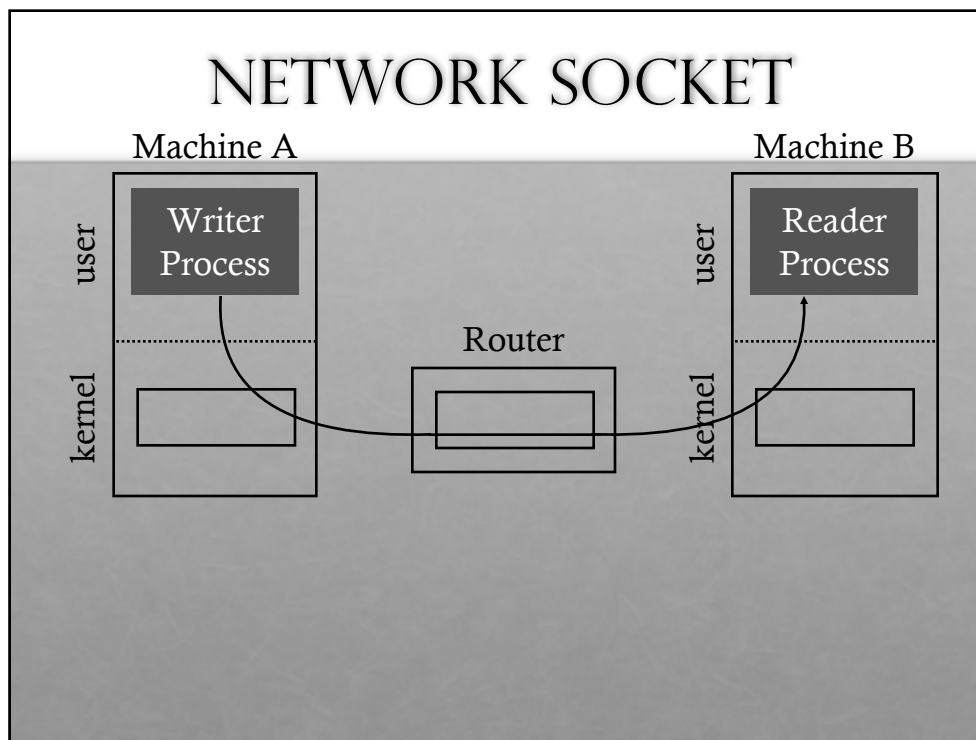
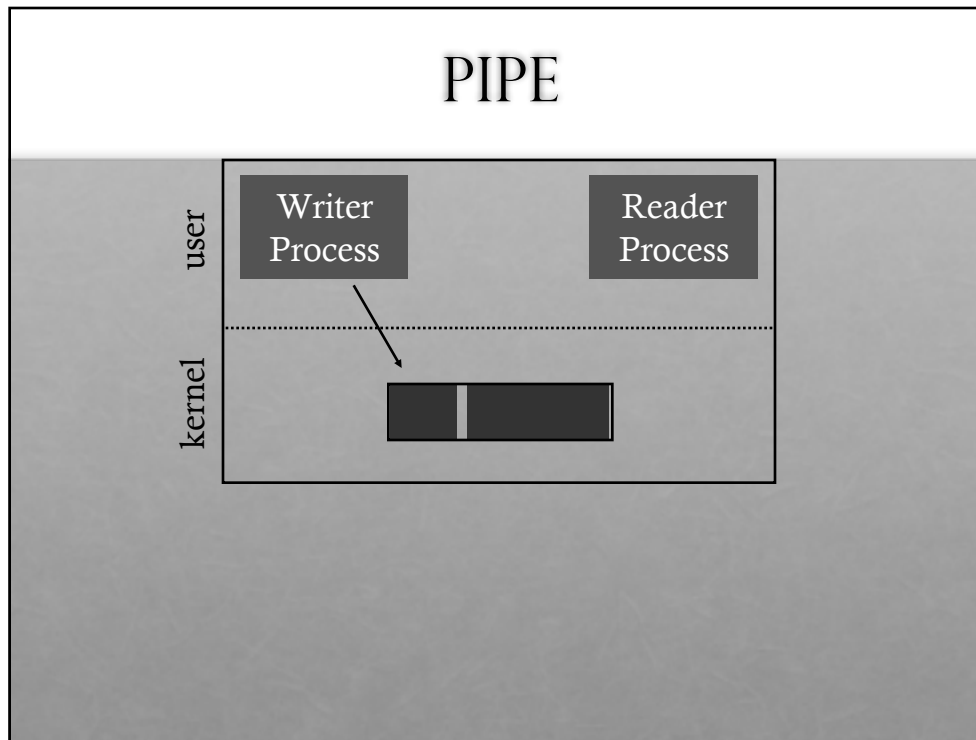


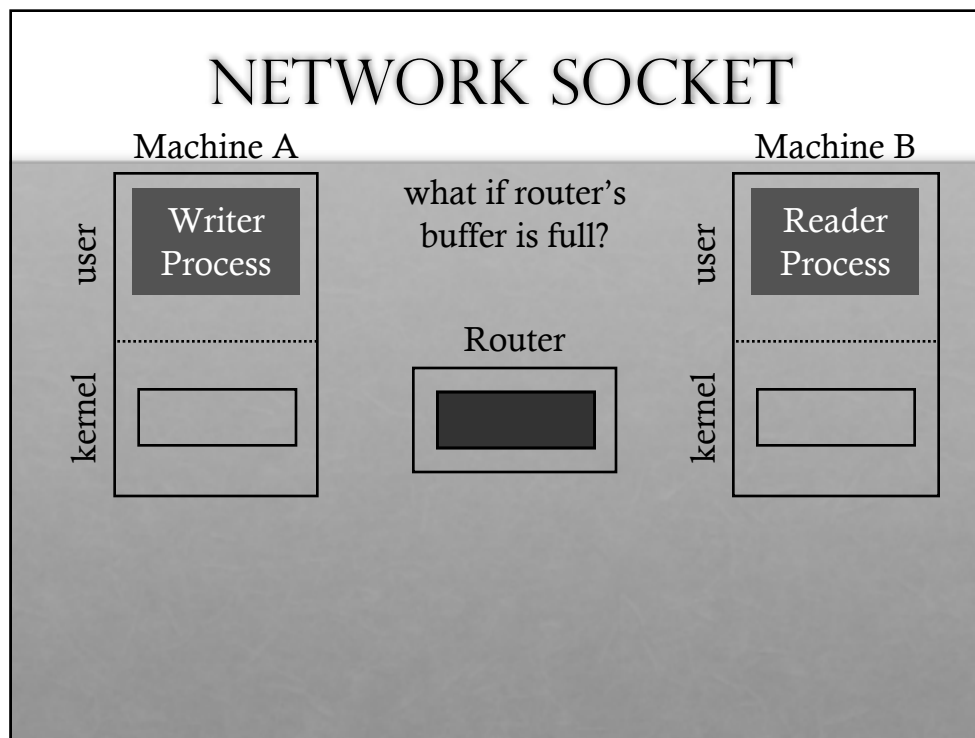
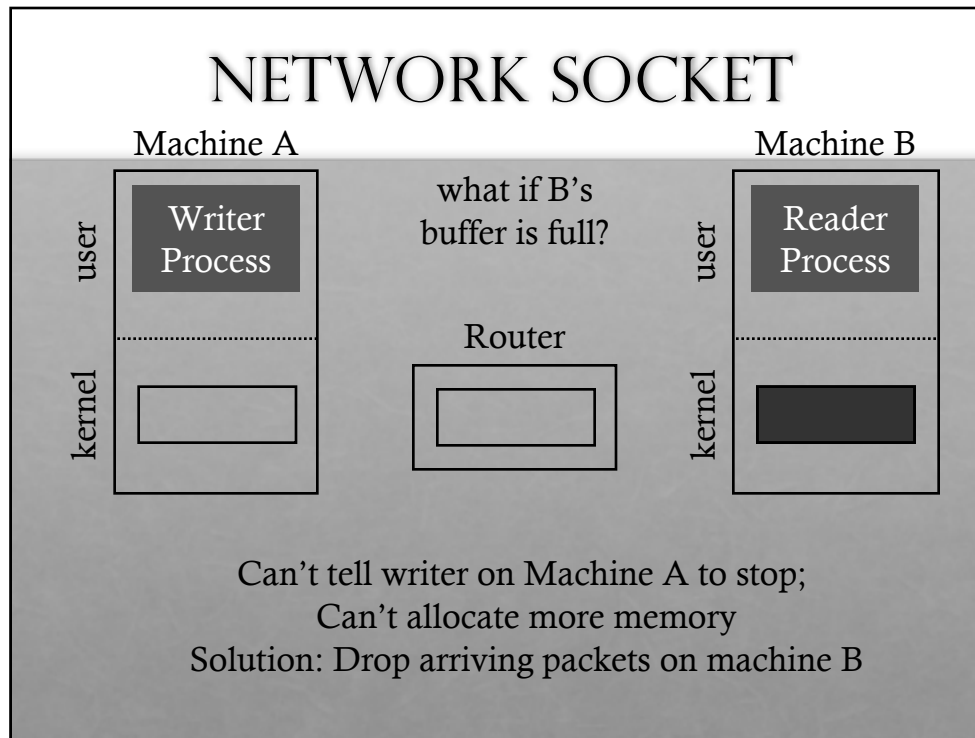




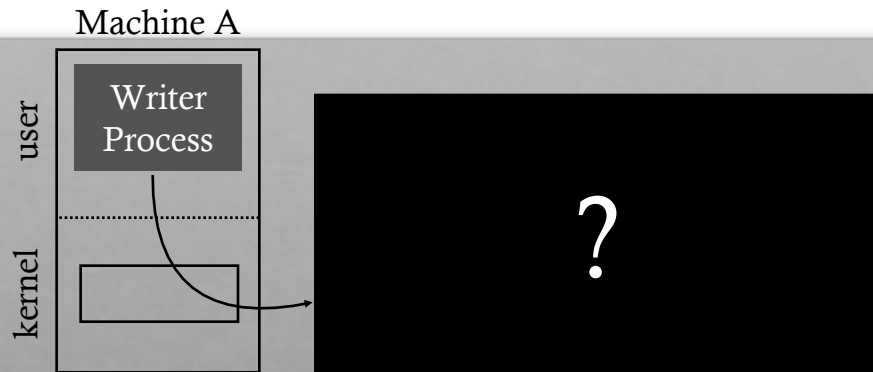








NETWORK SOCKET



Messages may get dropped, duplicated, re-ordered

DISTRIBUTED FILE SYSTEMS

File systems are great use case for distributed 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

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

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 (next lecture)

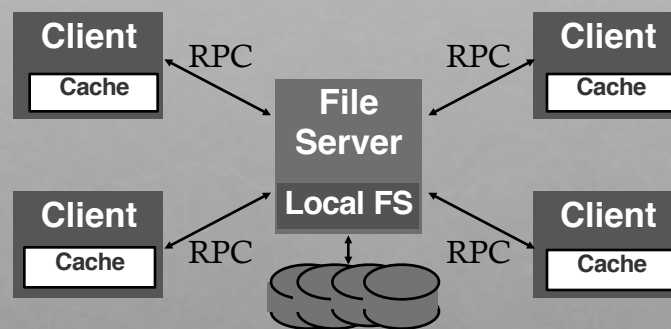
OVERVIEW

Architecture

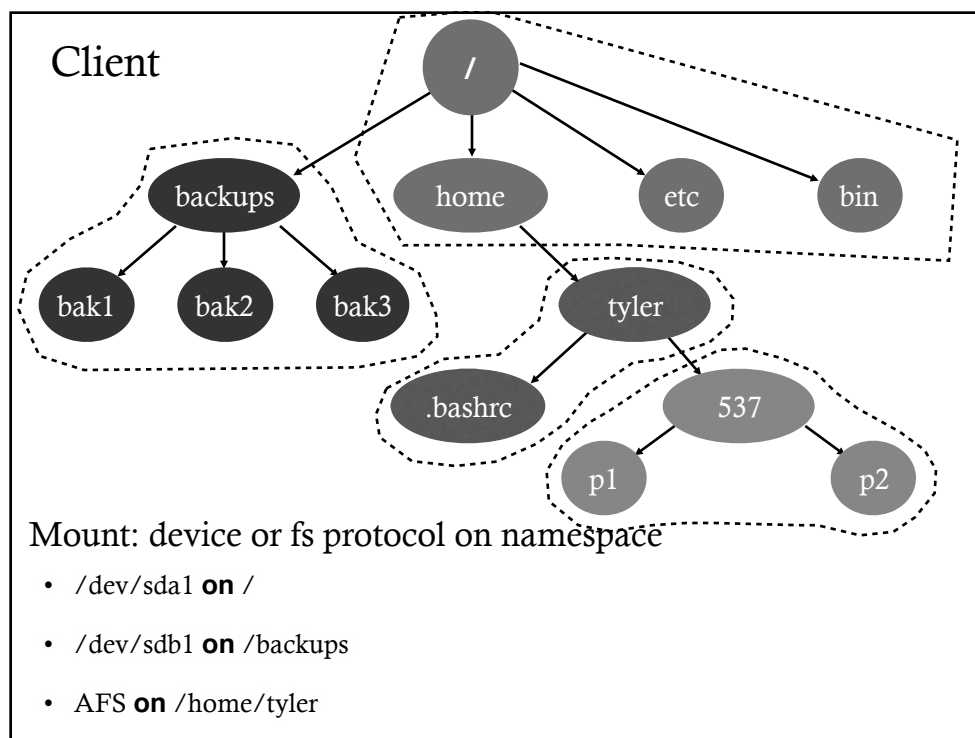
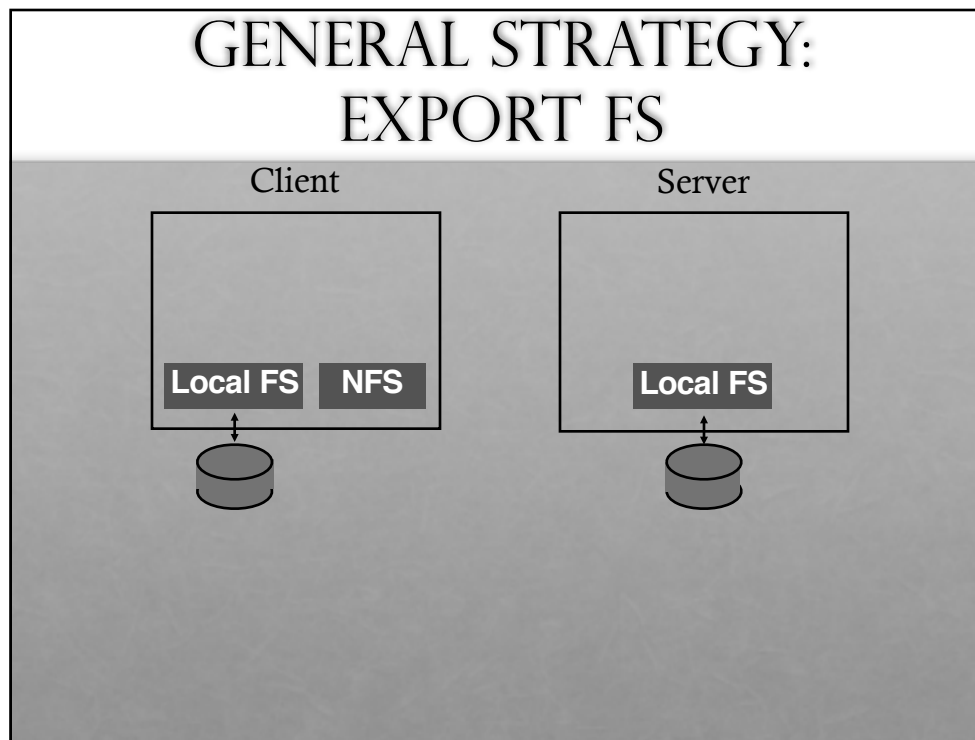
Network API

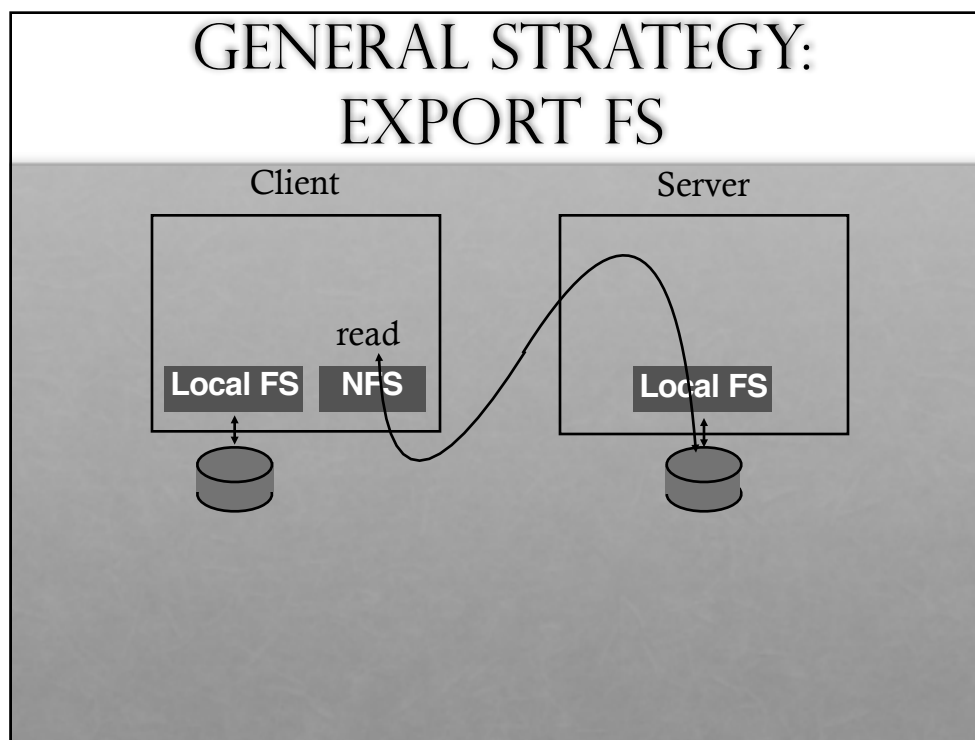
Caching

NFS ARCHITECTURE



RPC: Remote Procedure Call
Cache individual blocks of NFS files





OVERVIEW

Architecture

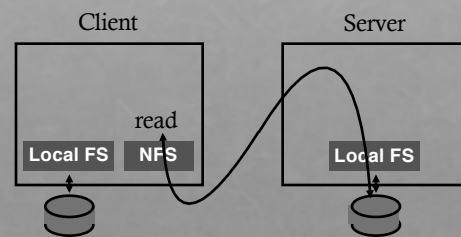
Network API

Caching

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



RPC

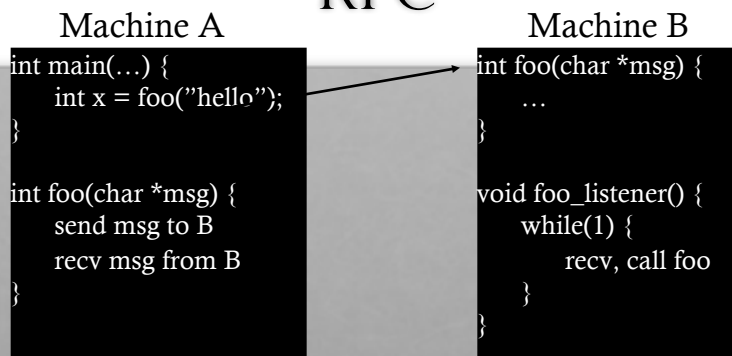
Remote Procedure Call

Motivation: What could be easier than calling a function?

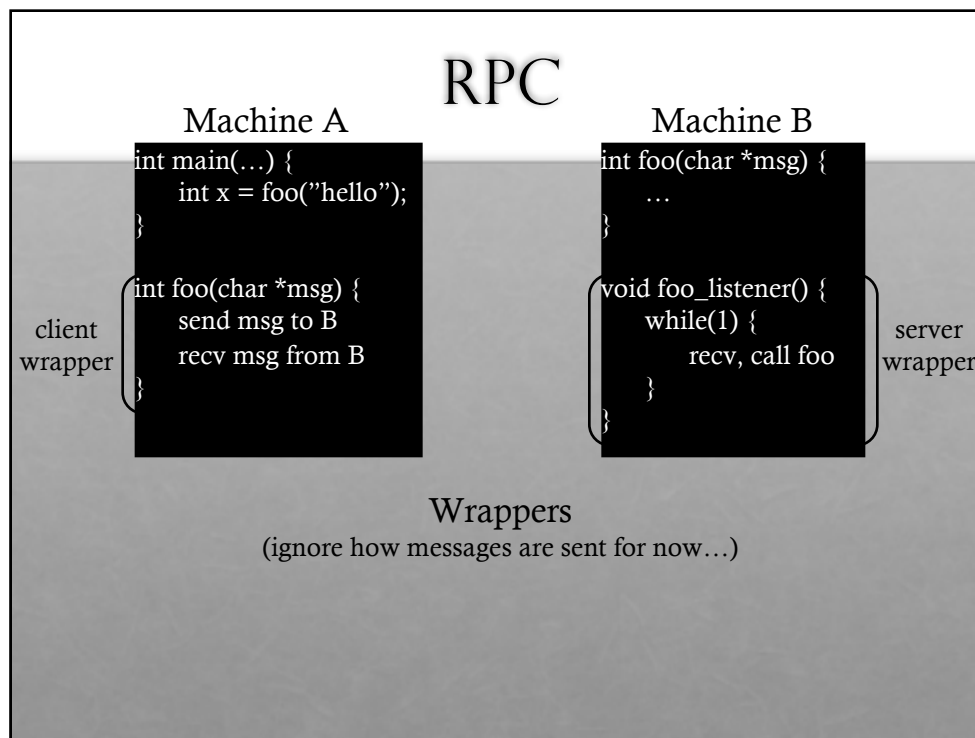
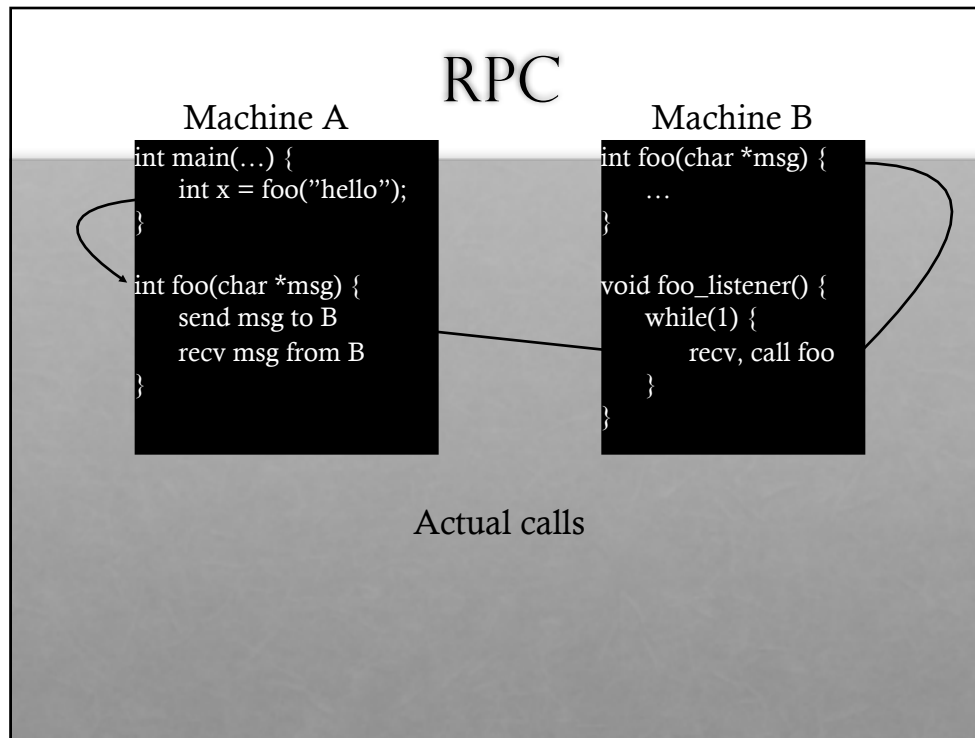
Strategy: create wrappers so calling function on remote machine appears like calling local function

Very common abstraction

RPC



How RPC appears to programmer



RPC TOOLS

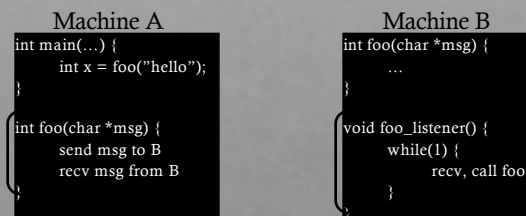
RPC packages help with two roles:

(1) Runtime library

- Thread pool
- Socket listeners call functions on server

(2) Stub/wrapper generation at compile time

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



WRAPPER GENERATION

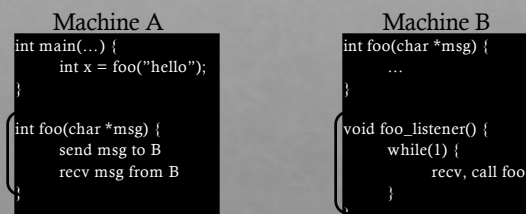
Wrappers must do conversions:

- 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
- serializing/deserializing



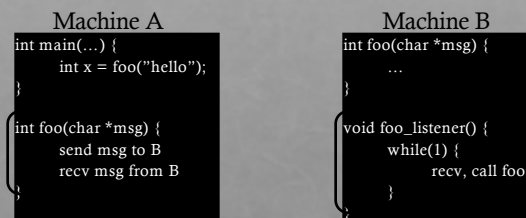
WRAPPER GENERATION: POINTERS

Why are pointers problematic?

Address passed from client not valid on server

Solutions?

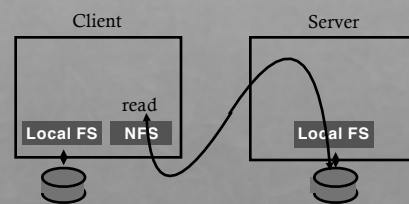
- Smart RPC package: follow pointers and copy data



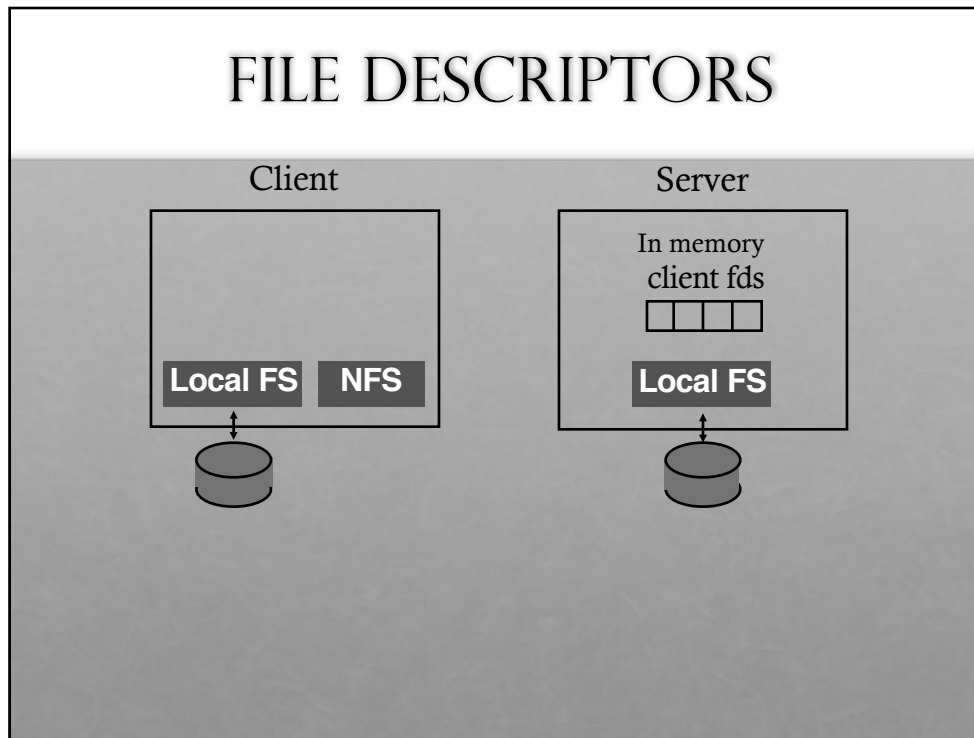
BACK TO NSF: STRATEGY 1

Attempt: Wrap regular UNIX system calls using RPC

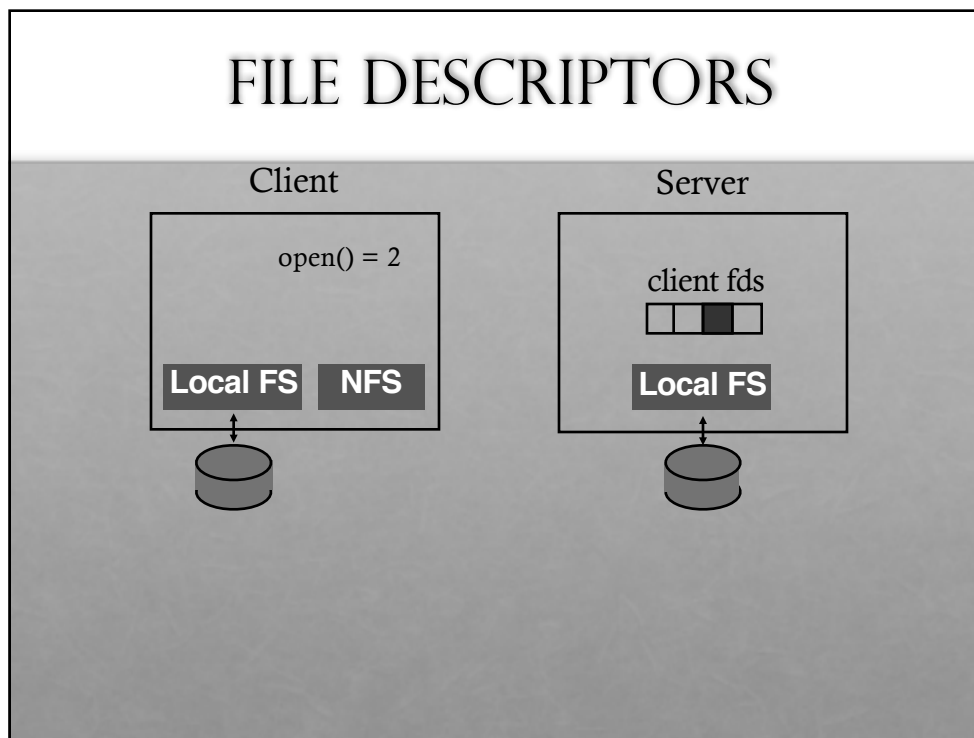
- `open()` on client calls `open()` on server
- `open()` on server returns fd back to client
- `read(fd)` on client calls `read(fd)` on server
- `read(fd)` on server returns data back to client



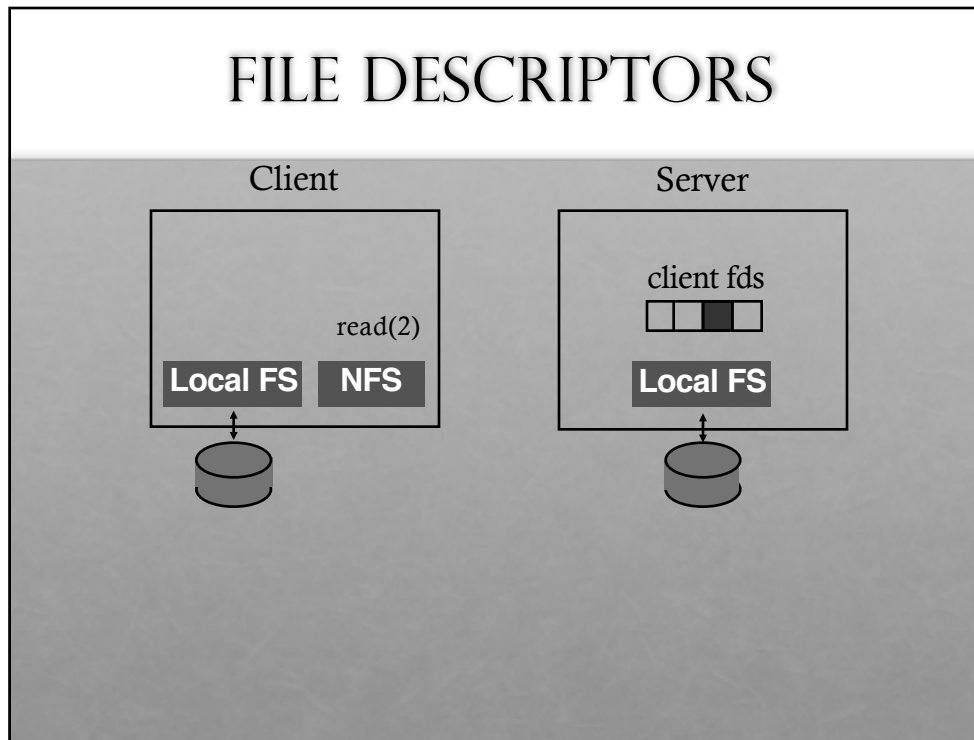
FILE DESCRIPTORS



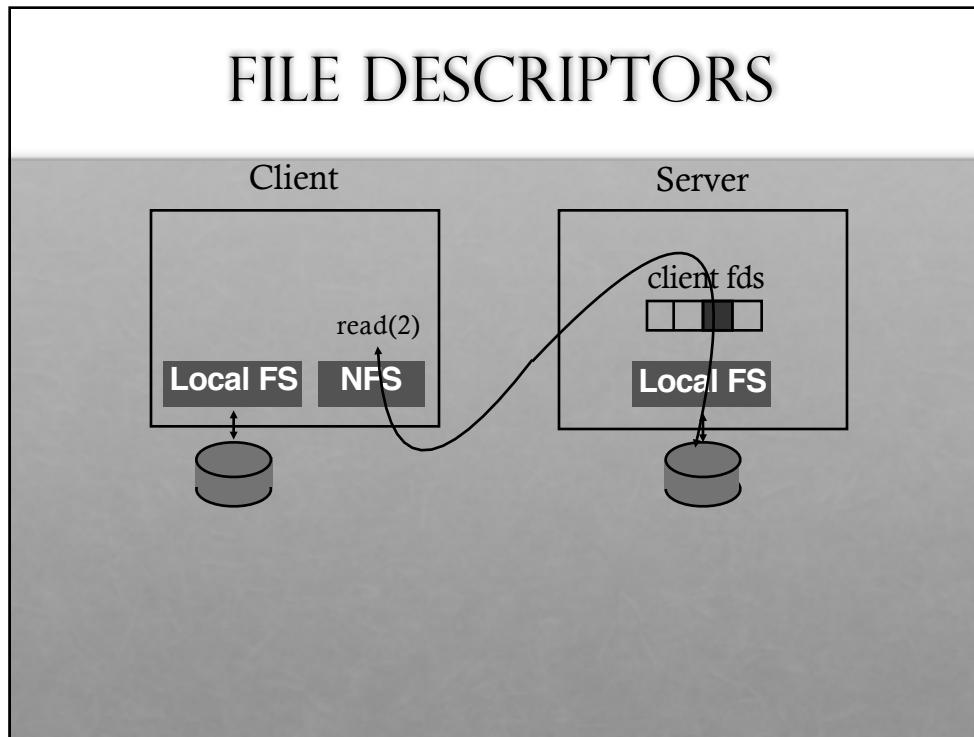
FILE DESCRIPTORS



FILE DESCRIPTORS



FILE DESCRIPTORS

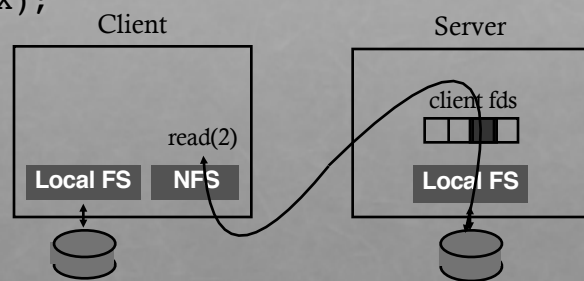


STRATEGY 1 PROBLEMS

What about server crashes? (and reboots)

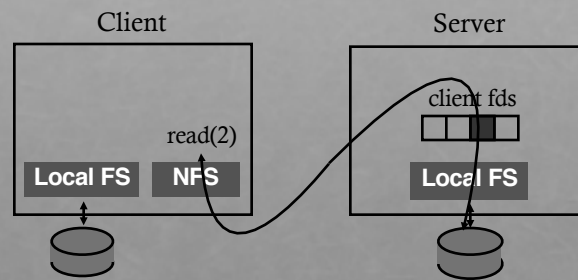
```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX); ← Server crash!
...
read(fd, buf, MAX);
```

Goal: behave like slow read



POTENTIAL SOLUTIONS

1. Run some crash recovery protocol upon reboot
 - Complex
2. Persist fds on server disk
 - Slow for disks
 - How long to keep fds? What if client crashes? misbehaves?

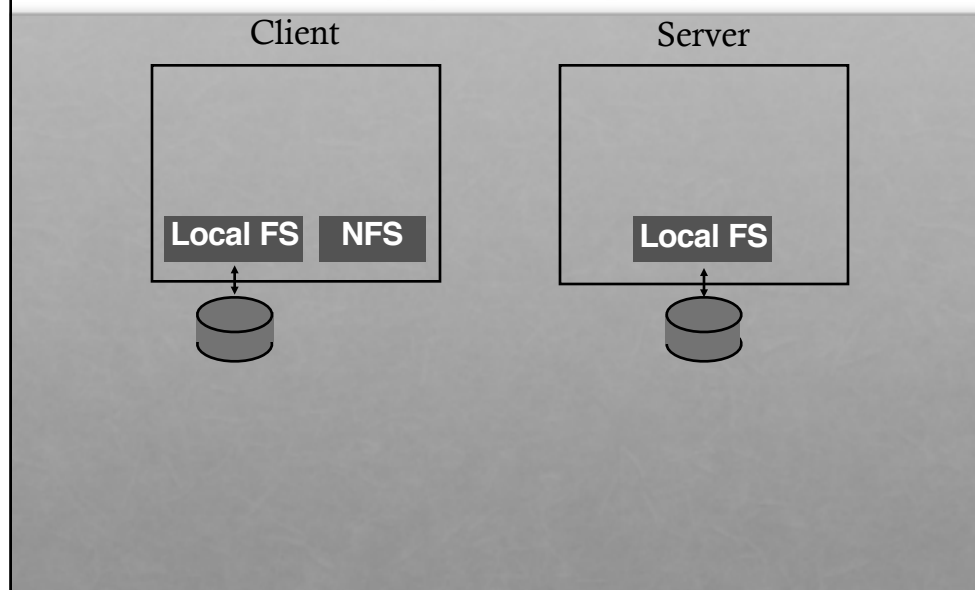


STRATEGY 2: PUT ALL INFO IN REQUESTS

Use “stateless” protocol!

- server maintains no state about clients
- server can still keep other state (cached copies)
- can crash and reboot with no correctness problems (just performance)

ELIMINATE FILE DESCRIPTORS



STRATEGY 2: PUT ALL INFO IN REQUESTS

Use “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 in each message

Server need not remember anything from clients

Pros? Server can crash and reboot transparently to clients

Cons? Too many path lookups

STRATEGY 3: INODE REQUESTS

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

This is pretty good! Any correctness problems?

If file is deleted, the inode could be reused

- Inode not guaranteed to be unique over time

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)

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()?

If RPC library resends messages, what happens when append() is retried?

Background: Why would RPC call same procedure multiple times?

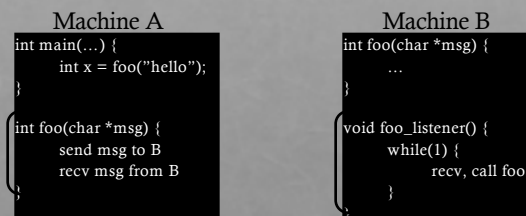
COMMUNICATION OVERVIEW

How are RPCs built on top of messages?

- How can RPC ensure remote procedure is called exactly once??

Raw messages: UDP

Reliable messages: TCP



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

RAW MESSAGES: UDP

Advantages

- Lightweight
- Some applications make better reliability decisions themselves (e.g., video conferencing programs)

Disadvantages

- More difficult to write applications correctly

RELIABLE MESSAGES: LAYERING STRATEGY

TCP: Transmission Control Protocol

Using software, build reliable, logical connections over unreliable connections

- **Make sure each message is received**
- Make sure messages are received in order
- Make sure no duplicates are received

Techniques:

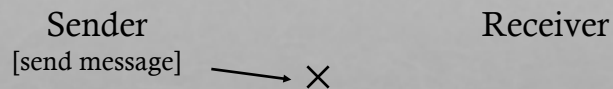
- acknowledgment (ACK)

TECHNIQUE #1: ACK



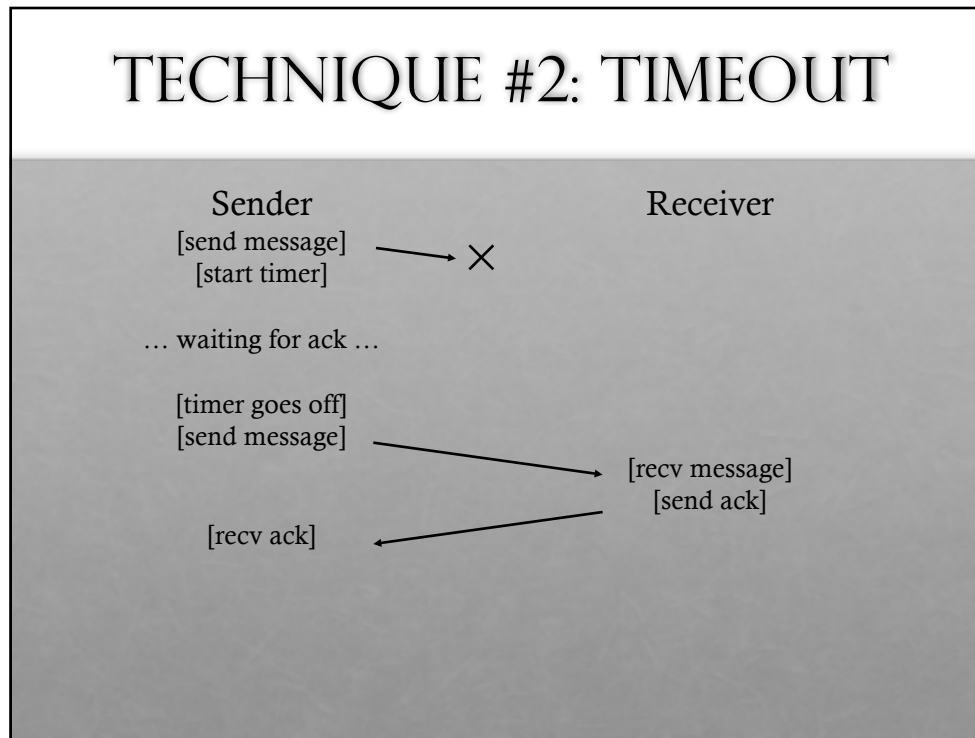
Sender knows message was received

ACK



Sender doesn't receive ACK...
What to do?

TECHNIQUE #2: TIMEOUT



LOST ACK: ISSUE 1

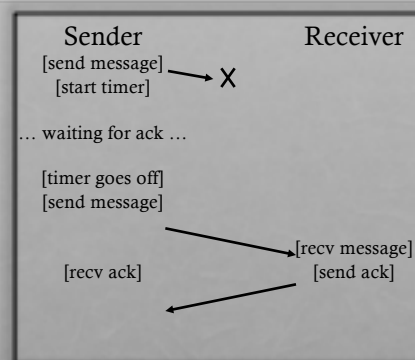
How long to wait?

Too long?

- System feels unresponsive

Too short?

- Messages needlessly re-sent (duplicates!!)
- Messages may have been dropped due to overloaded server. Resending makes overload worse!



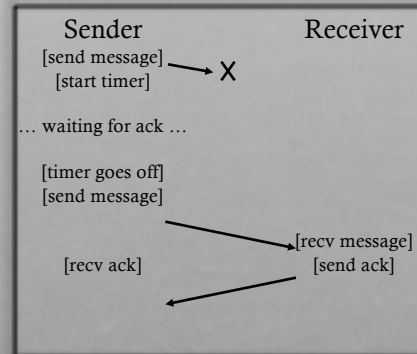
LOST ACK: ISSUE 1

How long to wait?

One strategy: be adaptive

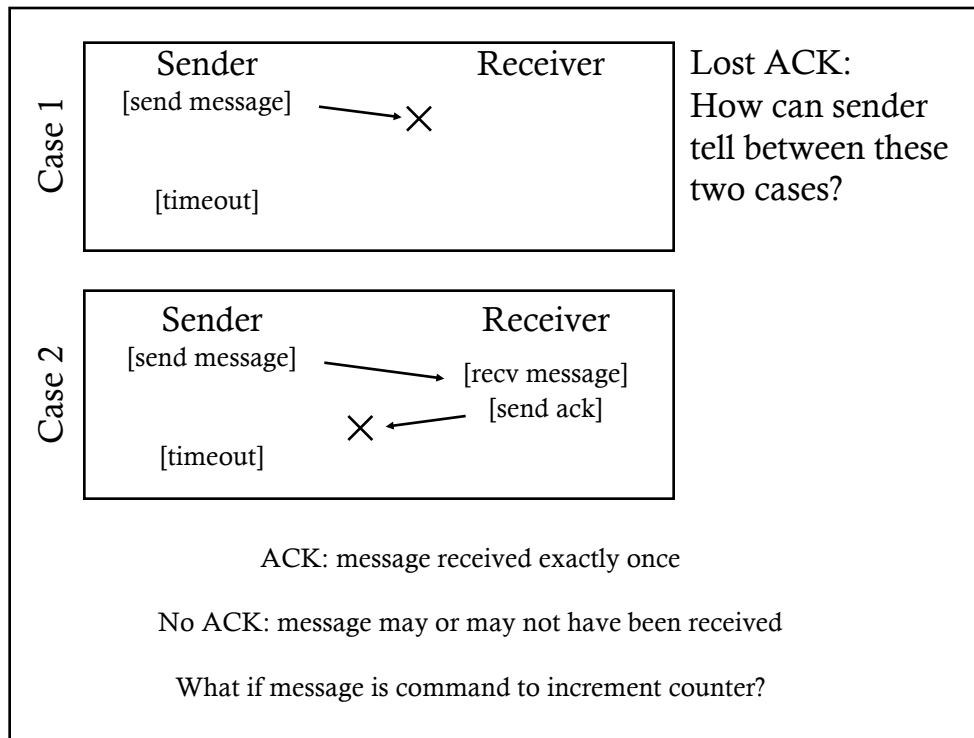
Adjust time based on how long acks usually take

For each missing ack, wait longer between retries

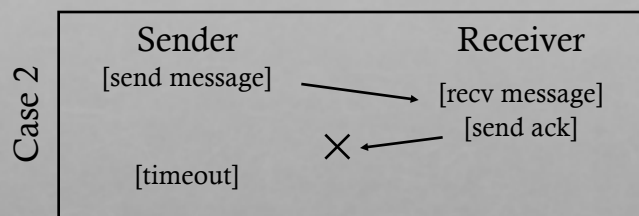


LOST ACK: ISSUE 2

What does not receiving an ACK really mean?



PROPOSED SOLUTION

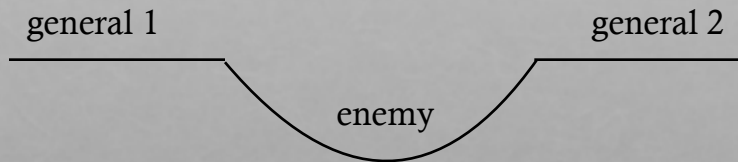


Proposal:

Sender could send an AckAck so receiver knows whether to retry sending an Ack

Sound good?

ASIDE: TWO GENERALS' PROBLEM



Suppose generals agree after N messages

Did the arrival of the N'th message change decision?

- if yes: then what if the N'th message had been lost?
- if no: then why bother sending N messages?

RELIABLE MESSAGES: LAYERING STRATEGY

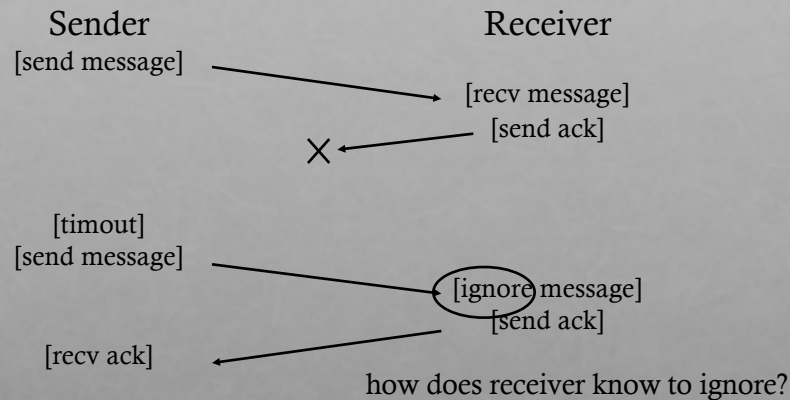
Using software, build reliable, logical connections over unreliable connections

- ~~Make sure each message is received~~
- **Make sure messages are received in order**
- **Make sure no duplicates are received**

Techniques:

- acknowledgment
- timeout
- **remember received messages**

TECHNIQUE #3: RECEIVER REMEMBERS MESSAGES



SOLUTIONS

Solution 1: remember every message ever received

Solution 2: sequence numbers

- senders gives each message an increasing unique seq number
- receiver tracks N
 - knows it has seen all messages before N
 - receiver remembers seq number of messages received after N

Suppose message K is received. TCP suppresses message if:

- $K < N$
- Msg K is already buffered (potentially adjust N)

Sequence numbers also enables TCP to sort msgs to be received in order

TCP

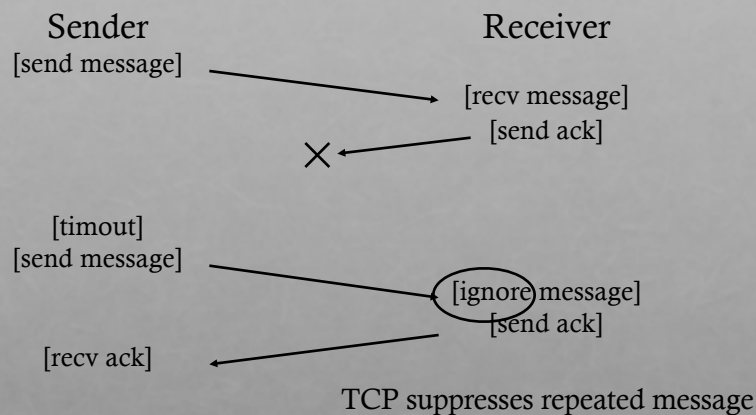
TCP: Transmission Control Protocol – Very Popular

Based on sequence numbers

Buffers and sorts messages so arrive in order

Timeouts are adaptive

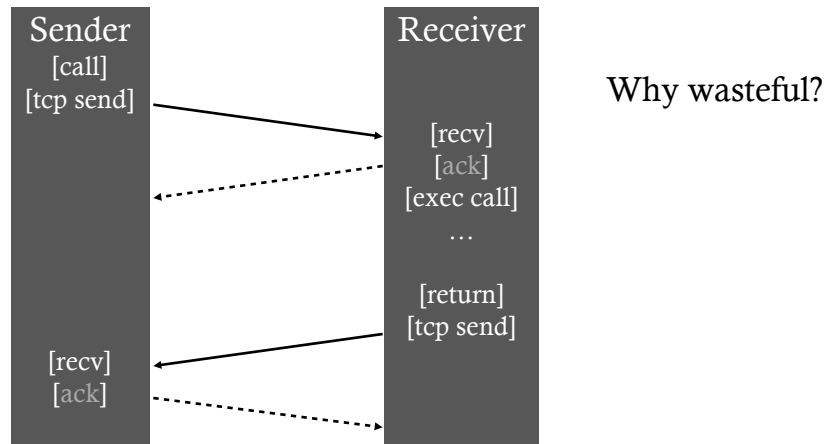
RPC OVER TCP?



Problem: TCP tracks sequence numbers → stateful

If server crashes, forgets which RPC's have been executed! Might replay!

RPC OVER TCP?



RPC OVER UDP

Use function return as implicit ACK

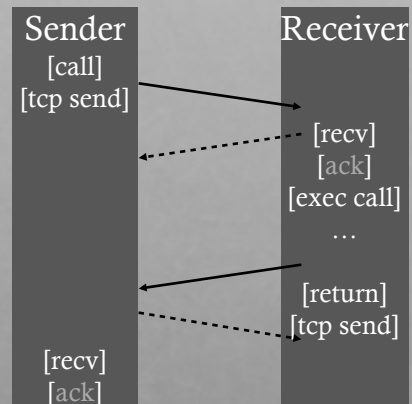
- Piggybacking technique

What if function takes long time?

- Receiver sends separate ACK

How can receiver suppress duplicates?

- Must do similar work as TCP...
- Requires remembering state...



If server crashes, forgets which RPC's have been executed! Might replay!

SO: 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()?

If RPC library retries, what happens when append() is retried?

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

IDEMPOTENT OPERATIONS

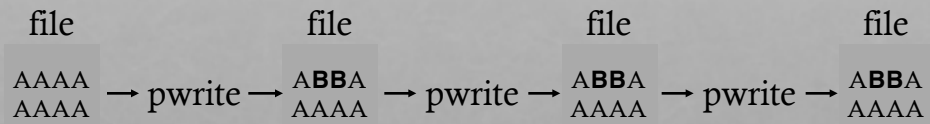
Solution:

Design API so no harm if execute function more than once

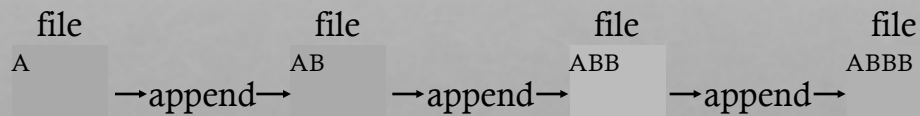
If $f()$ is **idempotent**, then:

$f()$ has the same effect as $f(); f(); \dots f(); f()$

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

STRATEGY 4: FILE HANDLES

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

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

STRATEGY 5: CLIENT LOGIC

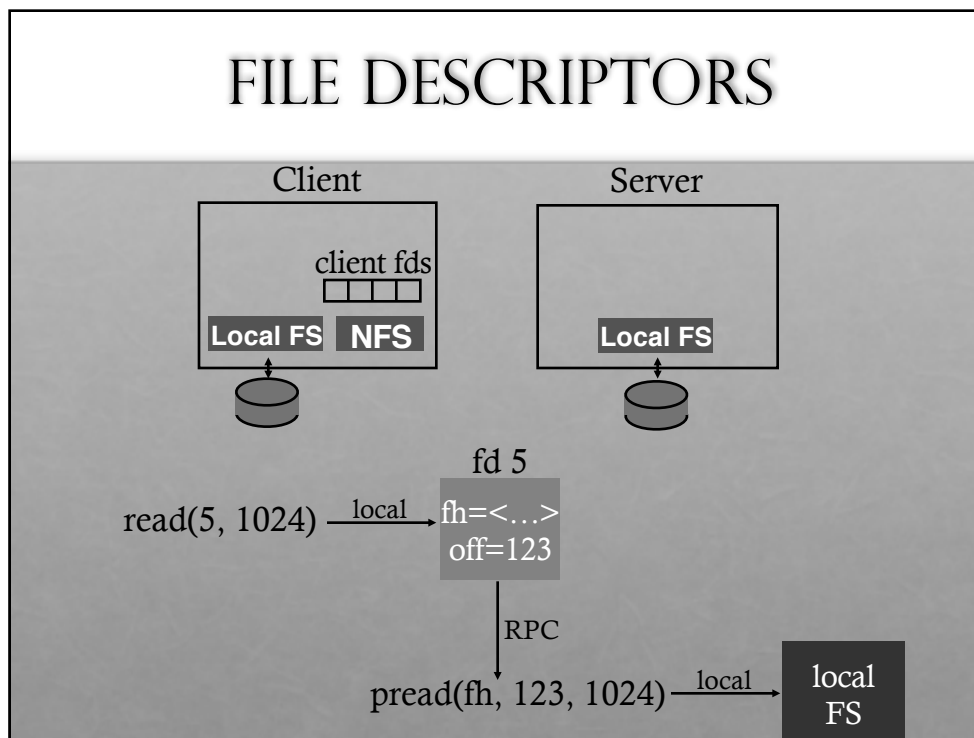
Build normal UNIX API on client side on top of idempotent, RPC-based API

Client `open()` creates a local fd object

Local fd object contains:

- file handle
- current offset

FILE DESCRIPTORS



OVERVIEW

~~Architecture~~

~~Network API~~

Cache

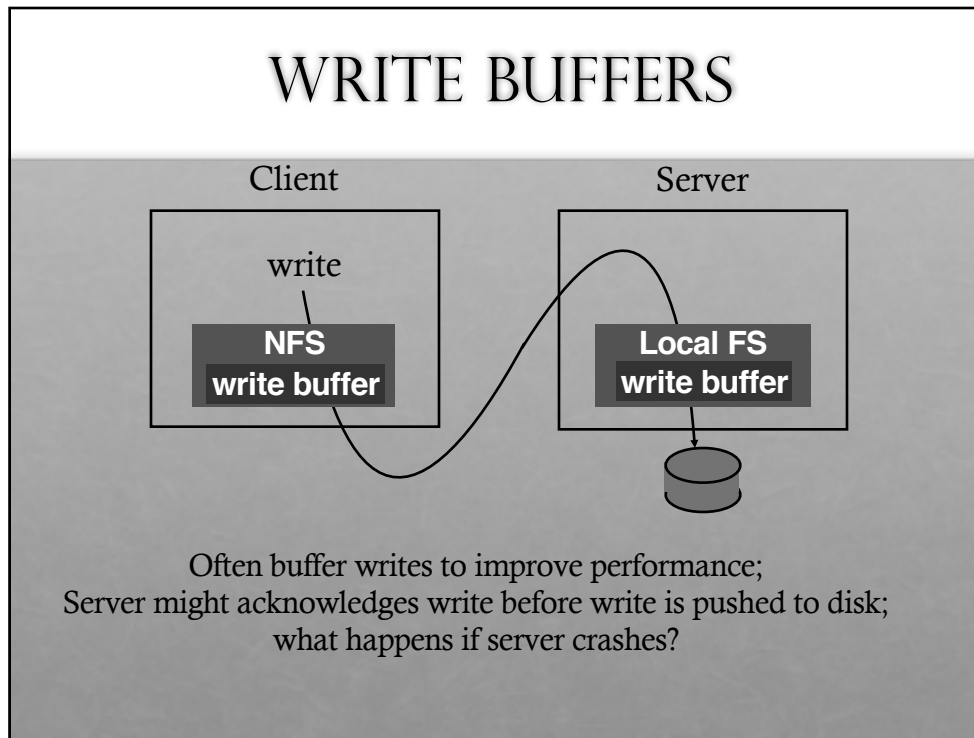
CACHE CONSISTENCY

NFS can cache data in three places:

- server memory
- client disk
- client memory

How to make sure all versions are in sync?

WRITE BUFFERS



SERVER WRITE BUFFER LOST

client:

write A to 0

write B to 1

write C to 2

server mem:



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

server mem:

A	B	C
---	---	---

server disk:

A	B	C
---	---	---

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

server mem:

X	B	C
---	---	---

server disk:

A	B	C
---	---	---

write X to 0

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

server mem:

X	B	C
---	---	---

server disk:

X	B	C
---	---	---

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

server mem:

X	Y	C
---	---	---

server disk:

X	B	C
---	---	---

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

server mem:



server disk:



write X to 0

write Y to 1

crash!

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

server mem:



server disk:



write X to 0

write Y to 1

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:



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:



server disk:

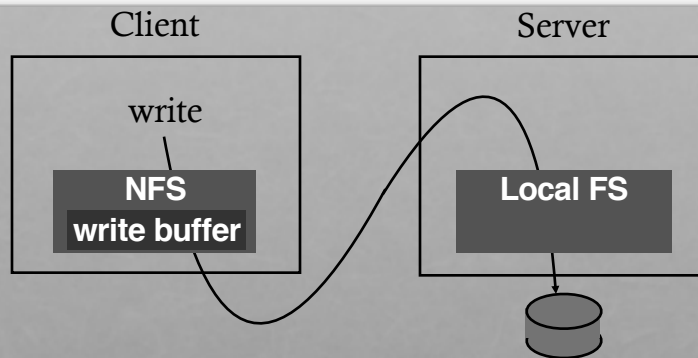


Problem:

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

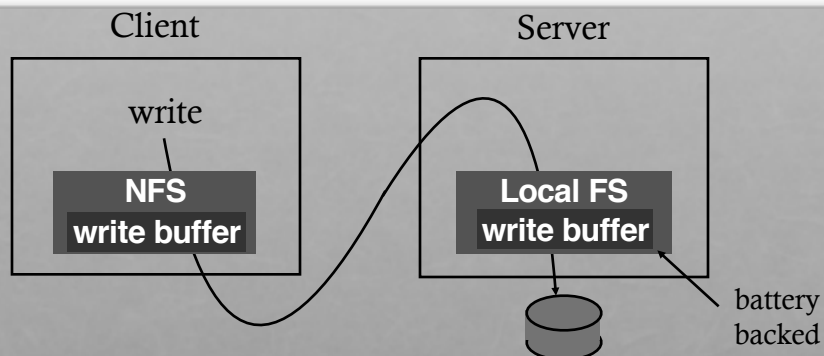
Solutions????

WRITE BUFFERS

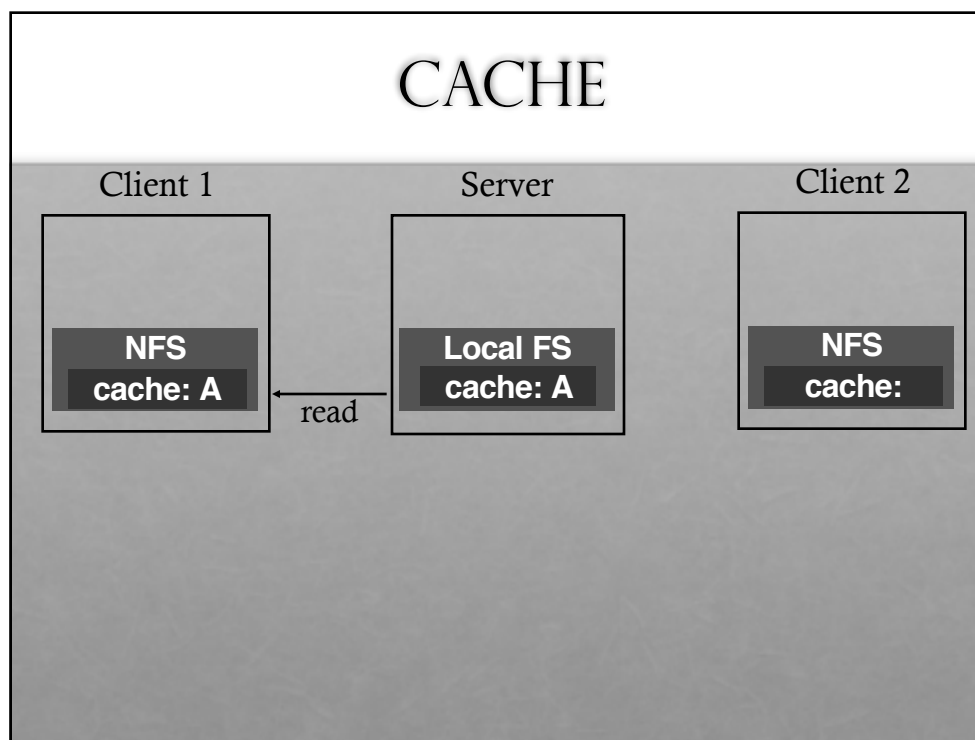
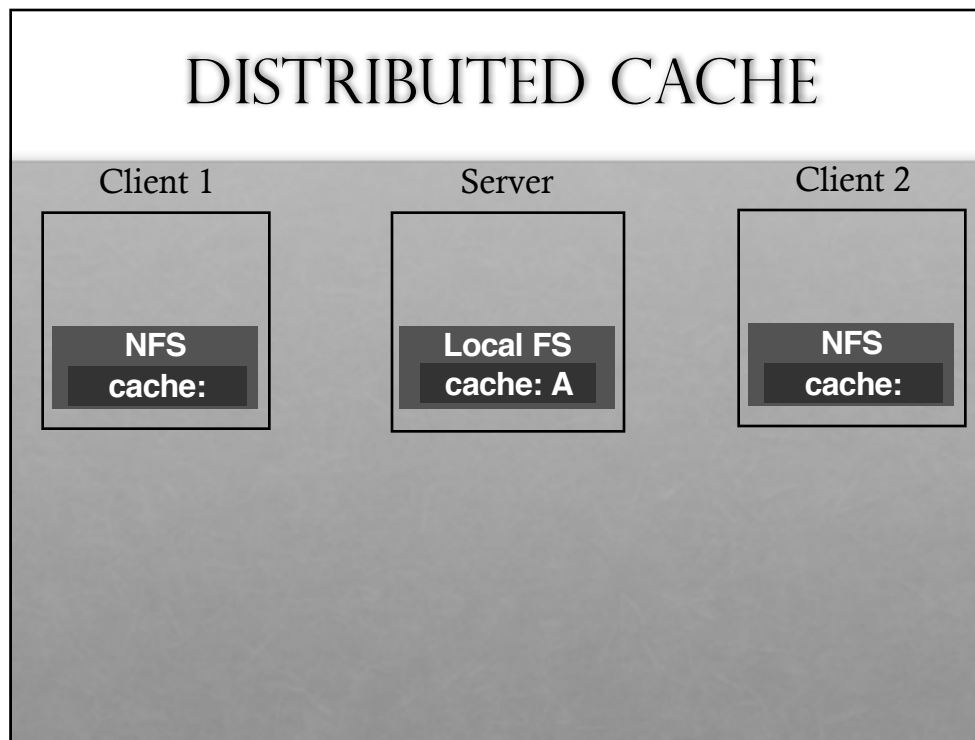


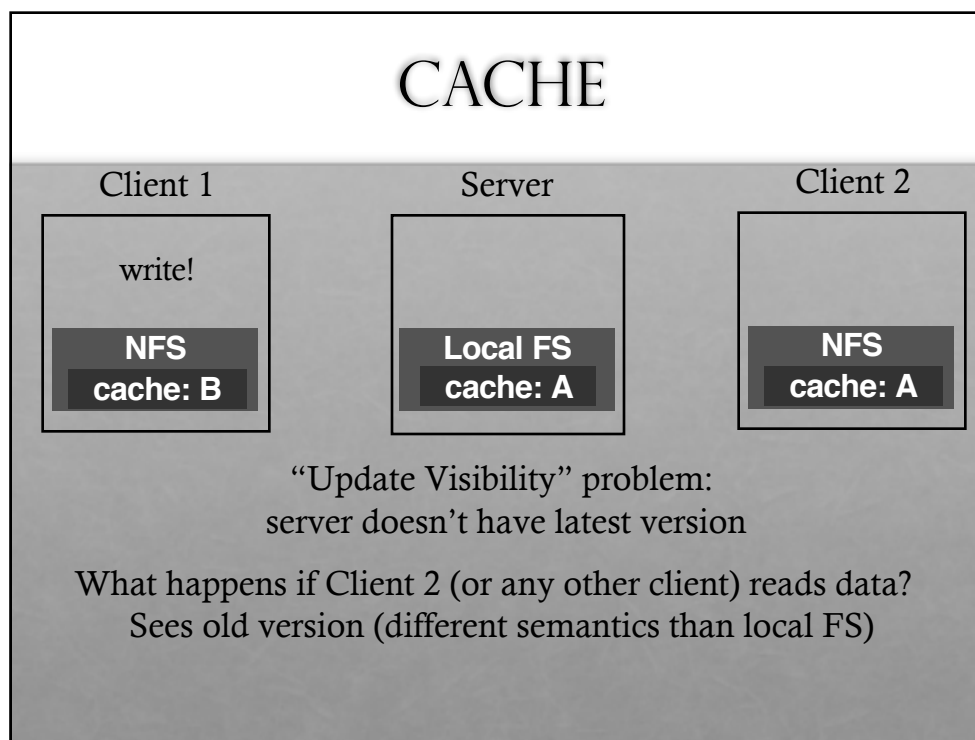
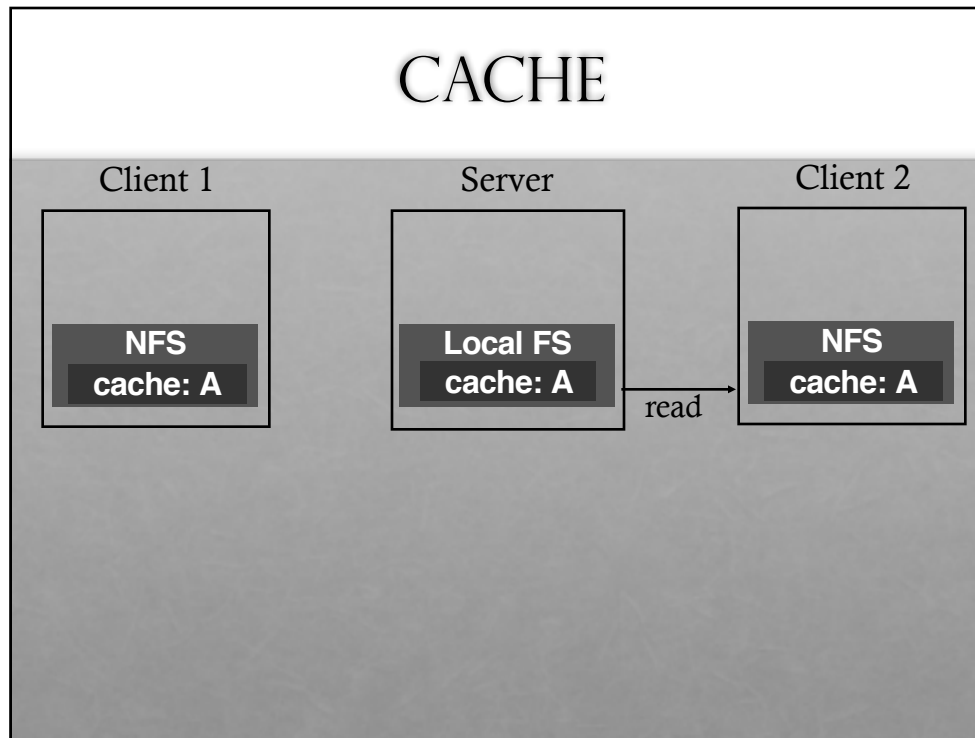
1. Don't use server write buffer
(persist data to disk before acknowledging write)
Problem: Slow!

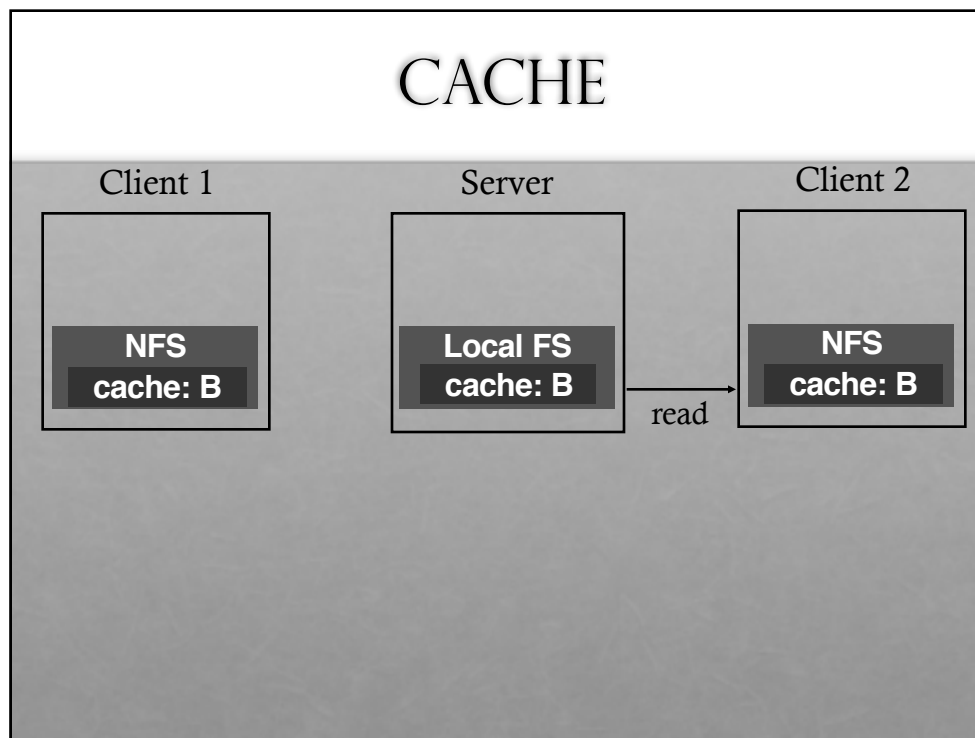
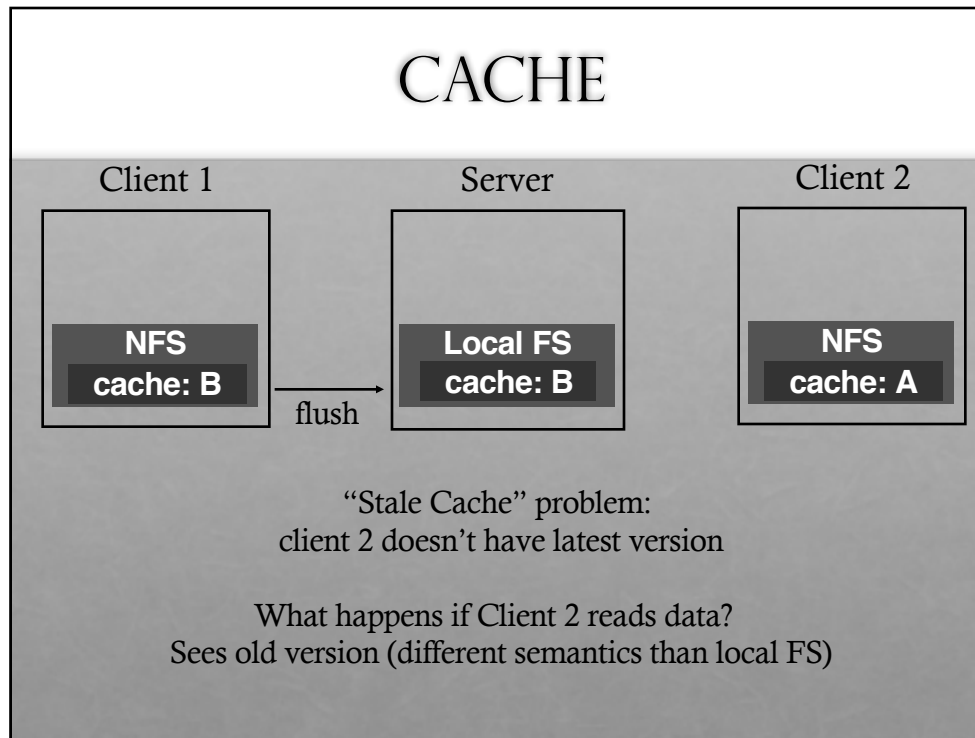
WRITE BUFFERS



2. use persistent write buffer (more expensive)



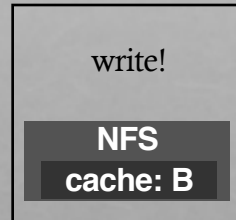




PROBLEM 1: UPDATE VISIBILITY

Client 1

Server



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

- Client flushes cache entry to server

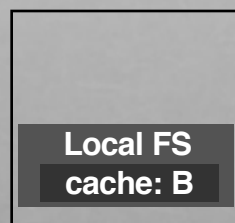
When should client perform flush????? (3 reasonable options??)

NFS solution: flush on fd close

PROBLEM 2: STALE CACHE

Server

Client 2



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

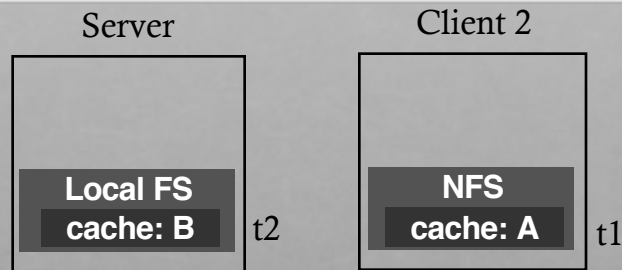
One possible solution:

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

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 ($t2 > t1$)

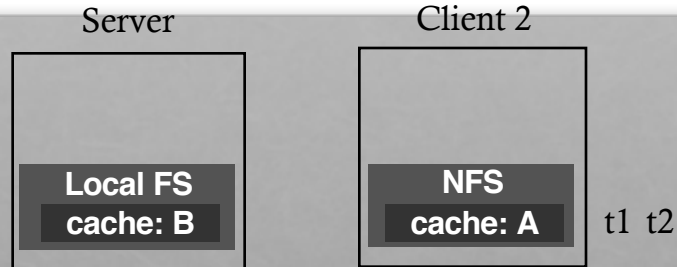
MEASURE THEN BUILD

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

What is the result? Never see updates on server!

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

What is the result? Could read data that is up to 3 seconds old

NFS SUMMARY

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

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

Caching and write buffering is harder 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

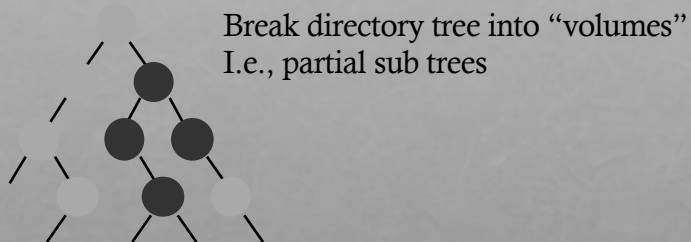
Primary goal: scalability! (many clients per server)

More reasonable semantics for concurrent file access

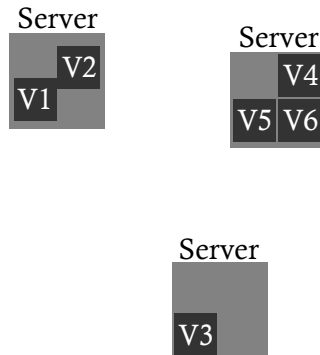
AFS DESIGN

NFS: Server exports local FS

AFS: Directory tree stored across many server machines
(helps scalability!)

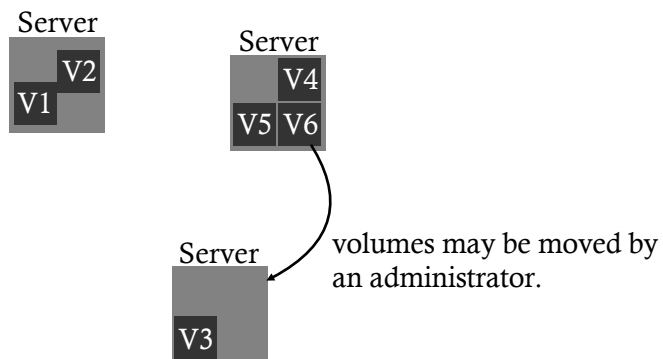


VOLUME ARCHITECTURE

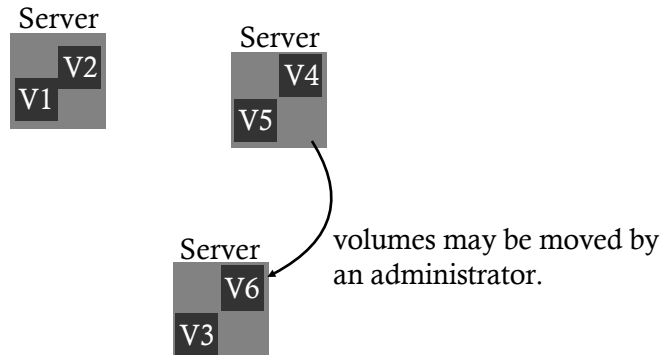


collection of servers store different volumes that together form directory tree

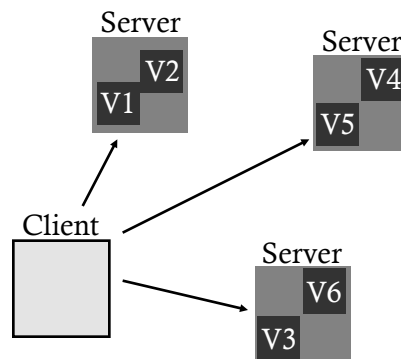
VOLUME ARCHITECTURE



VOLUME ARCHITECTURE



VOLUME ARCHITECTURE



Client library gives seamless view of directory tree by automatically finding volumes

Communication via RPC
Servers store data in local file systems

AFS CACHE CONSISTENCY

Update visibility

Stale cache

UPDATE VISIBILITY

Client 1

Server

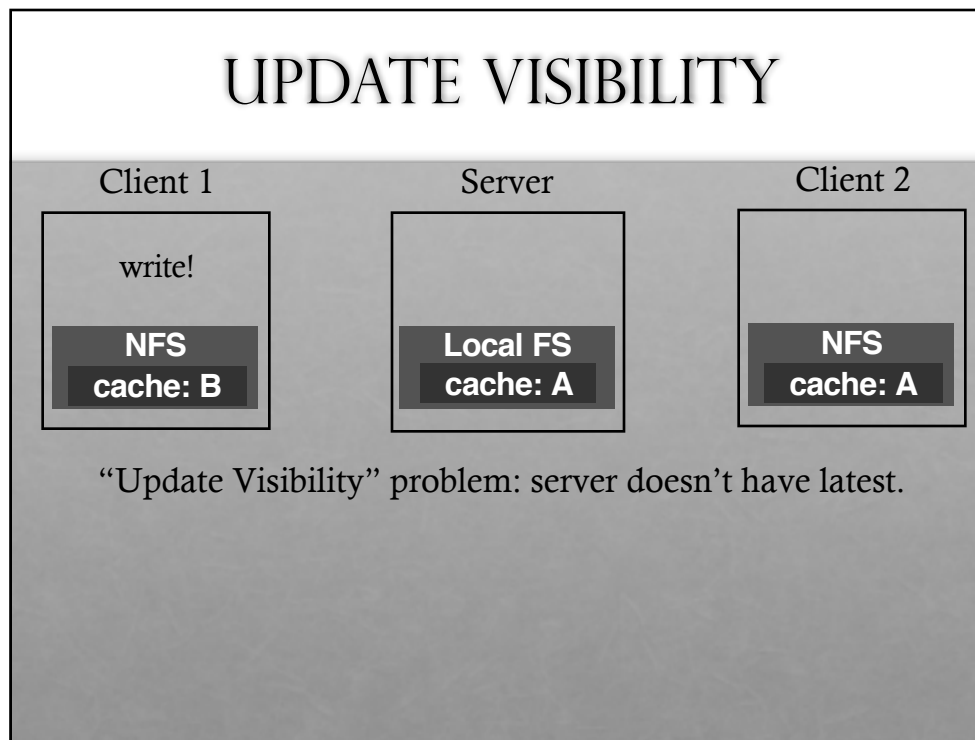
Client 2

NFS
cache: A

Local FS
cache: A

NFS
cache: A

UPDATE VISIBILITY



UPDATE VISIBILITY

NFS solution is to flush blocks

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

Problems

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

UPDATE VISIBILITY

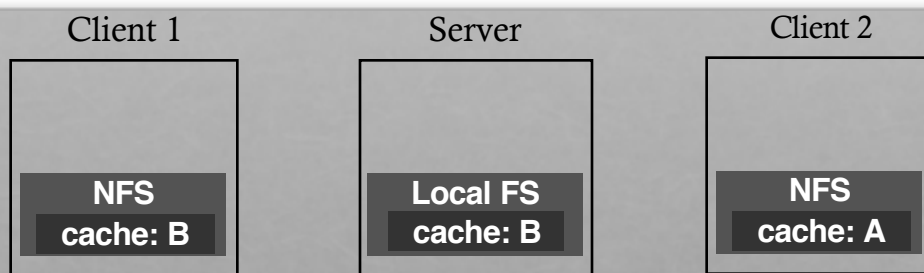
AFS solution:

- 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 mixed data on server

CACHE CONSISTENCY



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

STALE CACHE

NFS rechecks cache entries compared to server before using them, assuming check hasn't been done "recently"

How to determine how recent? (about 3 seconds)

"Recent" is too long?	client reads old data
"Recent" is too short?	server overloaded with stats

STALE CACHE

Server

Client 2

Local FS
cache: B

NFS
cache: A

AFS solution: Tell clients when data is overwritten

- Server must remember which clients have this file open right now

When clients cache data, ask for "callback" from server if changes

- Clients can use data without checking all the time

Server no longer stateless!

CALLBACKS: DEALING WITH STATE

What if client crashes?

What if server runs out of memory?

What if server crashes?

CLIENT CRASH

Server

Local FS
cache: B

Client 2

NFS
cache: A

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

LOW SERVER MEMORY

Server

Client 2

Local FS
cache: B

NFS
cache: A

Strategy: tell clients you are dropping their callback

What should client do?

Option 1: Discard entry from cache

Option 2: ??? Mark entry for recheck

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...

PREFETCHING

AFS paper notes: “the study by Ousterhout *et al.* has shown that most files in a 4.2BSD environment are read in their entirety.”

What are the implications for client prefetching policy?

Aggressively prefetch whole files.

WHOLE-FILE CACHING

Upon open, AFS client fetches whole file (even if huge), storing in local memory or disk

Upon close, client flushes file to server (if file was written)

Convenient and intuitive semantics:

- AFS needs to do work only for open/close
 - Only check callback on open, not every read
- reads/writes are local
 - Use same version of file entire time between open and close

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)

AFS VS NFS PROTOCOLS

Can you summarize the consistency semantics provided by NFS?

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
20	read(fd, block2);		read
30	read(fd, block1);		get attr
31	read(fd, block2);		
40		fd = open("file A");	lookup
50		write(fd, block1);	
60	read(fd, block1);		get attr
70		close(fd);	write bl to disk
80	read(fd, block1);		read()
81	read(fd, block2);		read()
90	close(fd);		
100	fd = open("fileA");		lookup
110	read(fd, block1);		get attr
120	close(fd);		

AFS PROTOCOL

Time	Client A	Client B	Server Action?
0	fd = open("file A");		setup callback for file A
10	read(fd, block1);		send all of file A
20	read(fd, block2);		
30	read(fd, block1);		
31	read(fd, block2);		
40		fd = open("file A");	setup callback
50		write(fd, block1);	send all of A
60	read(fd, block1);		
70		close(fd);	send write changes of A break all callbacks
80	read(fd, block1);		
81	read(fd, block2);		
90	close(fd);		
100	fd = open("fileA");		no callback! need to fetch A again
110	read(fd, block1);		
120	close(fd);		send A