Distributed Systems and NFS

CMPU 334 – Operating Systems

Jason Waterman

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
- More storage capacity
- Fault tolerance
- Data sharing

New Challenges

- System failure: need to worry about partial failure

- Communication failure: links unreliable
  - bit errors
  - packet loss
  - node/link failure

- Motivation example:
  - Why are network sockets less reliable than pipes?

Communication Overview

- Raw messages: UDP
- Reliable messages: TCP
- Remote procedure call: RPC

Raw Messages: UDP

- UDP : User Datagram Protocol
  - Reads and writes over socket file descriptors
  - Messages sent from/to ports to target a process on machine
- Provides 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

Raw Messages: UDP

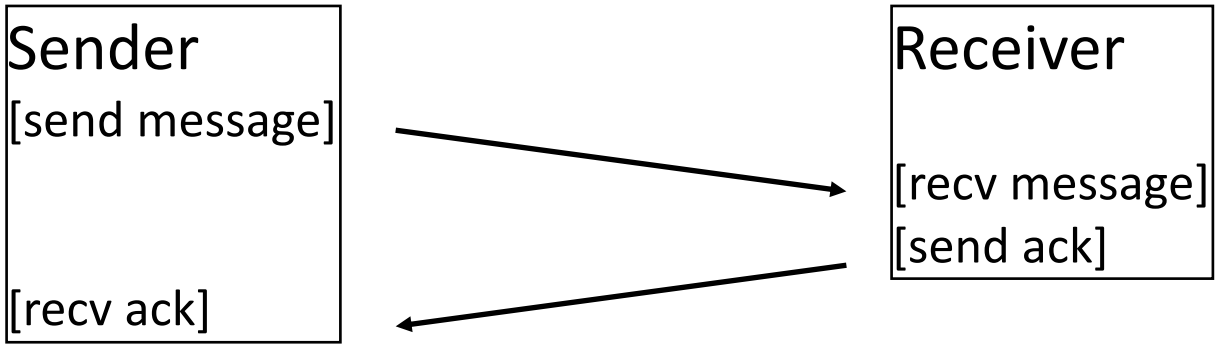
- Advantages
  - Lightweight
  - Some applications make better reliability decisions themselves (e.g., video conferencing programs)

- Disadvantages
  - More difficult to write applications correctly

Reliable Messages: Layering strategy

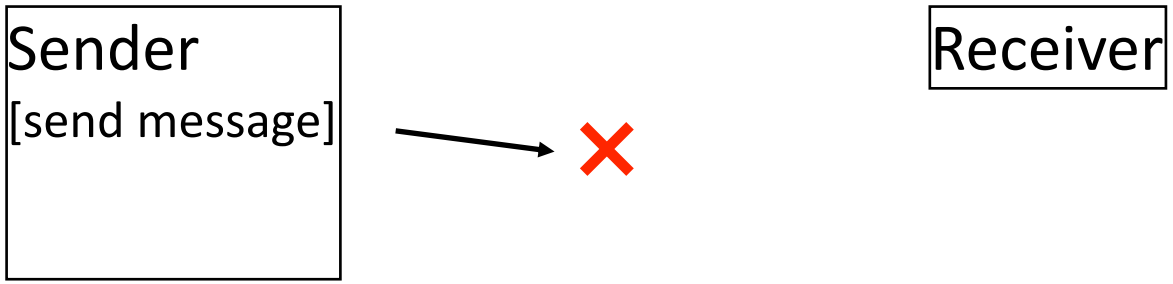
- TCP: Transmission Control Protocol
- Using software, build reliable, logical connections over unreliable connections
- Techniques:
- Acknowledgment (ACK)

Technique #1: ACK



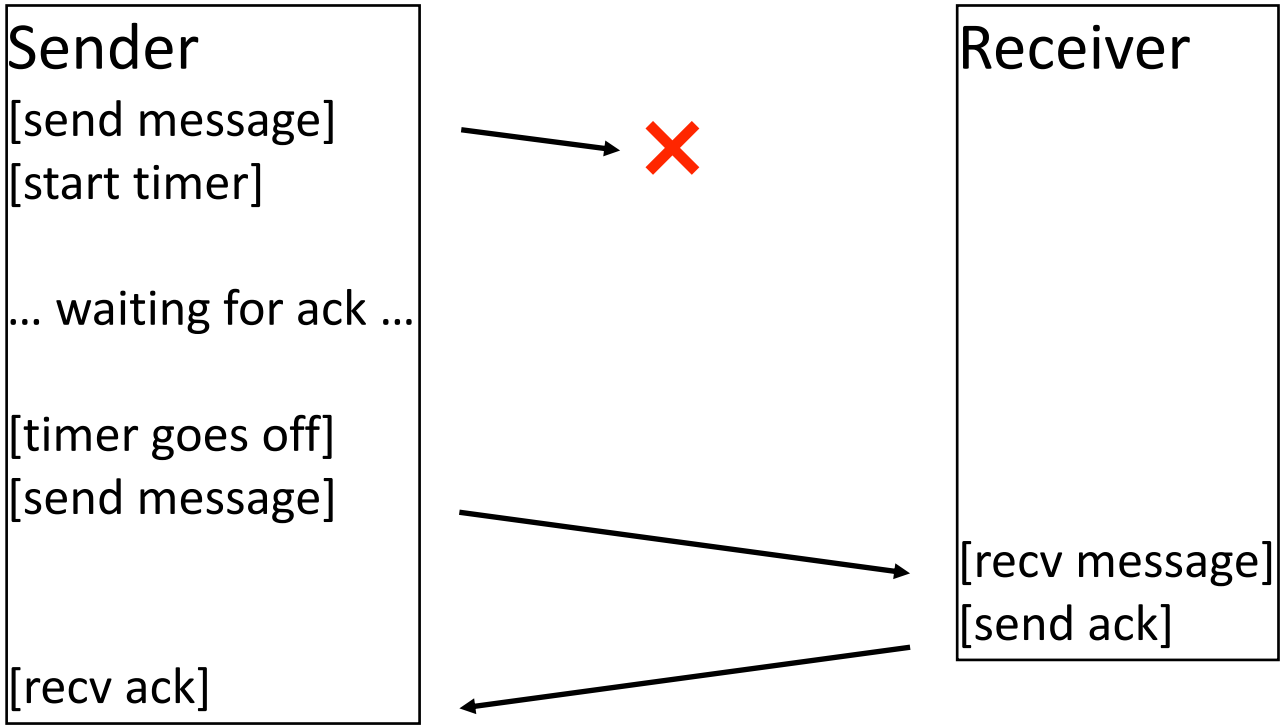
Sender knows message was received

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

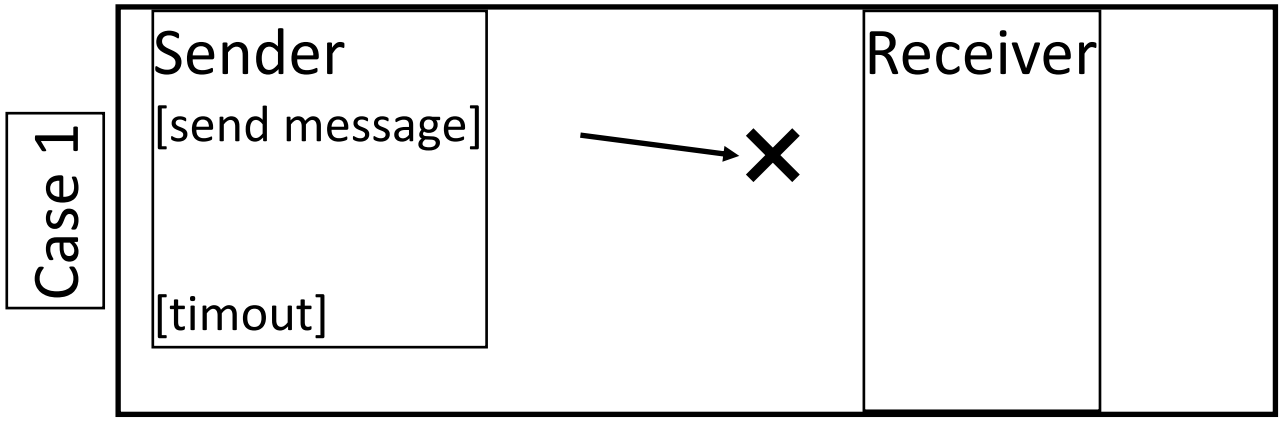
- Too short?
  - Messages needlessly re-sent
  - Messages may have been dropped due to overloaded server. Resending makes overload worse!

Lost ACK: Issue 1

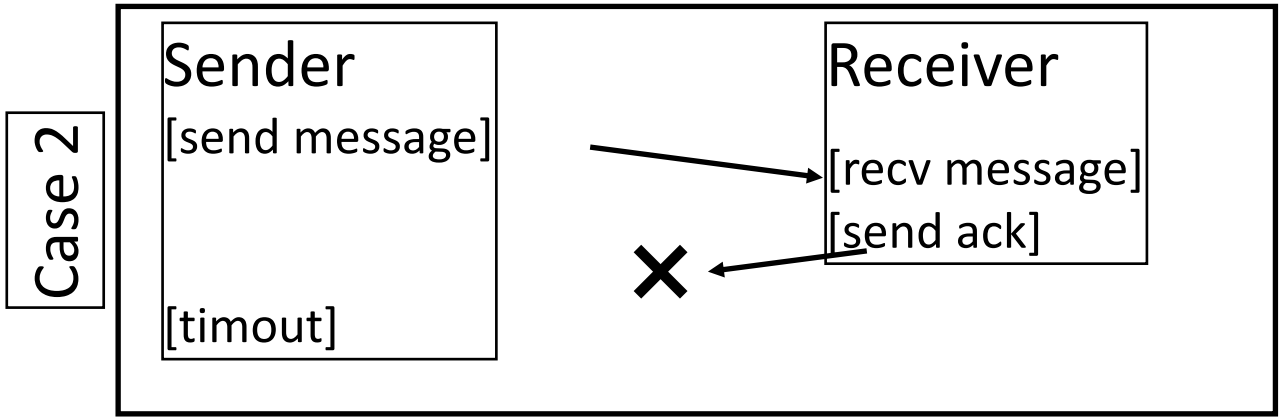
- How long to wait?
- One strategy: be adaptive
- Adjust time based on how long acks usually take
- For each missing ack, wait longer between retries

Lost ACK: Issue 2

- What does a lost ack really mean?



Lost ACK:
How can sender tell between these two cases?

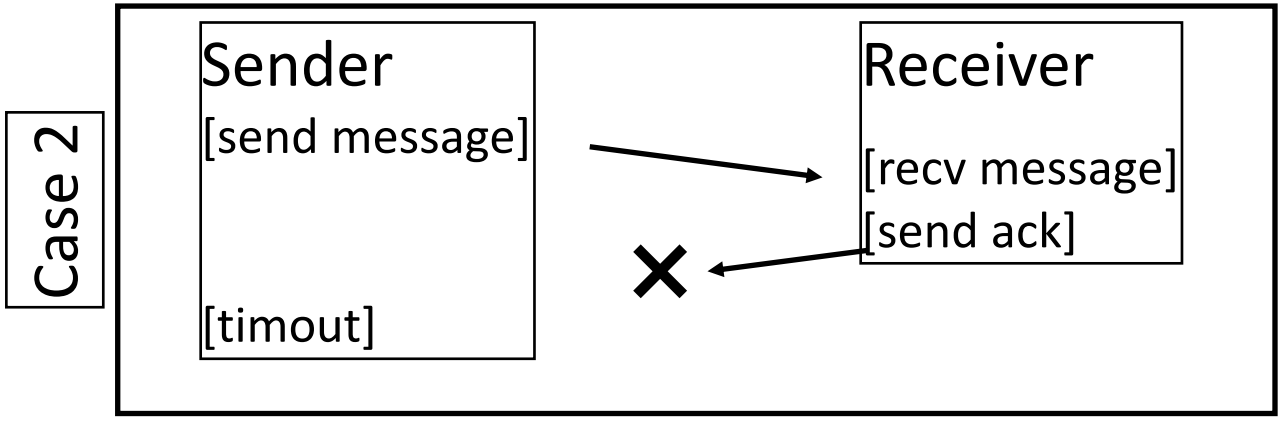


ACK: message received exactly once

No ACK: message may or may not have been received

What if message is command to increment counter?

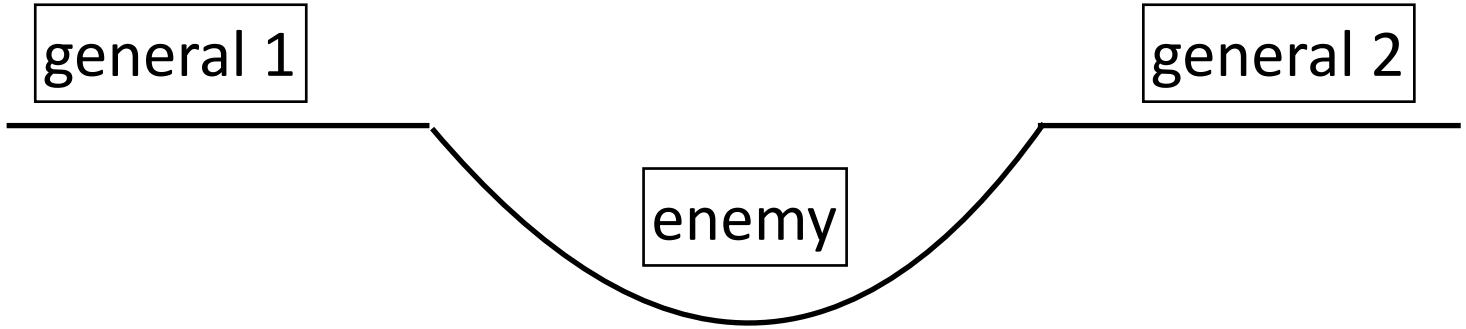
Proposed Solution



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

- Sound good?

Aside: Two Generals' Problem



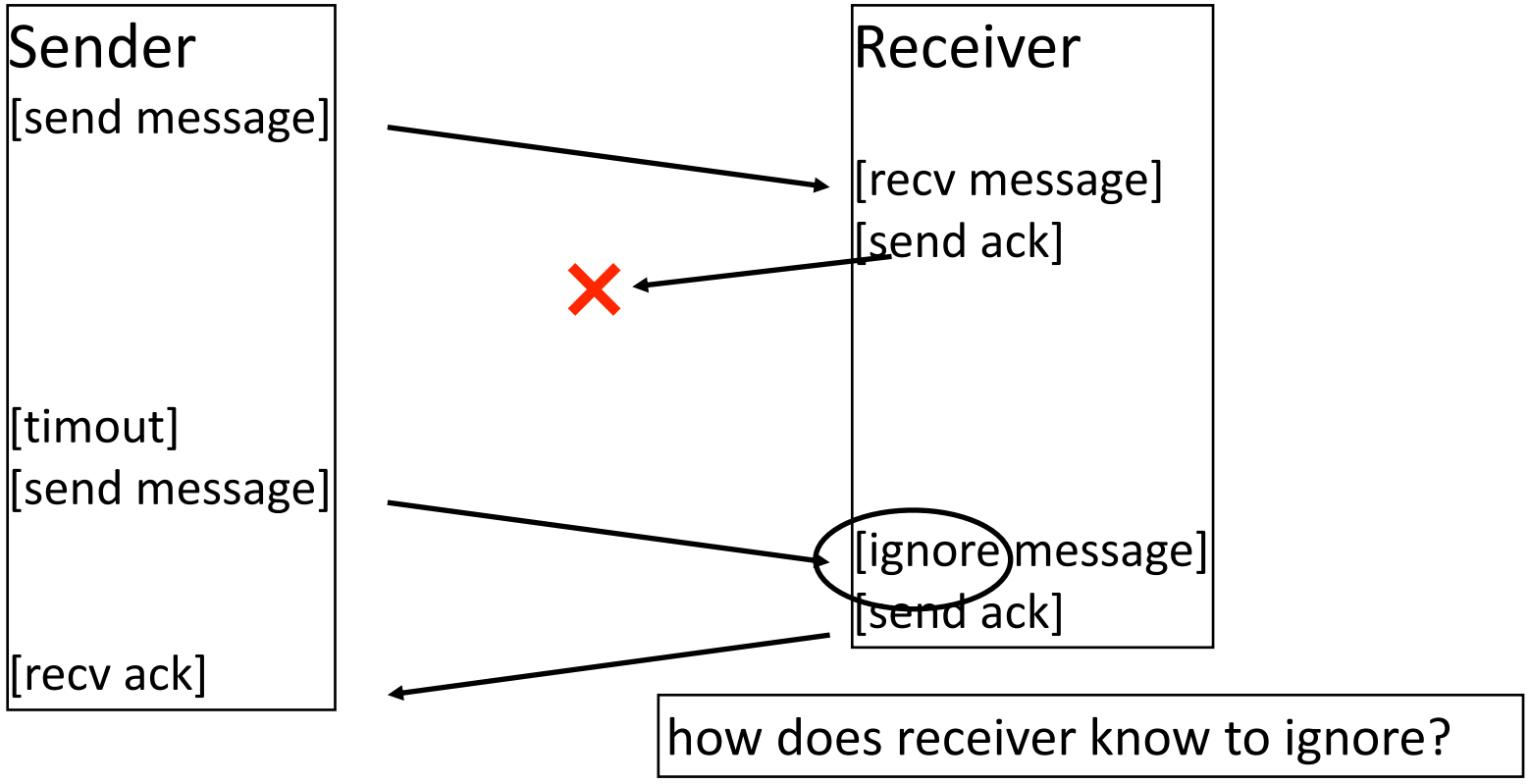
- Suppose generals agree after N messages
- Did the arrival of the N'th message change decision?
  - If yes: then what if the N'th message had been lost?
  - If no: then why bother sending N messages?

Reliable Messages: Layering Strategy

- Using software, build reliable, logical connections over unreliable connections

- Techniques:
  - Acknowledgment
  - Timeout
  - Remember sent 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 seqence number
  - Receiver knows it has seen all messages before N
  - Receiver remembers messages received after N

- Suppose message K is received. Suppress message if:
  - - K < N
  - - Message K is already buffered

TCP

- TCP: Transmission Control Protocol
- Most popular protocol based on seqence nums
- Buffers messages so they arrive in order
- Timeouts are adaptive

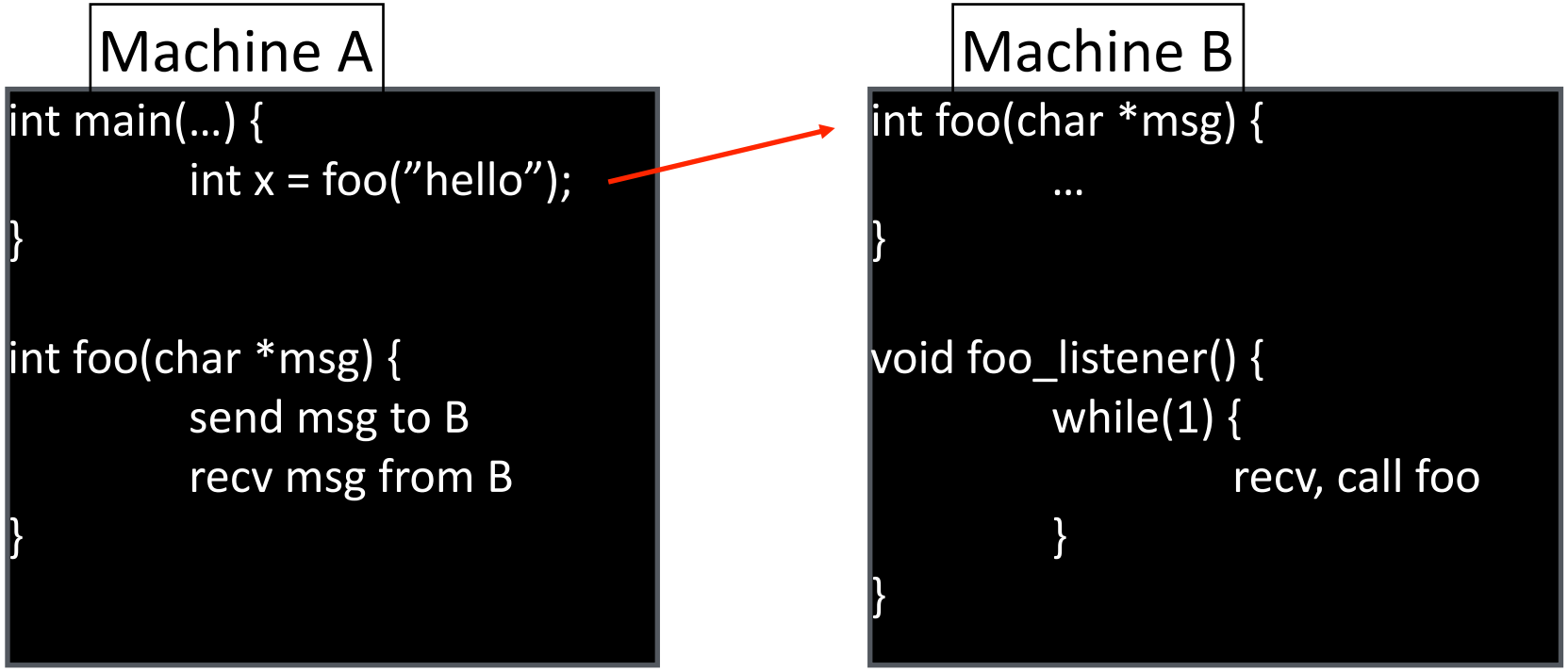
Communications Overview

- Raw messages: UDP
- Reliable messages: TCP
- Remote procedure call: RPC

RPC

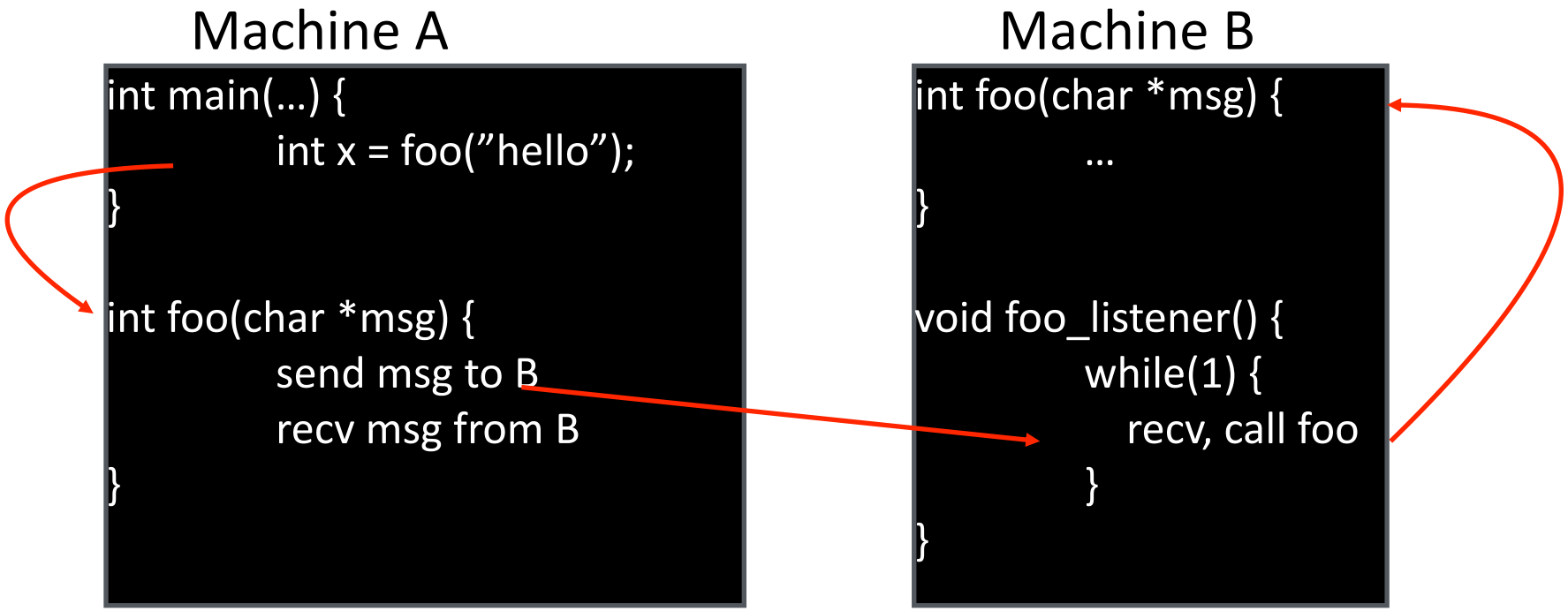
- Remote Procedure Call
- What could be easier than calling a function?
- Strategy: create wrappers so calling a function on another machine feels just like calling a local function
- Very common abstraction

RPC



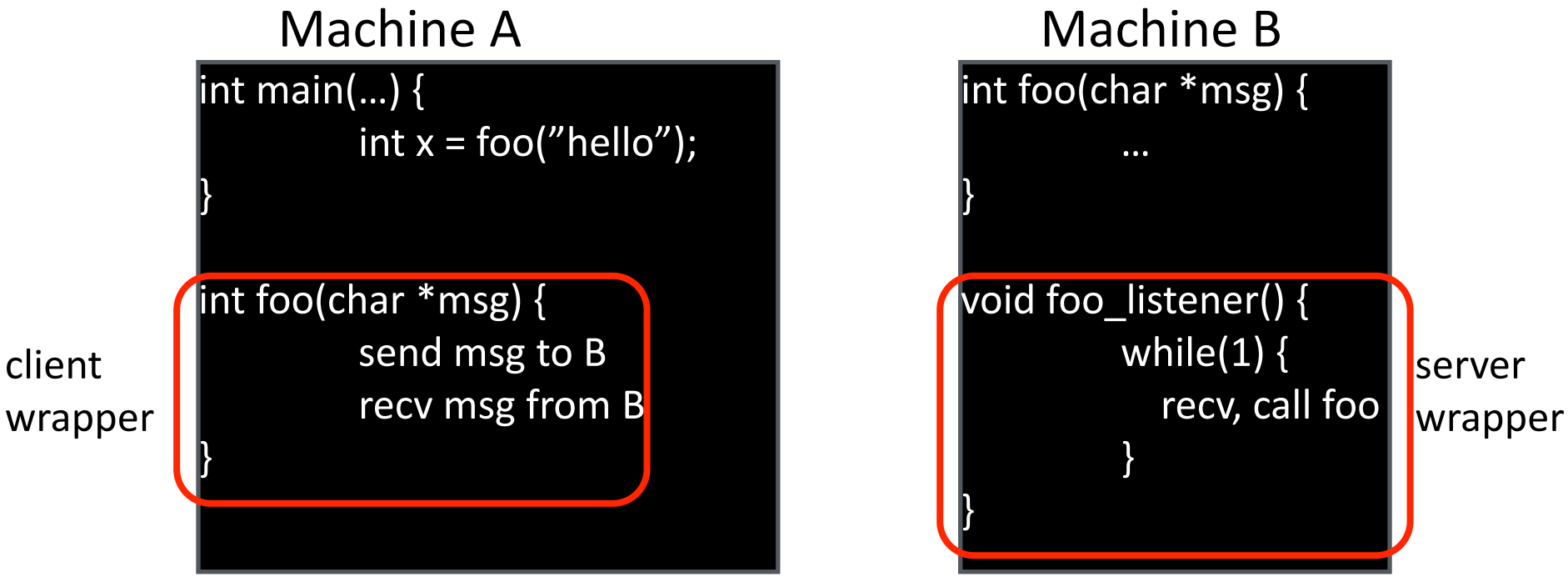
What it feels like for programmer

RPC



Actual calls

RPC



Wrappers

RPC Tools

- RPC packages help with two components
- (1) Runtime library
  - Thread pool
  - Socket listeners call functions on server
- (2) Stub generation
  - Create wrappers automatically
  - Many tools available (rpcgen, thrift, protobufs)

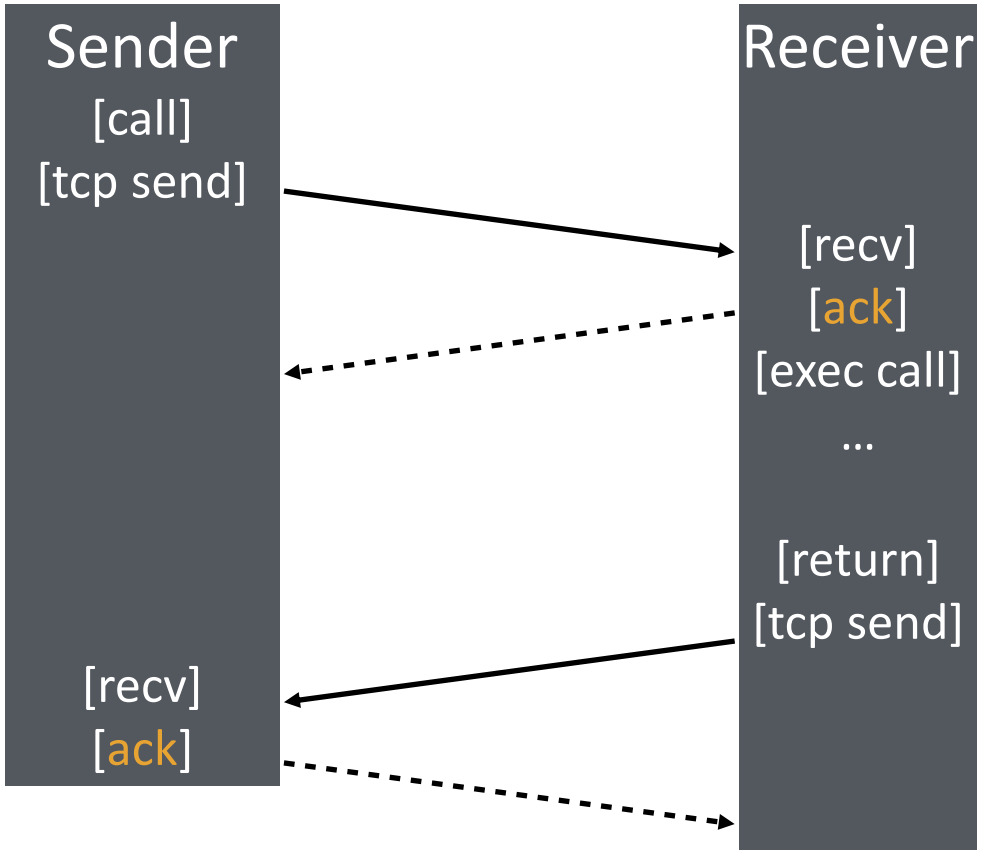
Wrapper Generation

- 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, 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

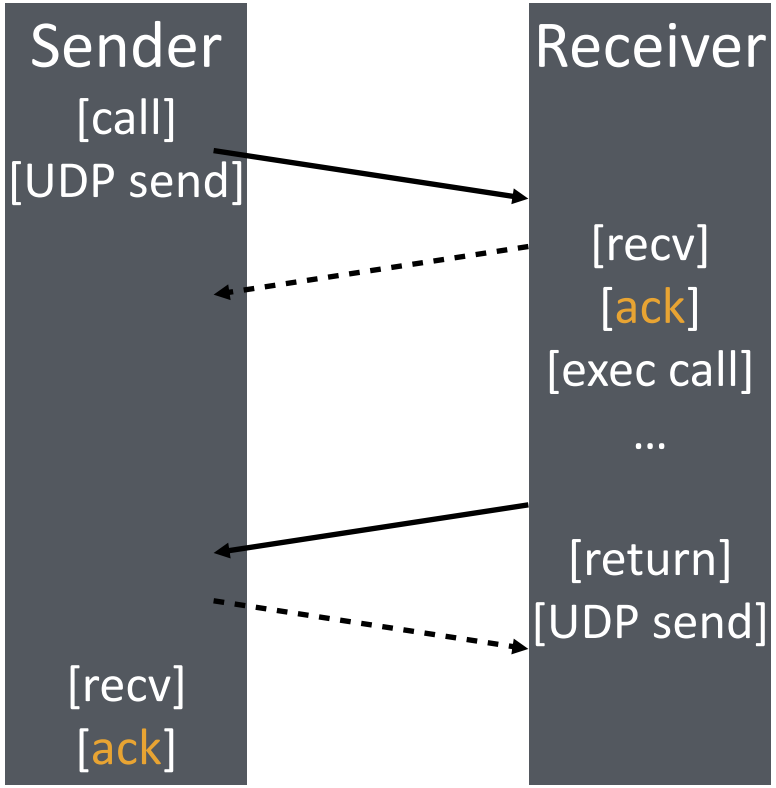
RPC over TCP?



Why wasteful?

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

- File systems are great use case for distributed systems

- Local FS:
  processes on same machine access shared files

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

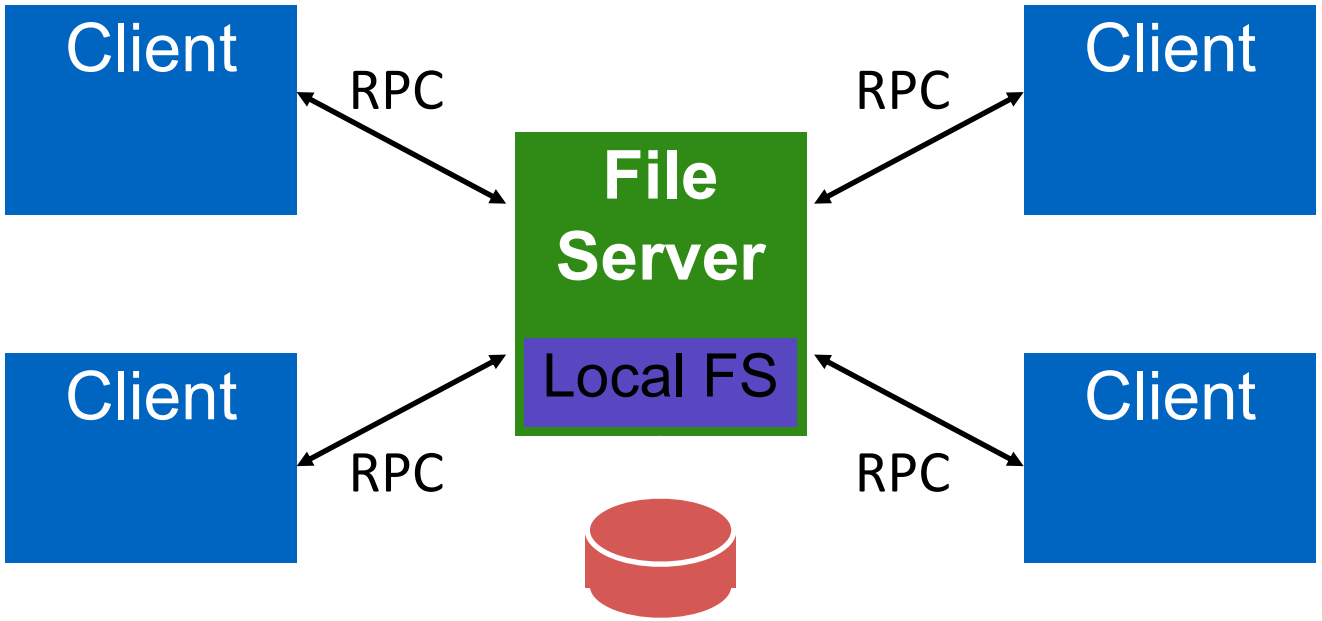
NFS

- Think of NFS as 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

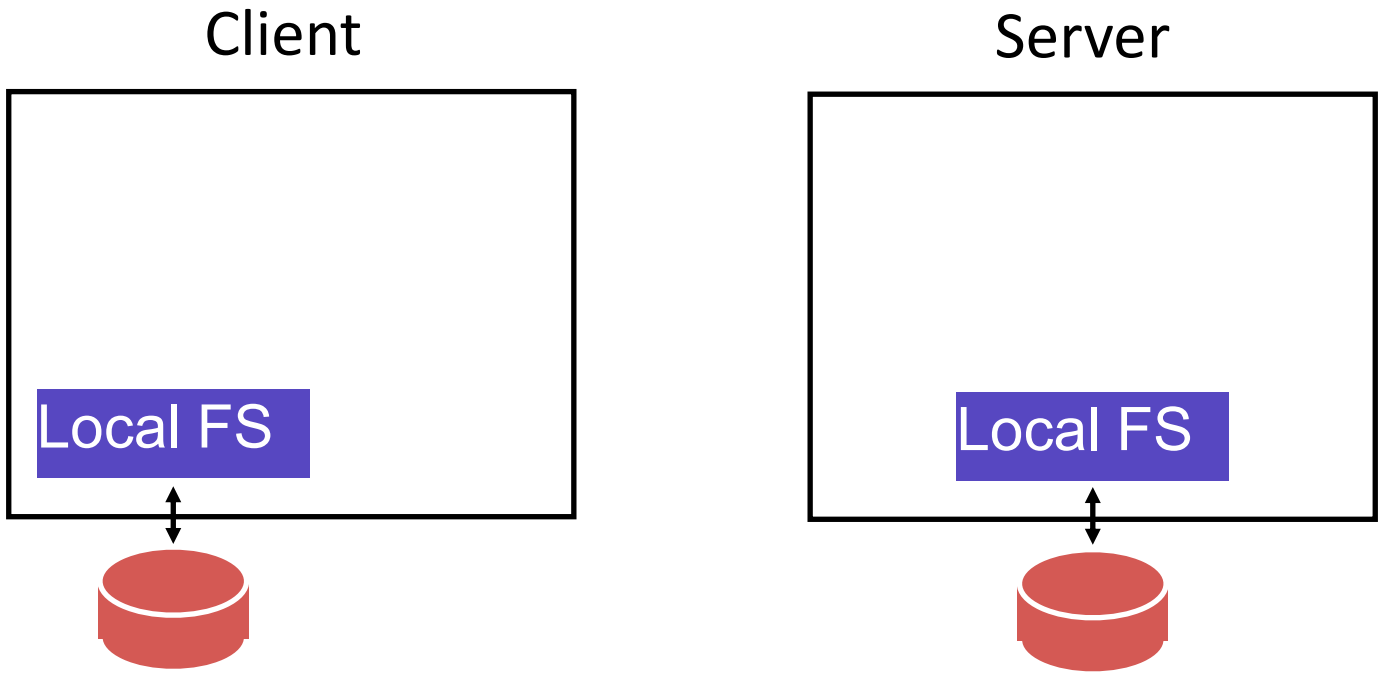
Overview

- Architecture
- Network API
- Write Buffering
- Cache

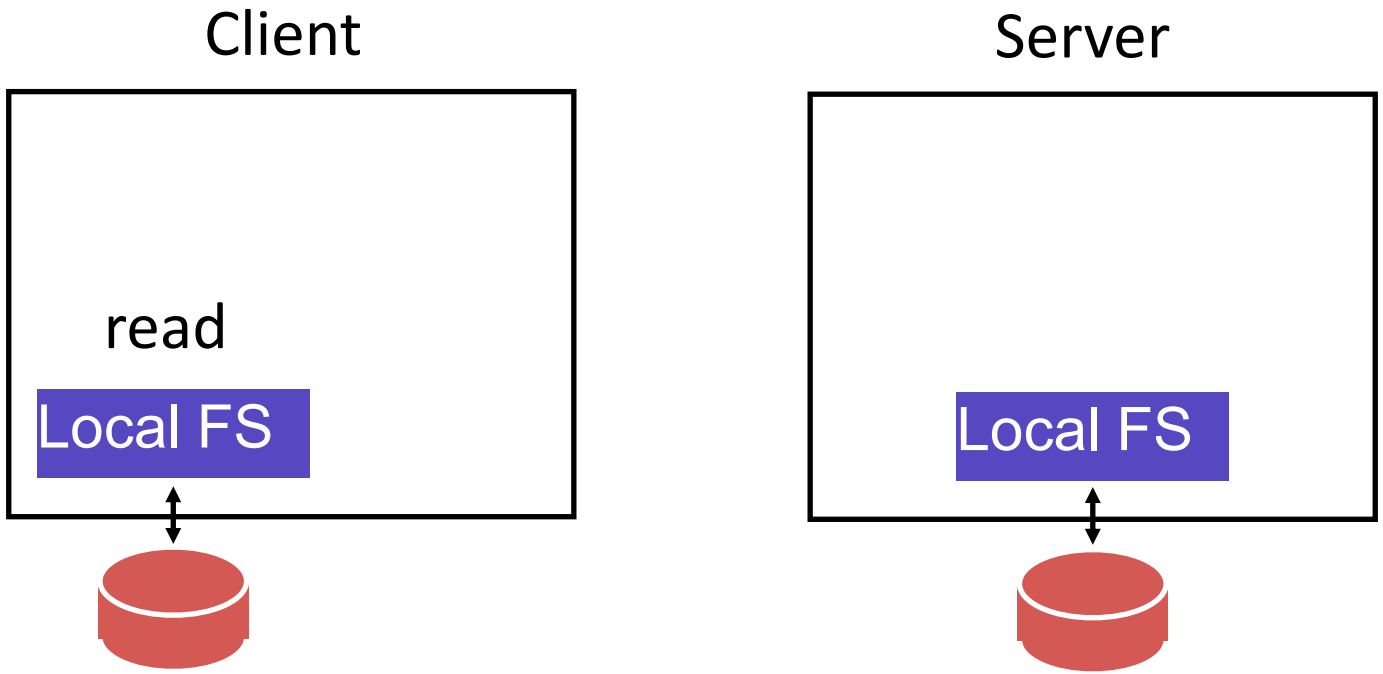
NFS Architecture



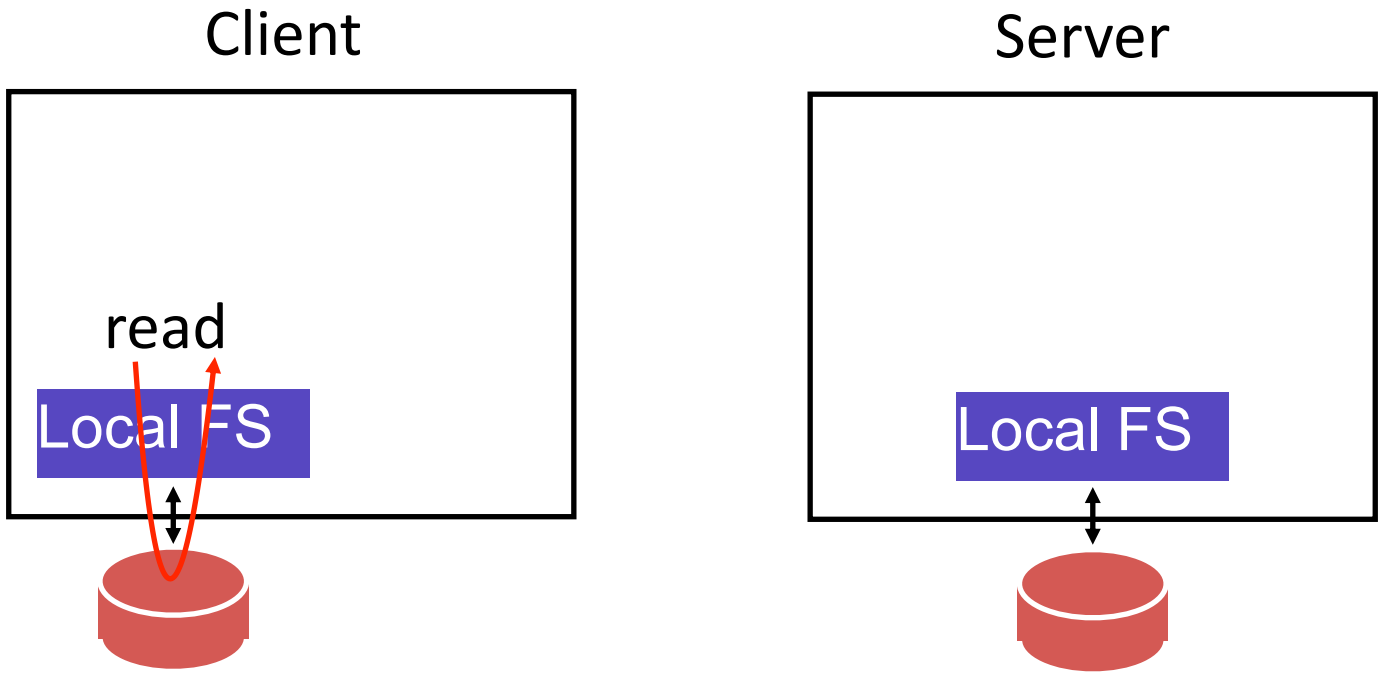
General Strategy: Export FS



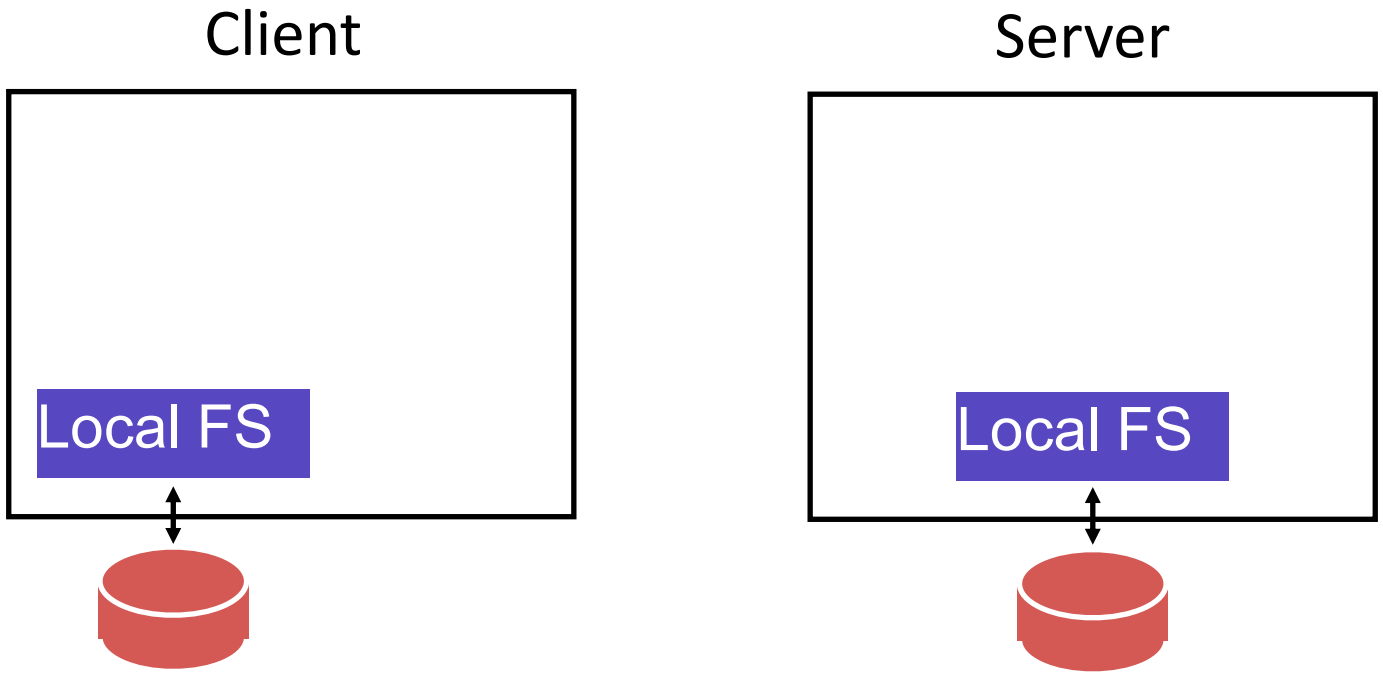
General Strategy: Export FS



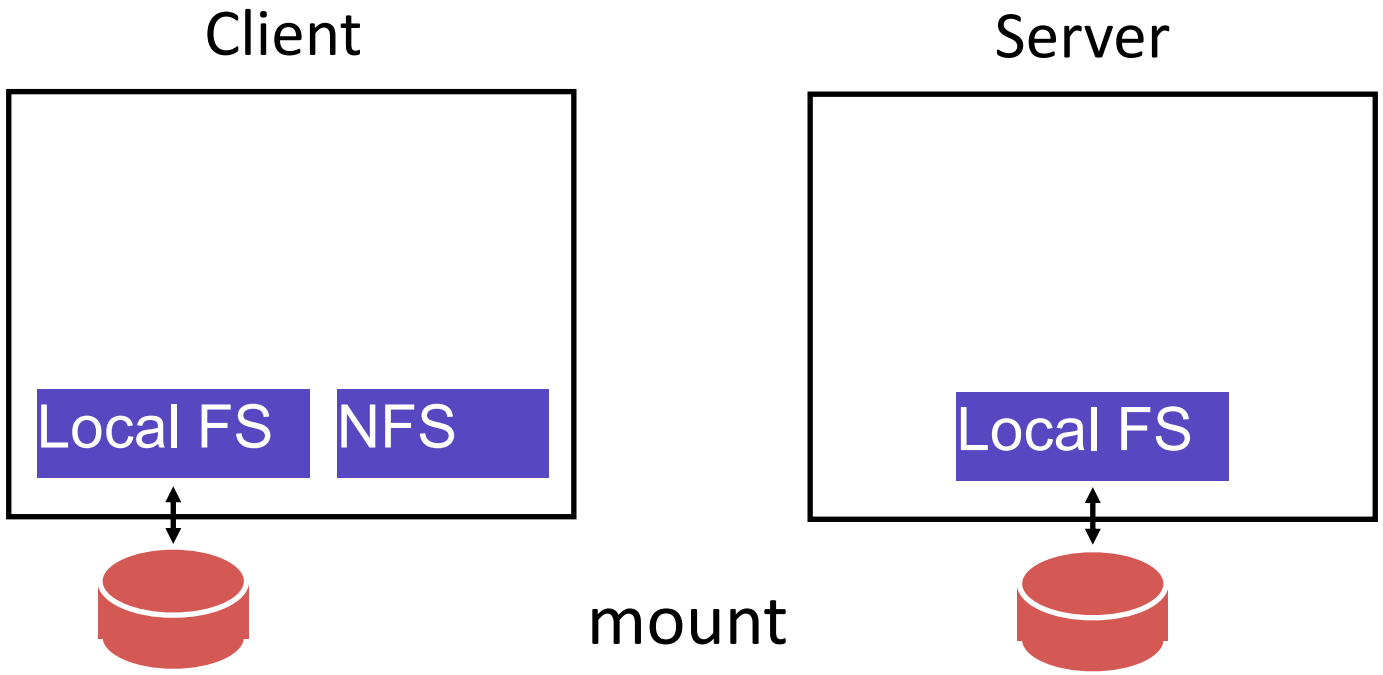
General Strategy: Export FS



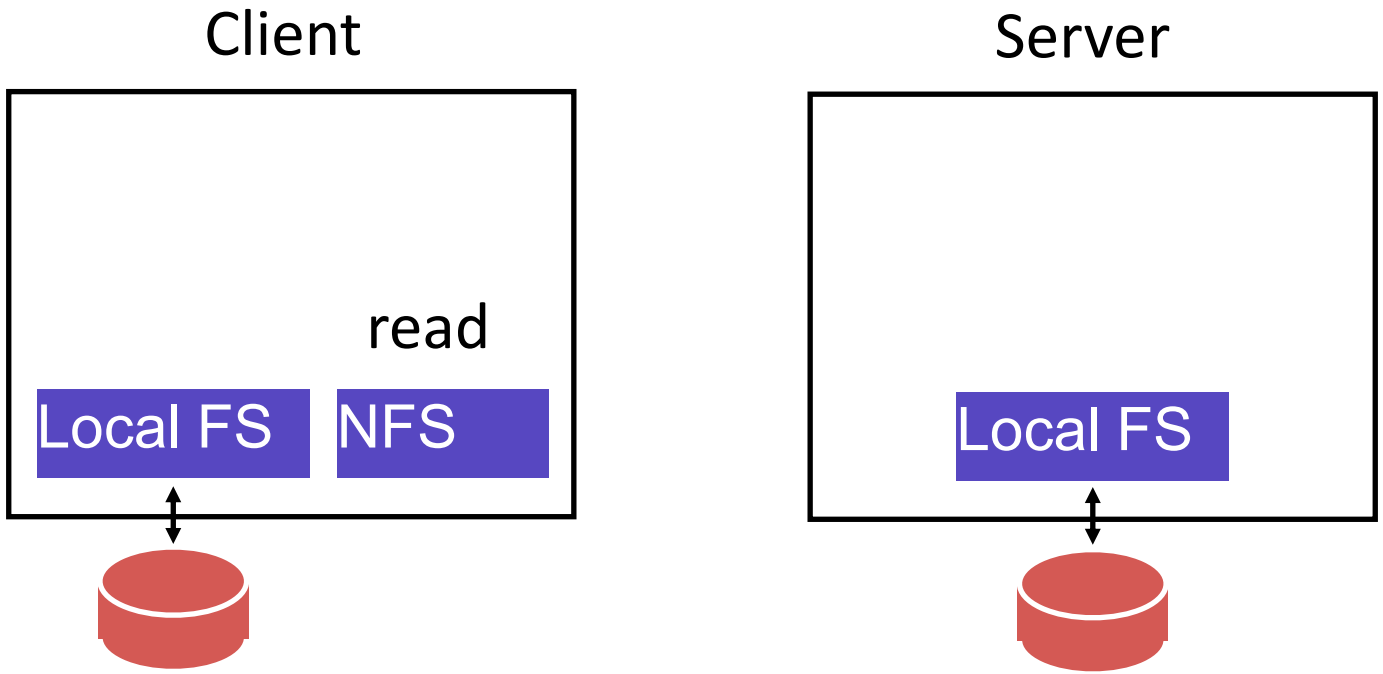
General Strategy: Export FS



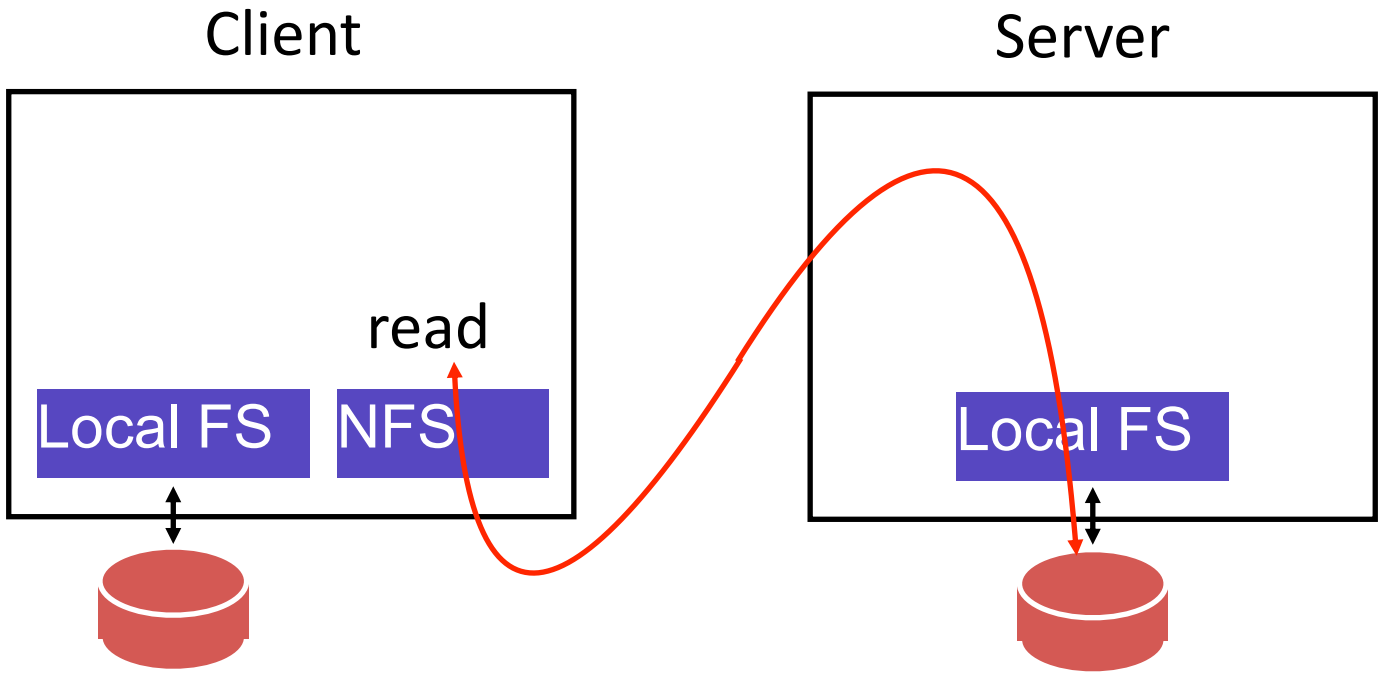
General Strategy: Export FS



General Strategy: Export FS



General Strategy: Export FS



NFS Questions

- 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 NFS cache consistency model?

Goals for NFS

- 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

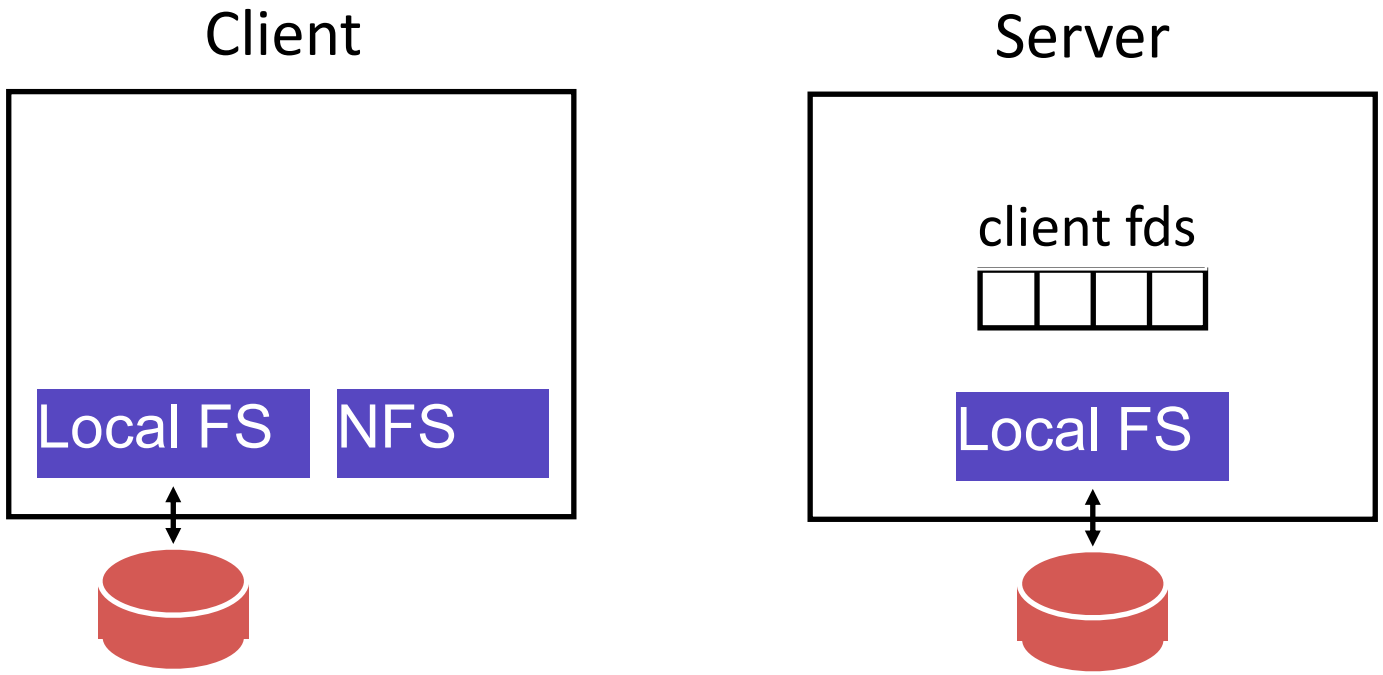
Strategy 1

- Attempt: Wrap regular UNIX system calls using RPC

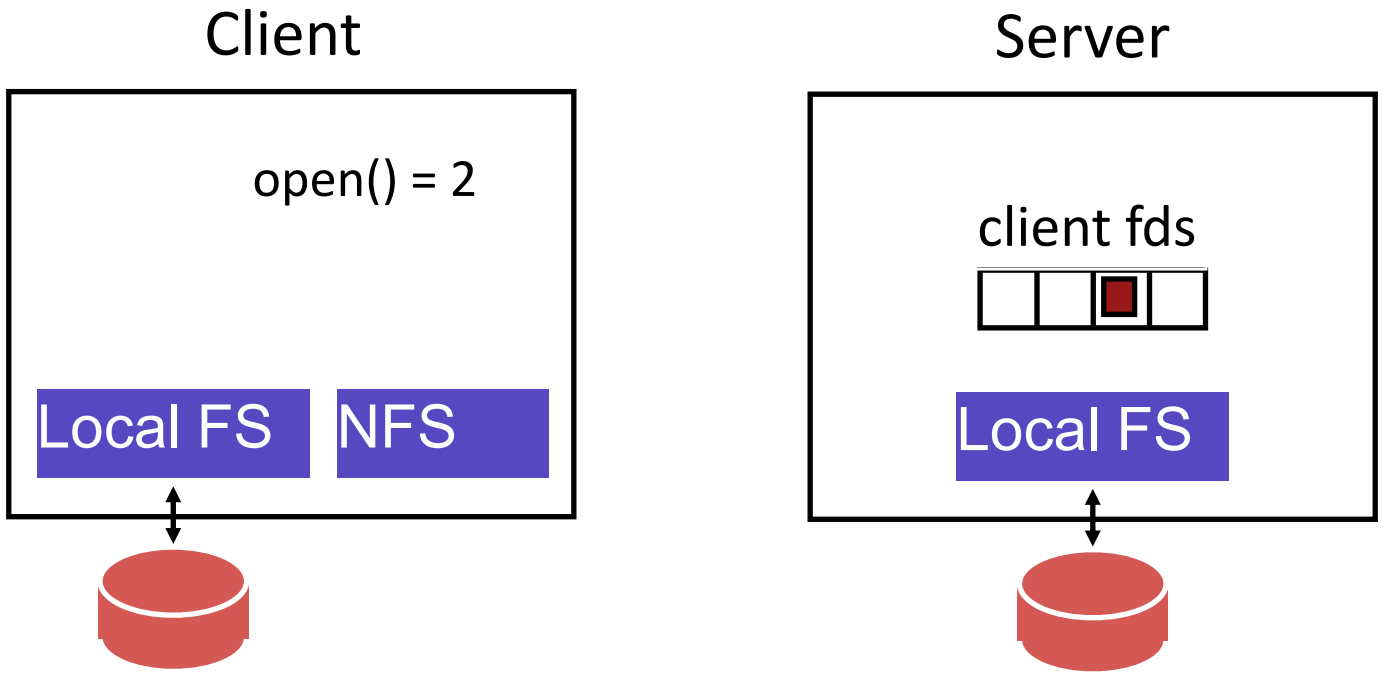
- open() on client calls open() on server
- open() on server returns fd back to client

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

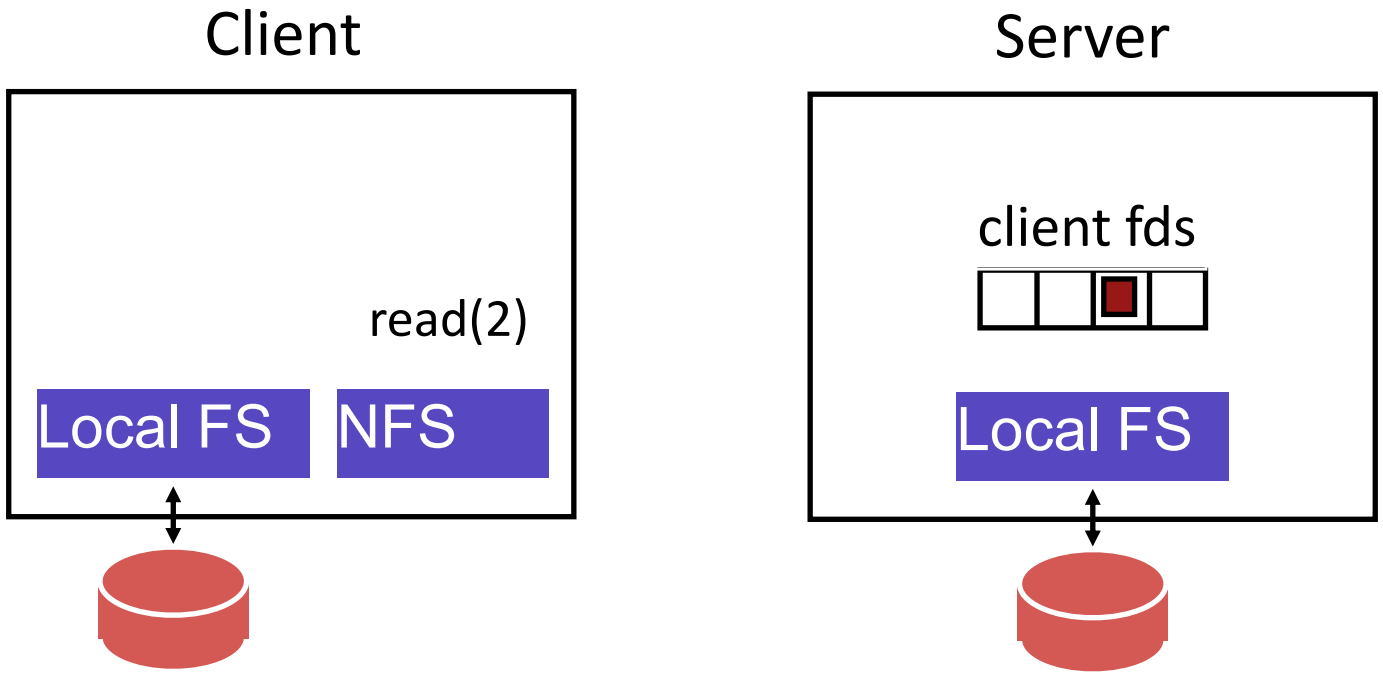
File Descriptors



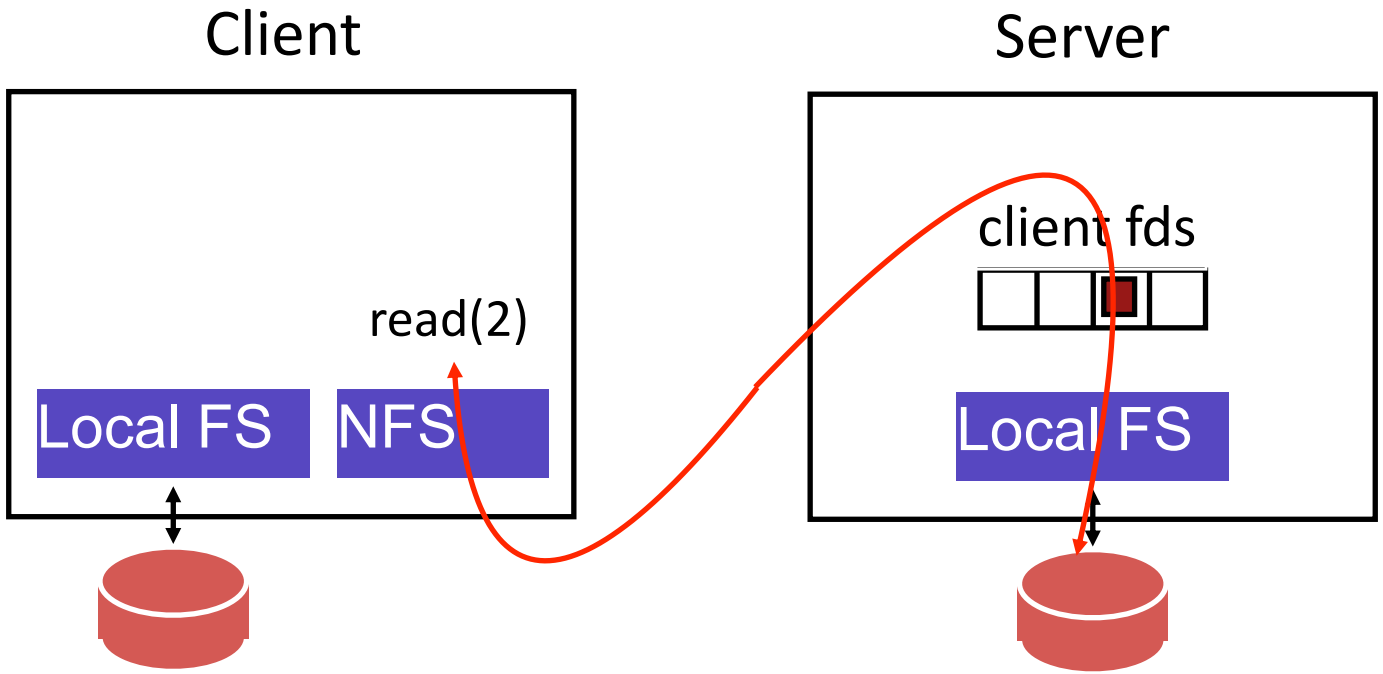
File Descriptors



File Descriptors



File Descriptors



Strategy 1 Problems

- What about crashes?
  - int fd = open(“foo”, O_RDONLY);
  - read(fd, buf, MAX);
  - read(fd, buf, MAX);

Server crash!

- Imagine server crashes and reboots between reads
  - Should behave as a slow read

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

- 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? 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);

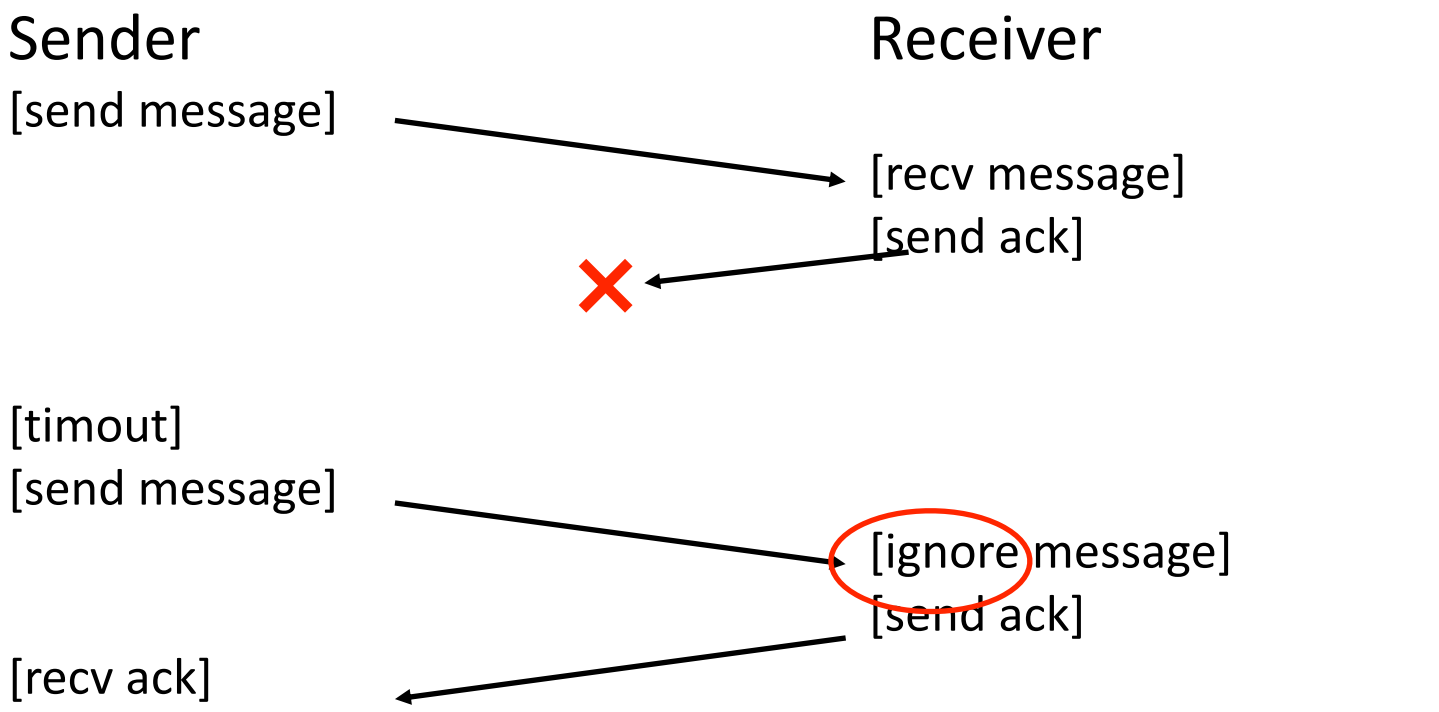
- File Handle = <volume ID, inode #, **generation #**>
- Opaque to client (client should not interpret internals)

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 retries, what happens when append() is retried?
- Problem: Why is it difficult to not replay append()?

Replica Suppression is Stateful



TCP suppresses repeated message

Problem: TCP is stateful

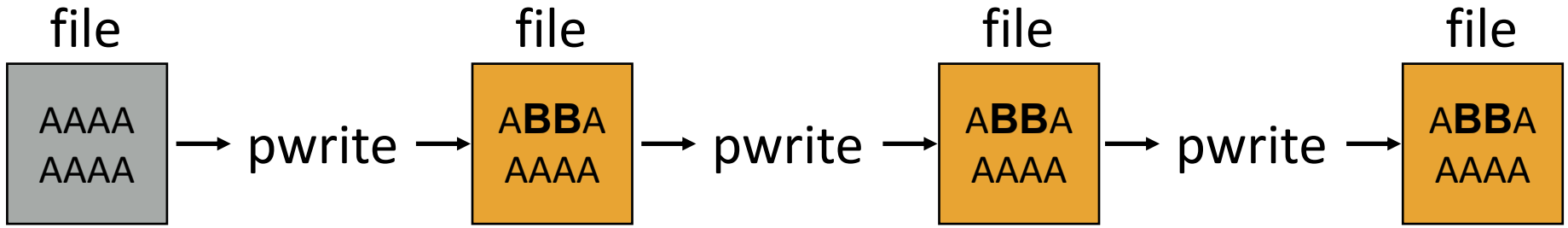
If server crashes, it forgets which RPC's have been executed!

Idempotent Operations

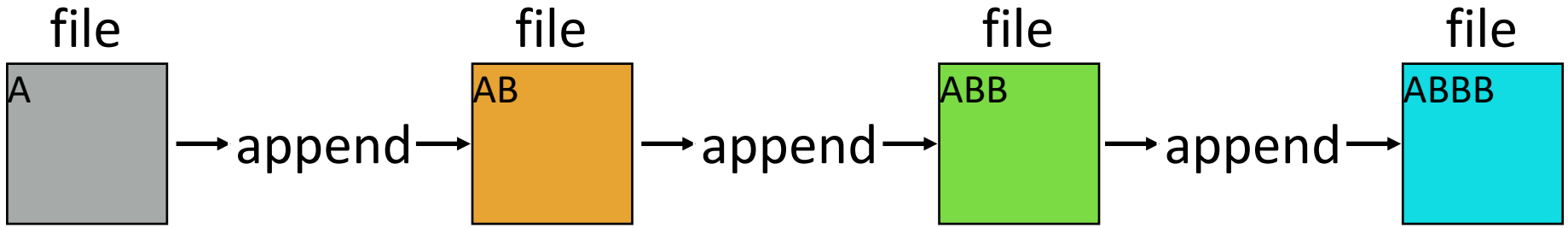
- Solution:
  Design API so no harm to executing function more than once

- If f() is idempotent, then:
  f() has the same effect as f(); f(); … f(); f()

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

Strategy 4: file handles

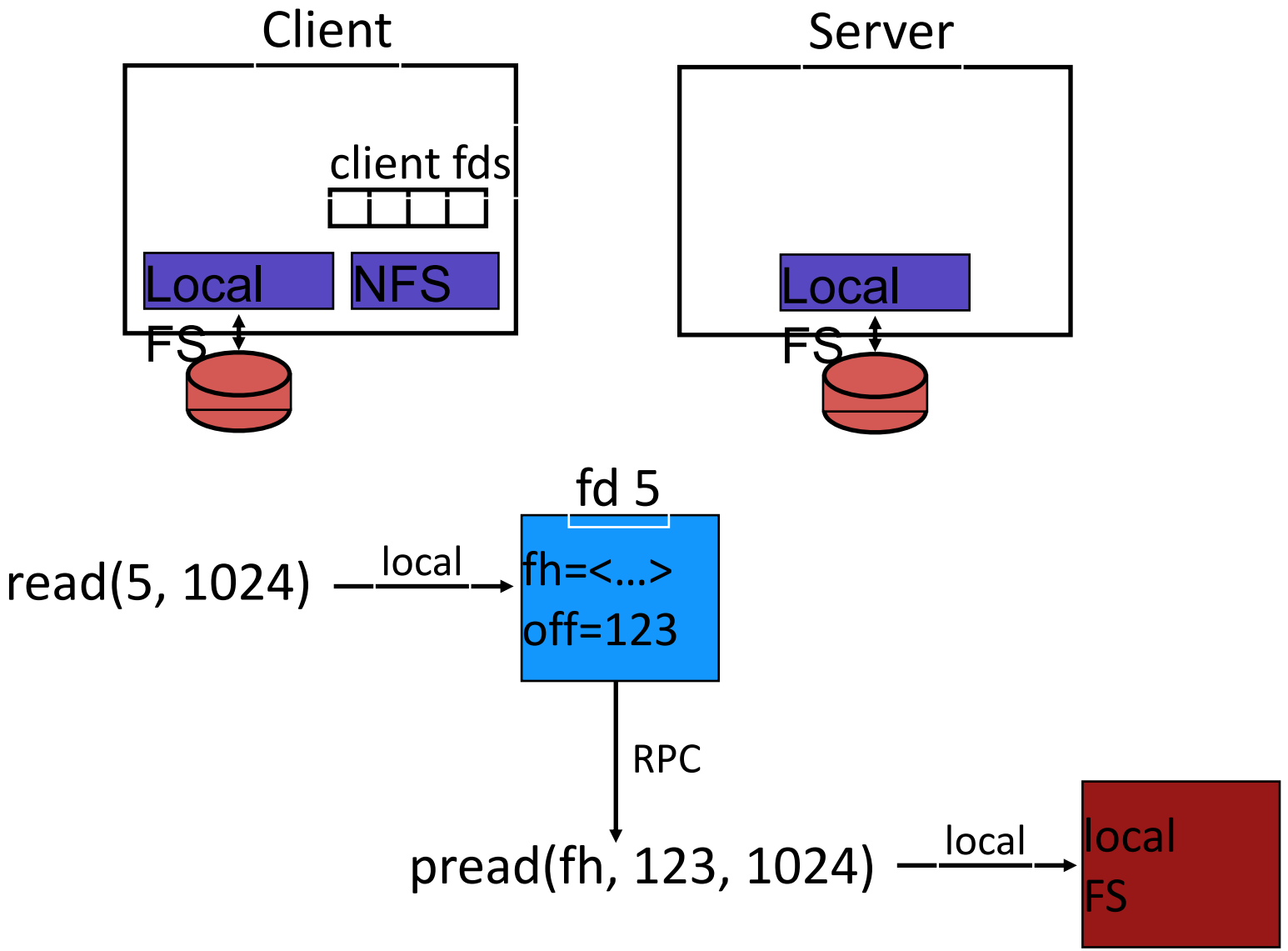
- fh = open(char *path);
- pread(fh, buf, size, offset);
- pwrite(fh, buf, size, offset);
- ~~append(fh, buf, size);~~

- 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
- It contains:
  - file handle
  - offset

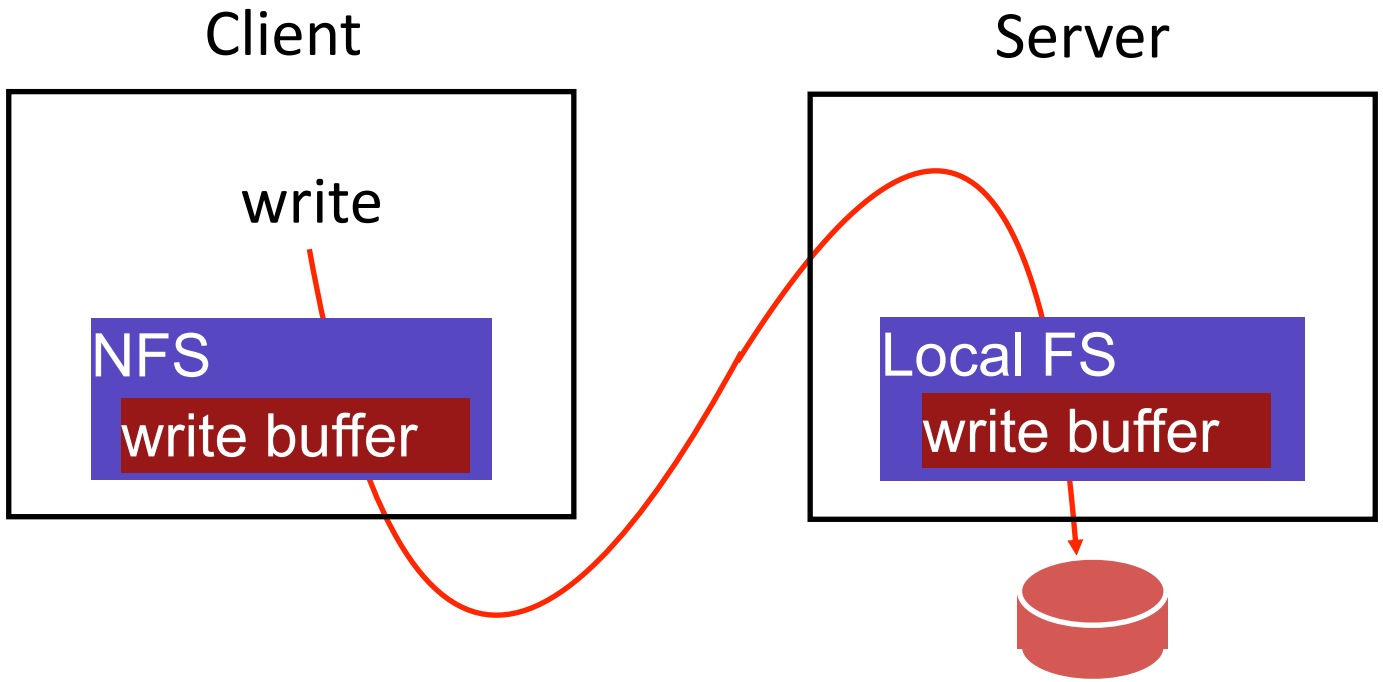
File Descriptors



Overview

- ~~Architecture~~
- ~~Network API~~
- Write Buffering
- Cache

Write Buffers

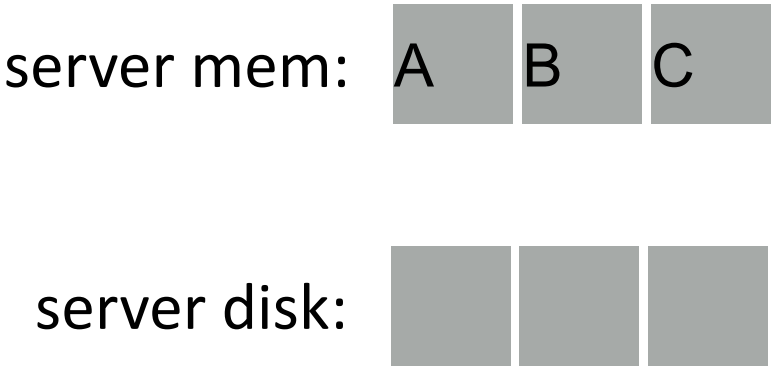


server acknowledges write before write is pushed to disk; what happens if server crashes?

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

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

- write X to 0

server mem: X 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

- 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

- write X to 0
- write Y to 1

server mem:

server disk:

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

- write X to 0
- write Y to 1

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: 

server disk: 

- write X to 0
- write Y to 1
- write Z 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

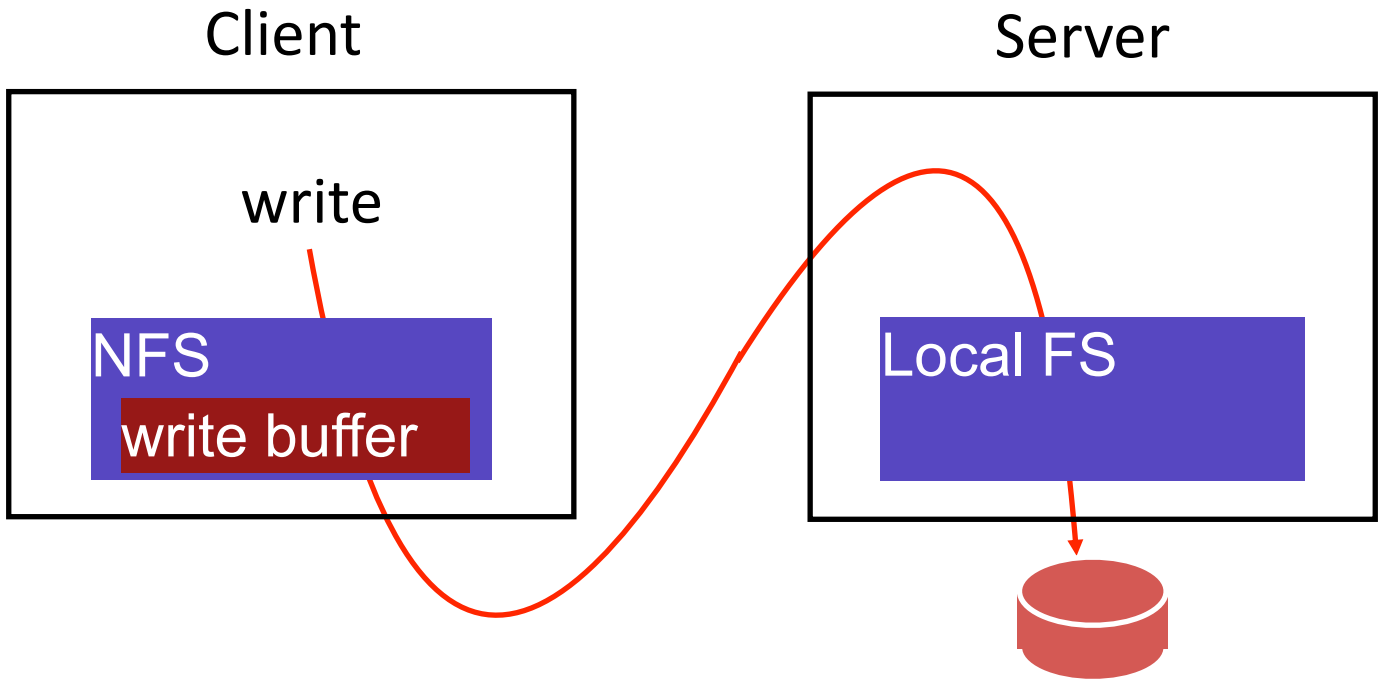
server mem: 

server disk: 

Problem:
No write failed, but disk state isn't correct

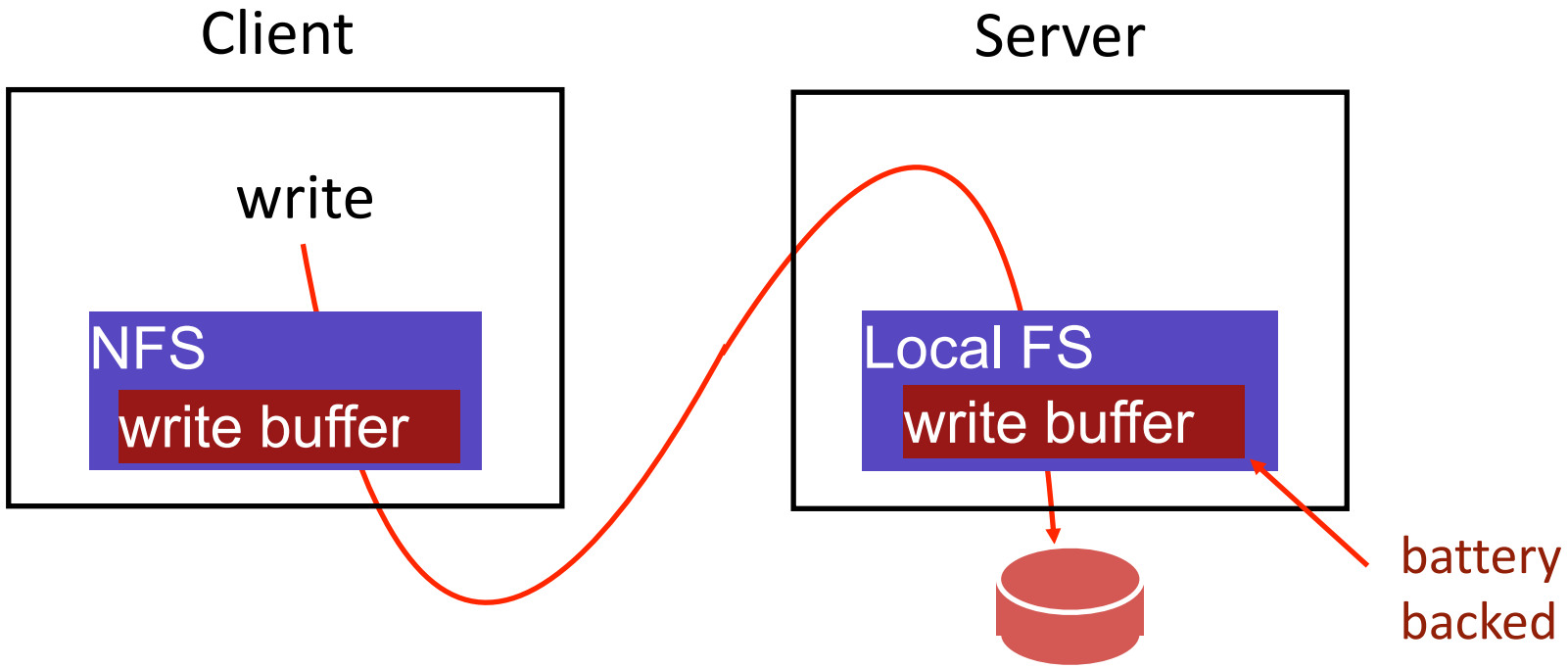
Solutions?

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)

Overview

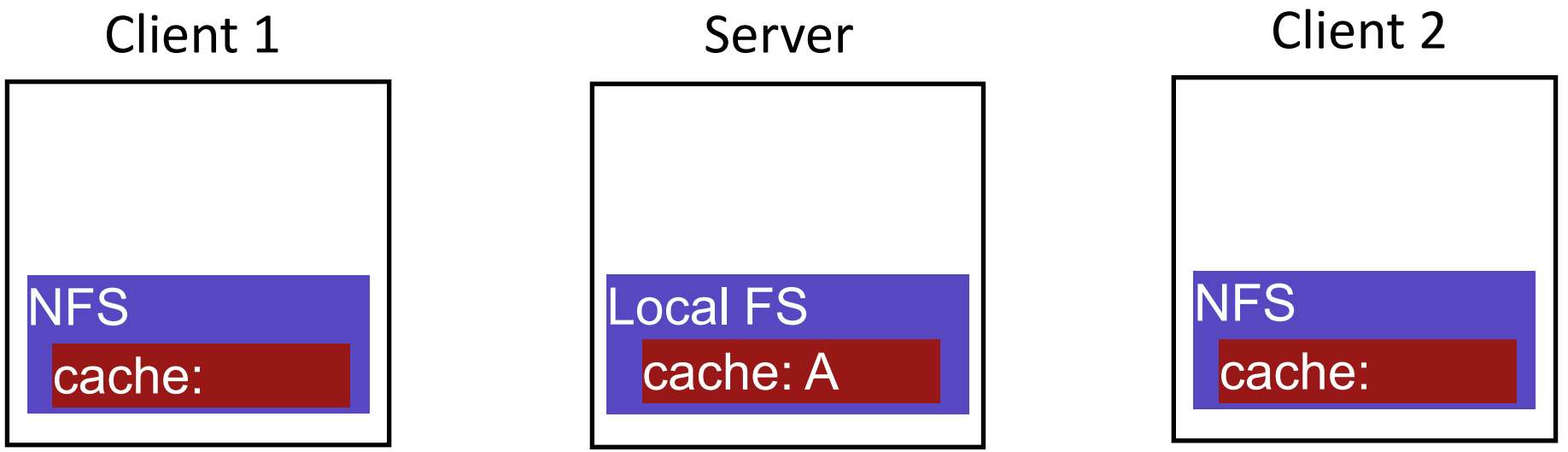
- ~~Architecture~~
- ~~Network API~~
- ~~Write Buffering~~
- Cache

Cache Consistency

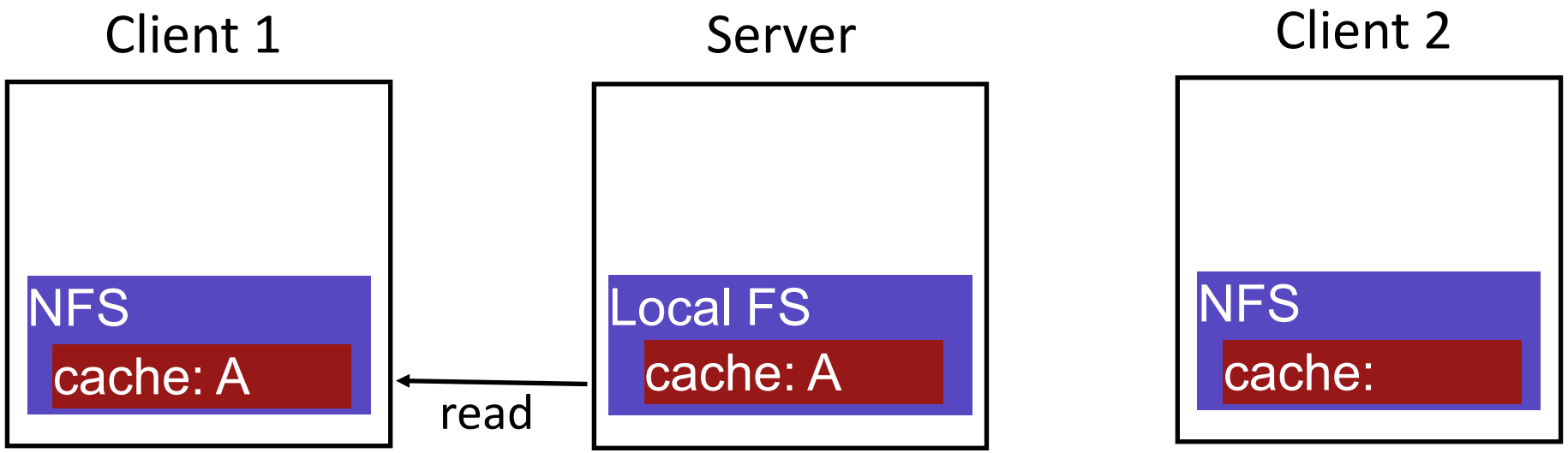
- NFS can cache data in three places:
  - Server memory
  - Client disk
  - Client memory

- How do we make sure all versions are in sync?

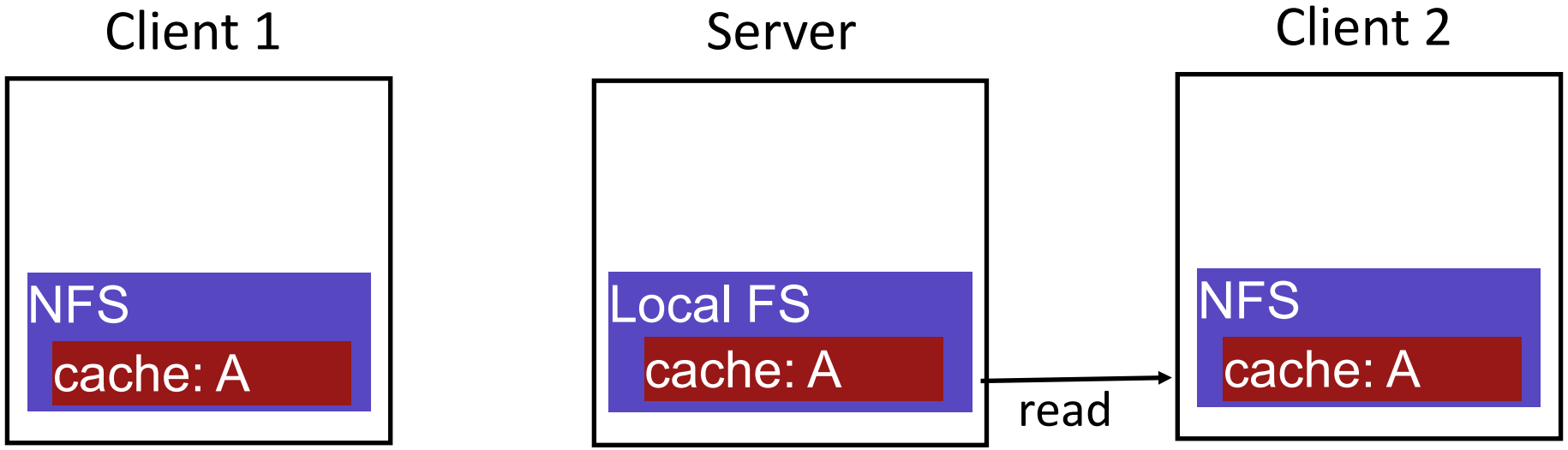
Cache



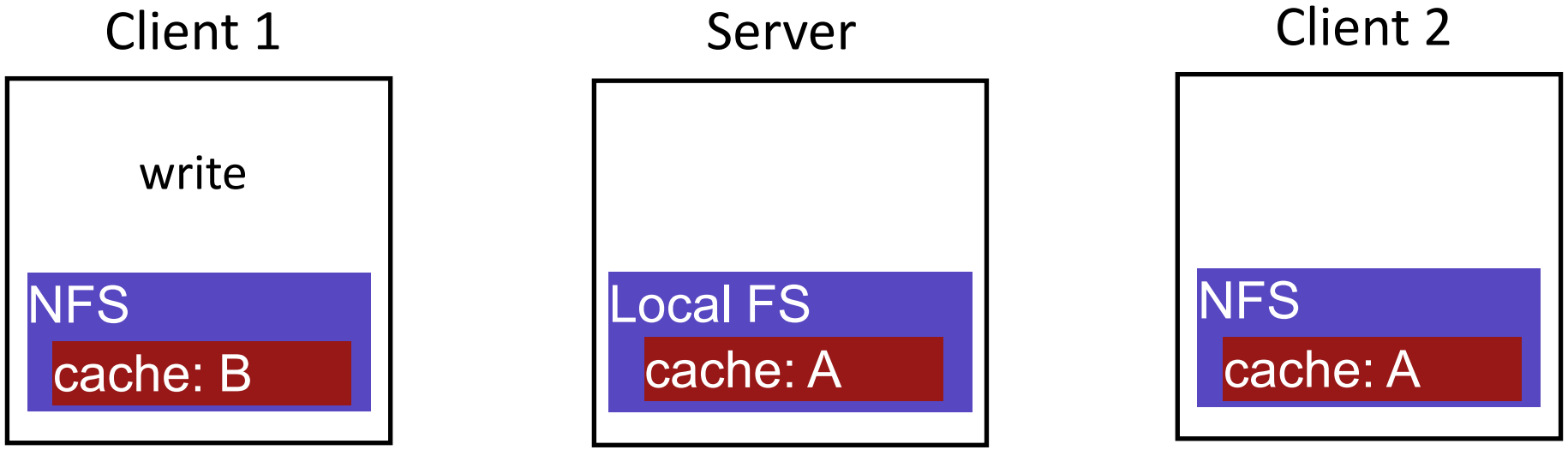
Cache


Client 1
Server
Client 2
NFS
cache: A
Local FS
cache: A
NFS
cache:
read

Cache


Client 1
Server
Client 2
NFS
cache: A
Local FS
cache: A
read
NFS
cache: A

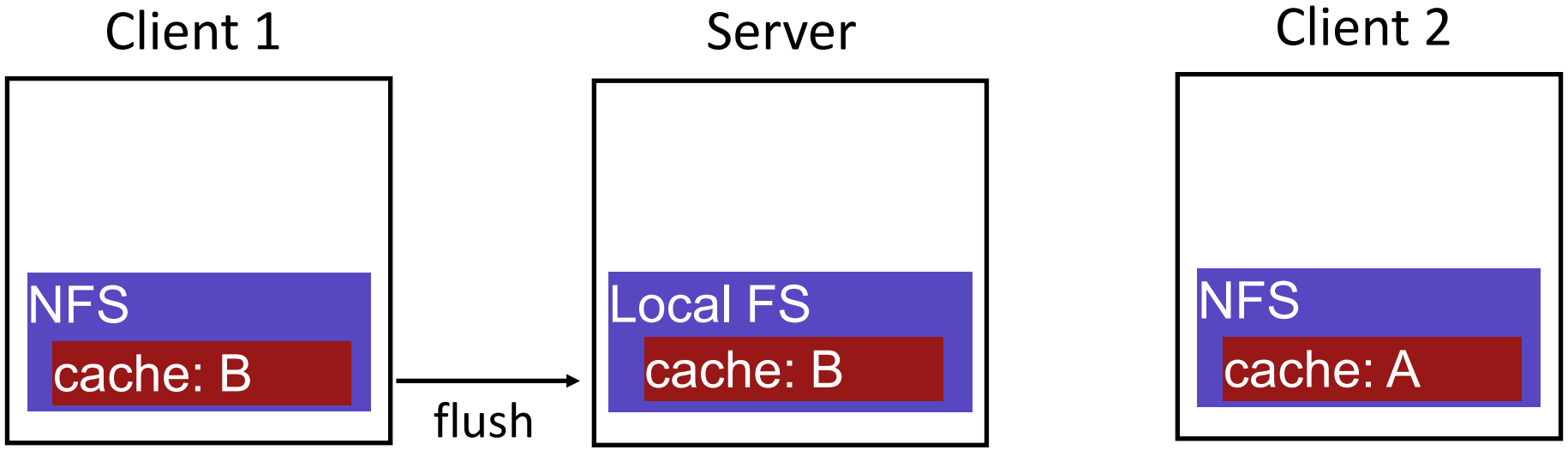
Cache



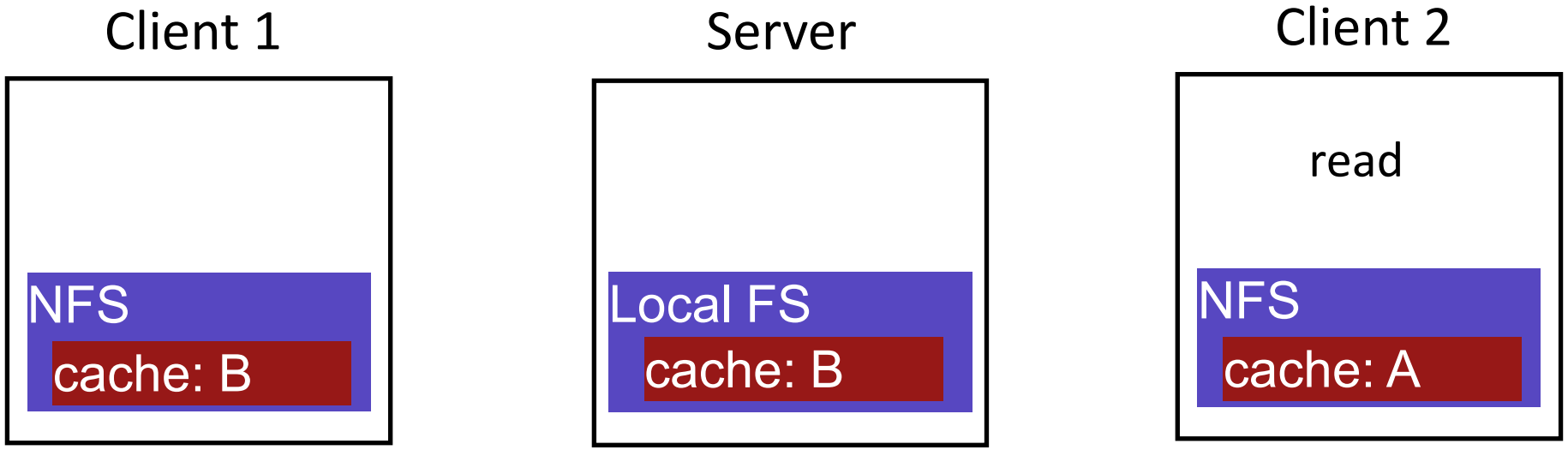
"Update Visibility" problem:
server doesn't have latest version

What happens if Client 2 (or any other client) reads data?
Sees old version (different semantics than local FS)

Cache


Client 1
Server
Client 2
NFS
cache: B
flush
Local FS
cache: B
NFS
cache: A

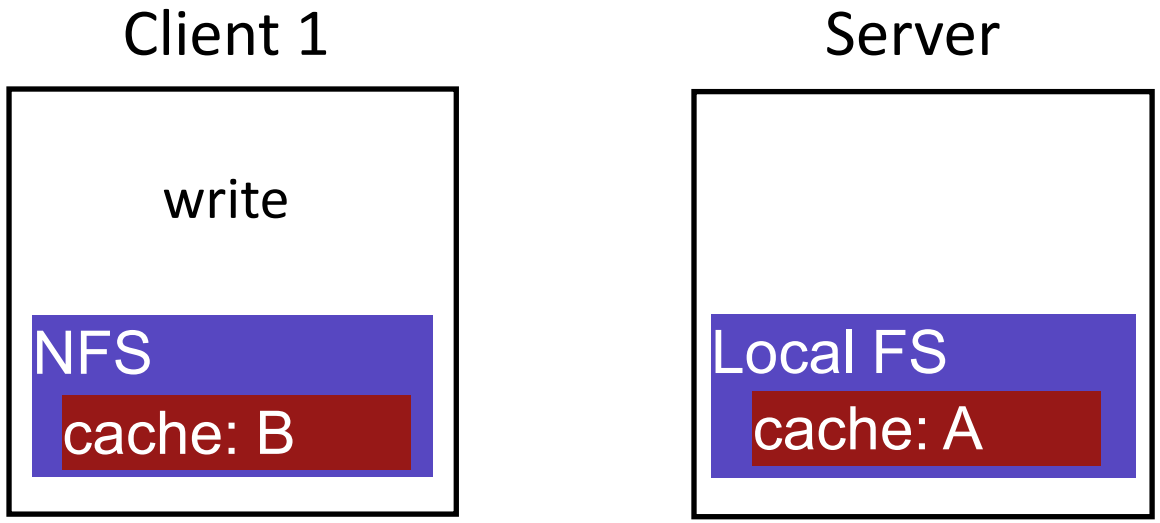
Cache



"Stale Cache" problem:
client 2 doesn't have latest version

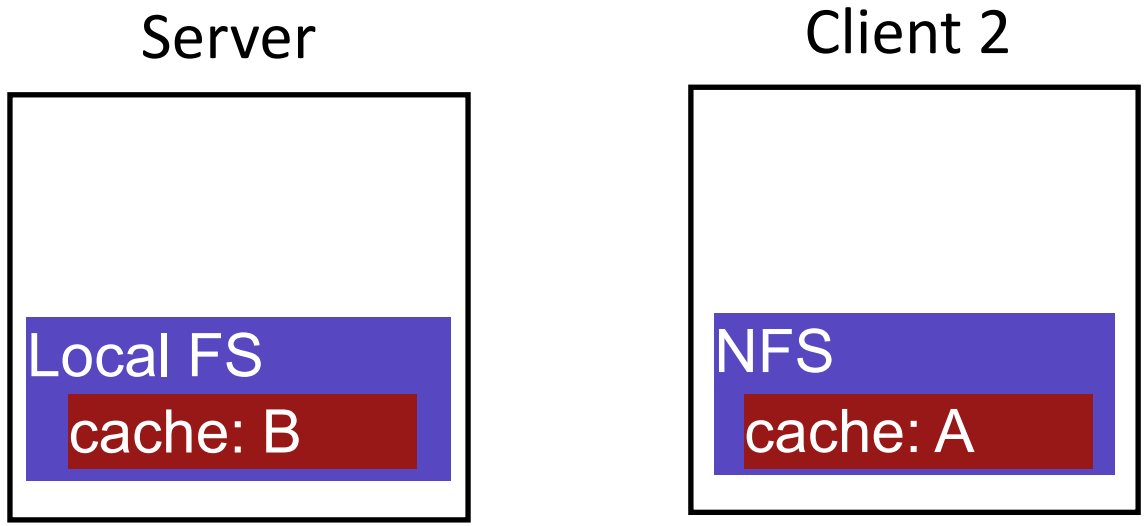
What happens if Client 2 reads data?
Sees old version (different semantics than local FS)

Problem 1: Update Visibility



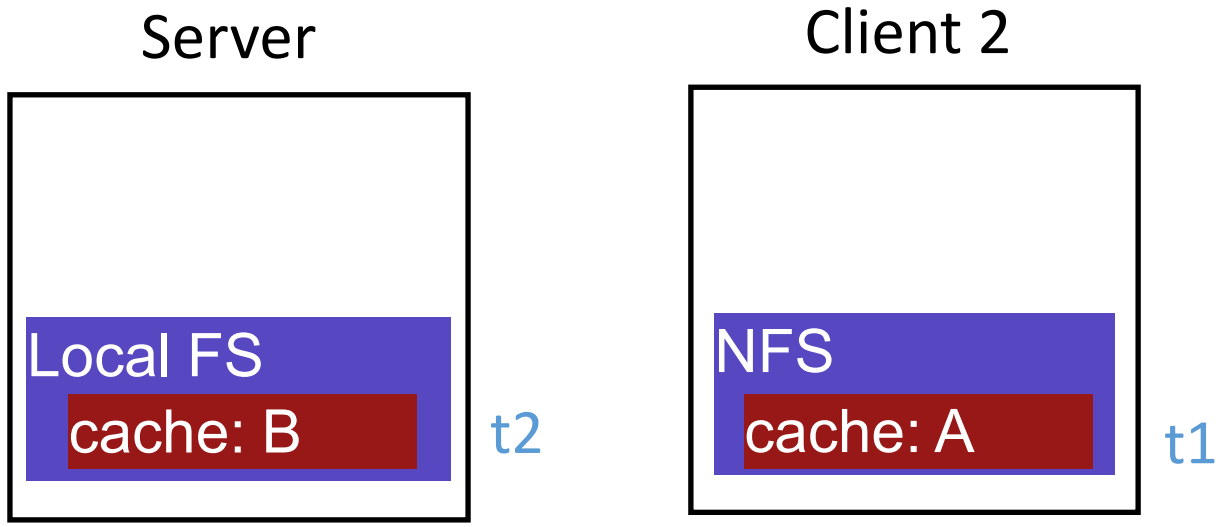
- When client buffers a write, how can server (and other clients) see update?
  - Client flushes cache entry to server
- When should client perform flush?
- NFS solution: flush on fd close

Problem 2: Stale Cache



- 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

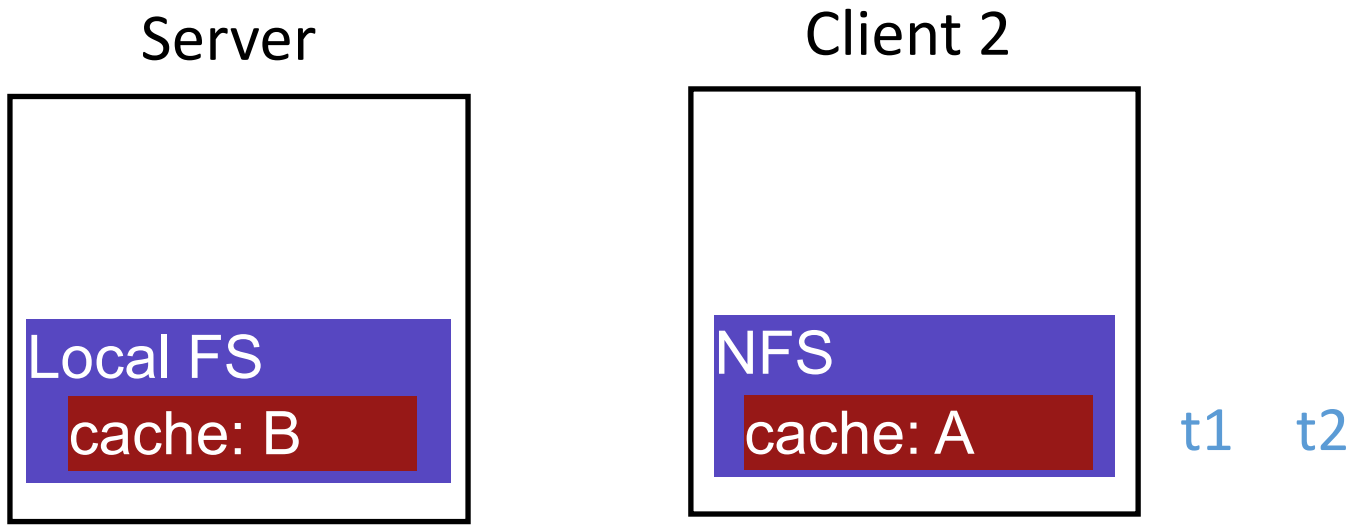


- Client cache records time when data block was fetched (t1)
- Before using data block, client does a STAT request to server
  - gets last modified timestamp for this file (t2) (not block…)
  - compare to cache timestamp
  - refetch data block if changed since timestamp (t2 > t1)

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: doing things twice never hurts
- 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