We acknowledge and pay our respects to the Kaurna people, the traditional custodians whose ancestral lands we gather on.

We acknowledge the deep feelings of attachment and relationship of the Kaurna people to country and we respect and value their past, present and ongoing connection to the land and cultural beliefs.

Drop-in session next week

Cruz: Wednesday 11-12pm

Olaf: Friday 11-1pm (Summary lecture on “what we did in class”)

Ingkarni Wardli B18

(No other drop-in sessions anymore!)

Q1

Which one of the following statements is correct?

	Answer
×	Hard links store the target as a string, while symbolic links store the target as an i-node number.
✓	Symbolic links can refer to files on other filesystems, whereas hard links can only refer to files on the same filesystem.
×	All hard links to a given file must reside in the same directory.
×	A given file may have at most one symbolic link referring to it

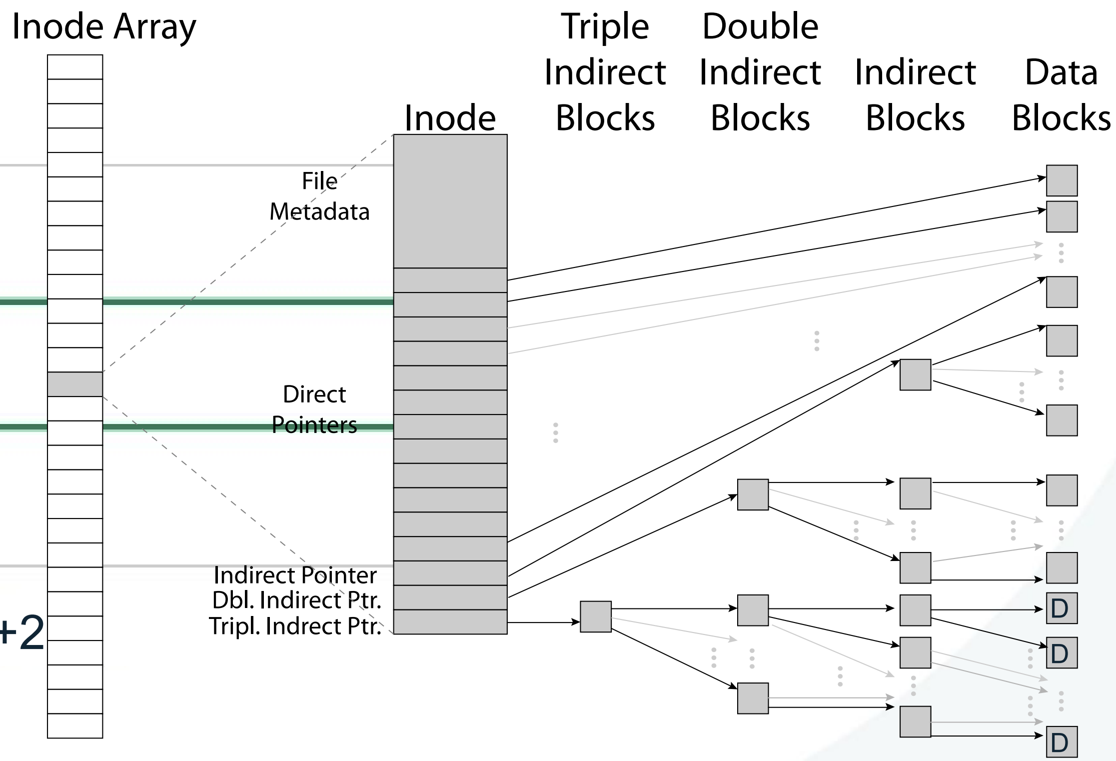
THE UNIVERSITY of ADELAIDE

Q2

Assume if a single-indirect block can reference 256 (2^8) data blocks, calculate how many data blocks a triple-indirect block (indirectly) can reference?

Answer

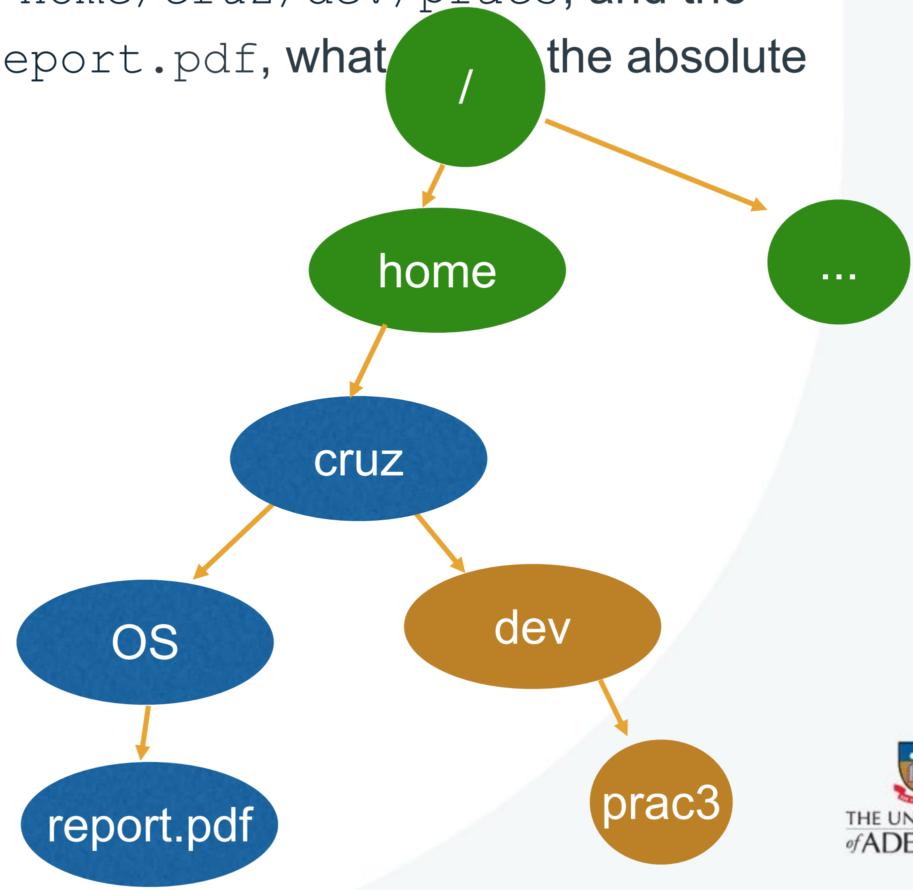
- ✗ 65,536 (2^{16})
- ✓ 16,777,216 (2^{24})
- ✗ 6,842,752 ($2^{16} + 2^{24}$)
- ✗ 16,843,008 ($2^8 + 2^{16} + 2$



Q3

If the current working directory of a process is `/home/cruz/dev/prac3`, and the process attempts to open the file `../../OS/report.pdf`, what the absolute path name of the file it tries to open?

	Answer
✗	/home/cruz/dev/OS/report.pdf
✓	/home/cruz/OS/report.pdf
✗	/home/cruz/dev/prac3/OS/report.pdf
✗	/home/OS/report.pdf



Q4

Which of the following is a disadvantage of allocating block on disk in a contiguous fashion?

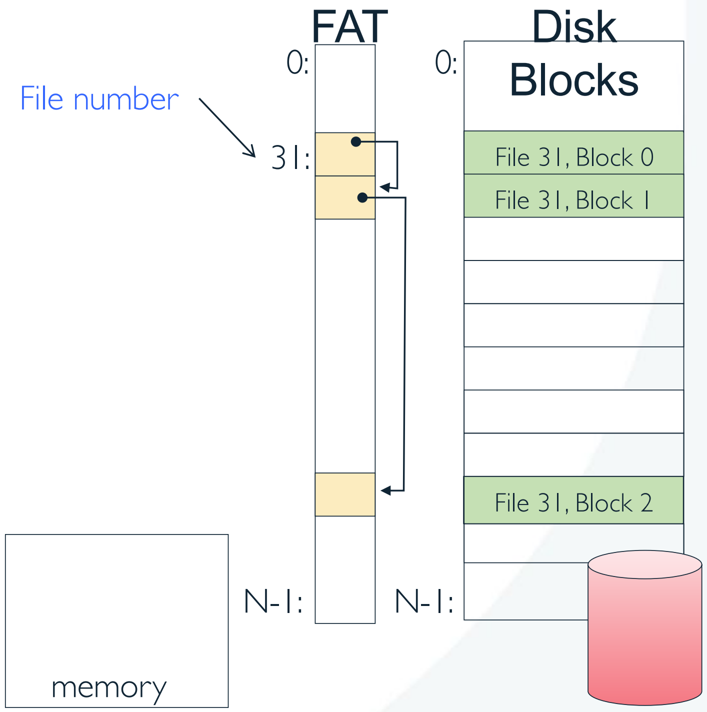
	Answer
✕	All files must be of the same size.
✓	Often, files cannot be expanded in a contiguous fashion without moving them to another location on disk.
✕	Once a file has been deleted, the space it occupied is given to another file. If the file is small, most space is wasted.
✕	A section of the file cannot be deleted - only the whole file can be deleted

THE UNIVERSITY of ADELAIDE

Q5

Which one of the following statements about FAT filesystem is true?

	Answer
✓	FAT supports both sequential and random access, but random access is slower on average
✕	FAT supports both sequential and random access, but sequential access is slower on average.
✕	FAT supports neither sequential nor random access.
✕	FAT only support sequential access.



THE UNIVERSITY of ADELAIDE

Q6

Why does the operating system provide system calls for reading directories, instead of having applications read and interpret the on-disk data structures directly?

	Answer
✕	To ensure filesystem reliability in the event of a crash.
✕	So that multiple types of files can be stored in a directory, father than just those that the application supports.
✓	So that applications can easily work with many different types of filesystems.
✕	To prevent malicious applications from overwriting an executable file with code.

THE UNIVERSITY of ADELAIDE

Q7

In the Fast File System (FFS), what is the purpose of the inode structure, and what information does it typically store?

	Answer
✕	The inode structure is used for organizing free space and managing disk fragmentation.
✕	The inode stores some small data directly in the inode, and uses linked list references to other block on disk for larger files.
✓	The inode structure stores metadata about files, including file permissions, timestamps, and disk block pointers.
✕	The maximum file size is bound by the number of blocks you can access through direct blocks.

THE UNIVERSITY of ADELAIDE

Q8

What purpose does journaling serve for a file system?

	Answer
✓	It ensures that on-disk structures remain constant.
✕	It contains all the metadata structures for the file system
✕	It records all reads and write performed on the file system
✕	It guarantees that data written to disk is never lost

THE UNIVERSITY of ADELAIDE

Q9

In log-structured file systems, which of the following statements is true regarding their advantages and disadvantages?

	Answer
✗	They provide faster random access and minimize write amplification.
✗	They consume less disk space and perform garbage collection efficiently.
✓	They improve write performance and simplify crash recovery.
✗	They are less prone to data fragmentation and have no concerns with wear leveling.

THE UNIVERSITY of ADELAIDE

Q10

In the Fast File System (FFS), what is the purpose of block/cylinder groups, and how do they contribute to file system management?

	Answer
✗	Block/cylinder groups are used to store large files exclusively, reducing fragmentation.
✗	Block/cylinder groups improve random access performance by storing metadata in a contiguous layout.
✗	Block/cylinder groups provide data redundancy through mirroring and RAID.
✓	Block/cylinder groups help distribute and manage disk space efficiently and contain a portion of the file system metadata.

THE UNIVERSITY of ADELAIDE

COMP SCI 3004 Operating Systems

Week 12 – Distributed Systems

NFS, AFS, CDN and a bit of Security

make history.

THE UNIVERSITY of ADELAIDE

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

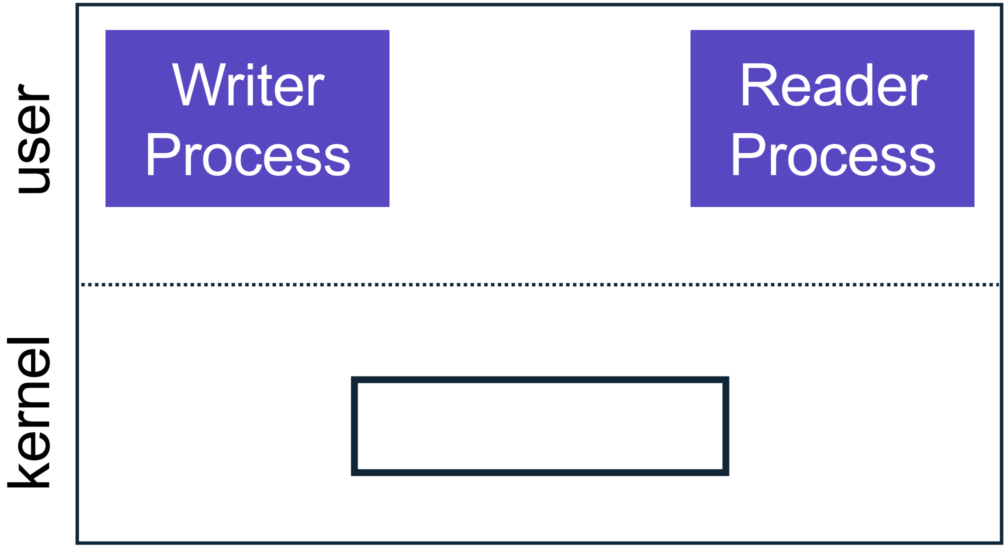
New Challenges

System failure: need to worry about <u>partial</u> failure

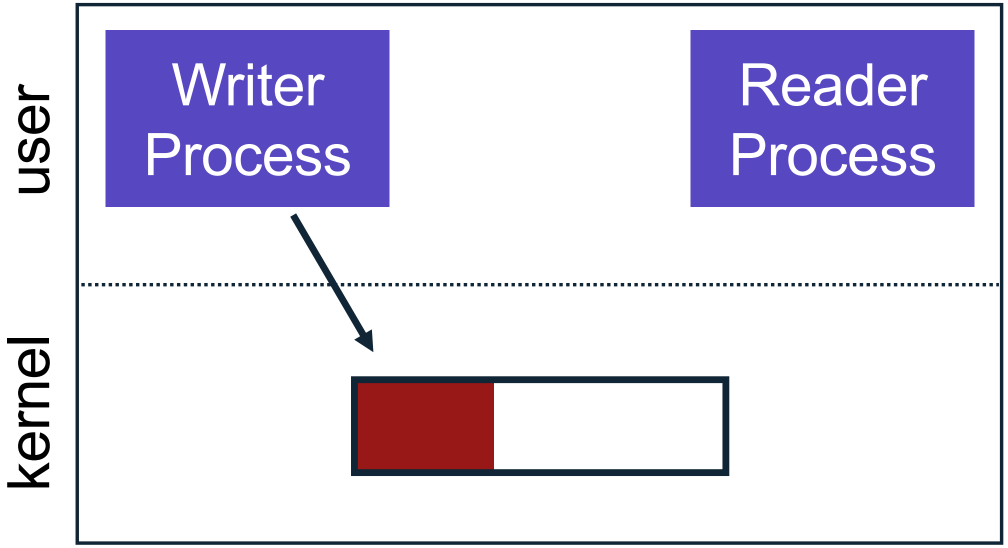
Communication failure: links unreliable

- bit errors
- packet loss
- node/link failure

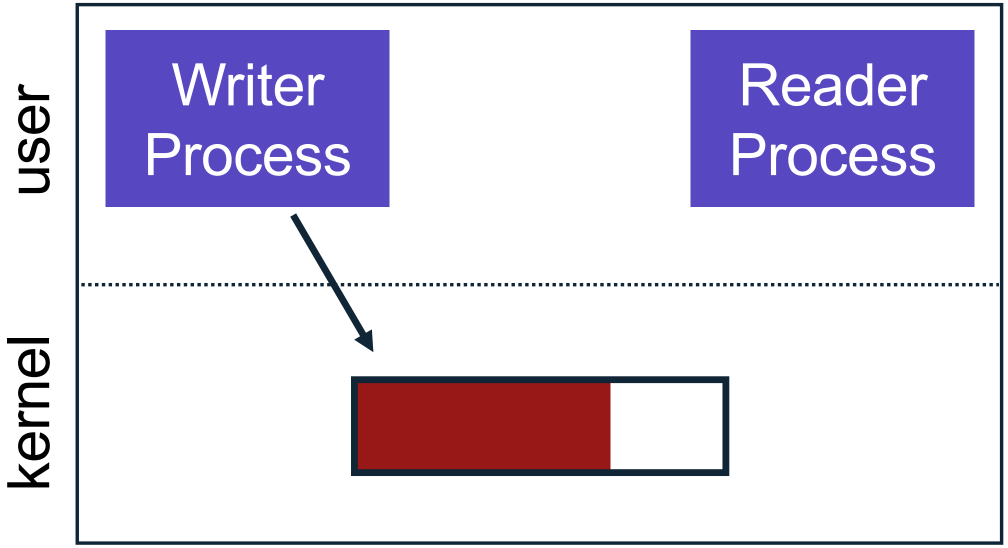
Pipe



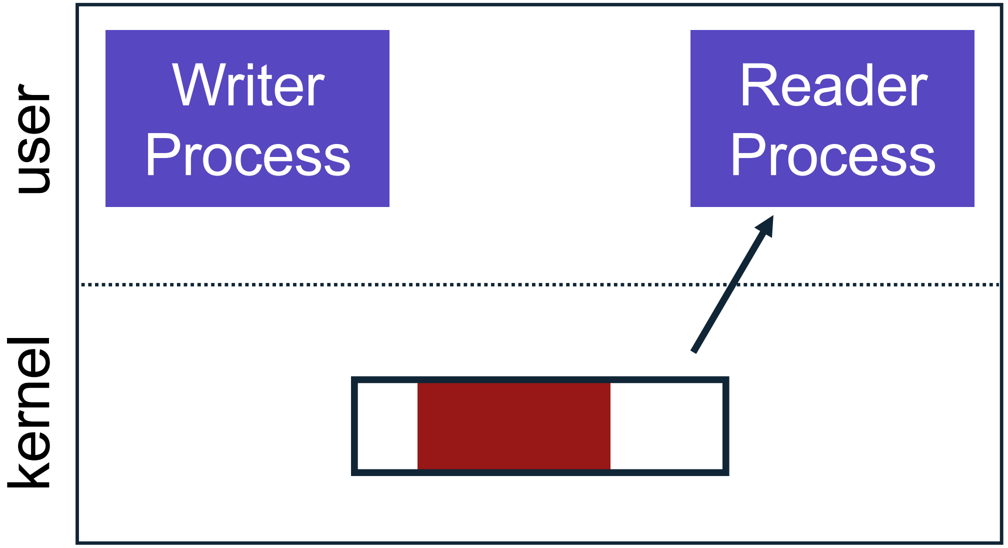
Pipe



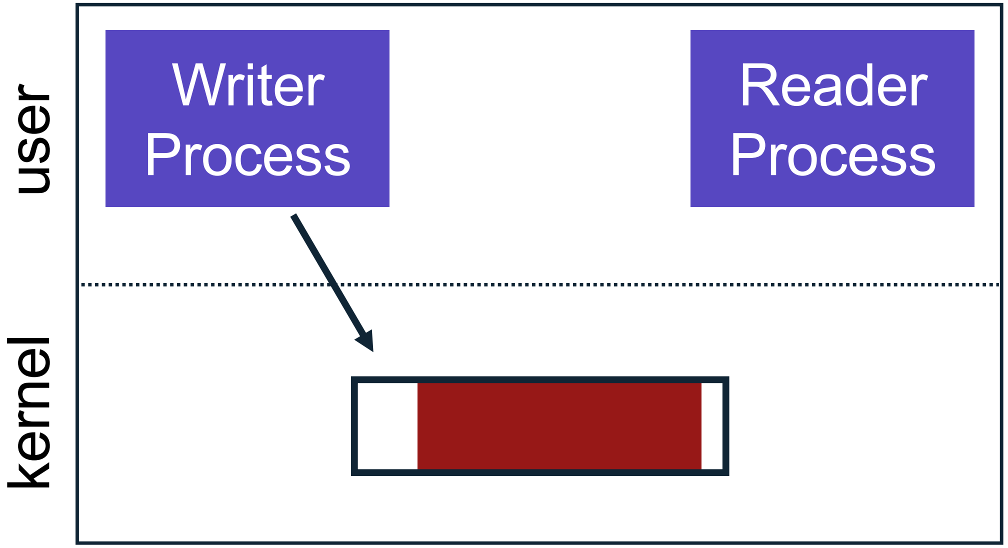
Pipe



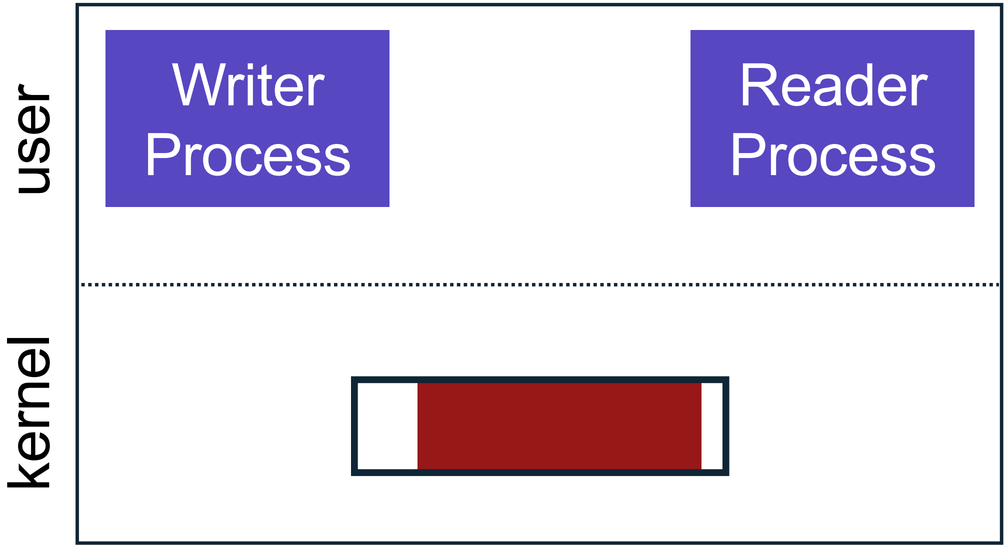
Pipe



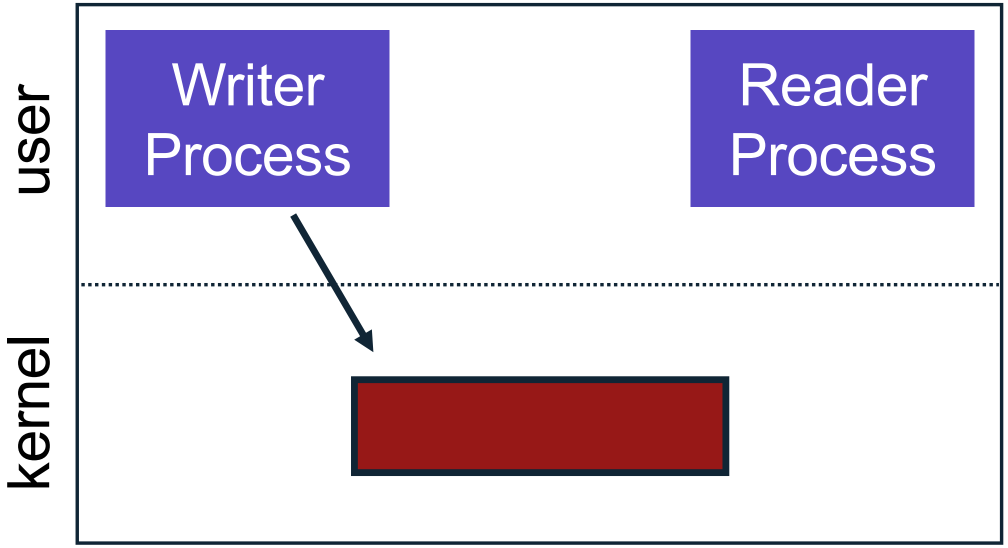
Pipe



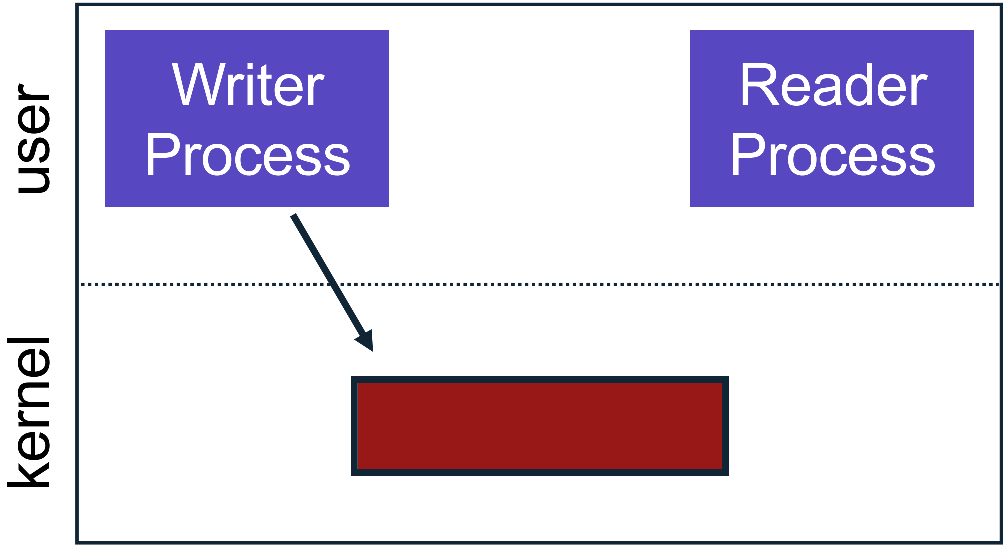
Pipe



Pipe

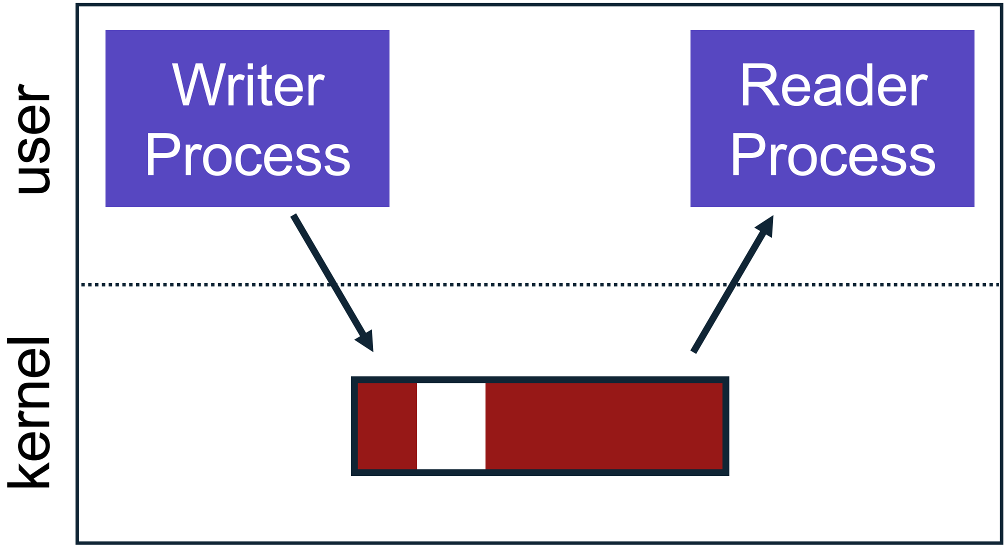


Pipe



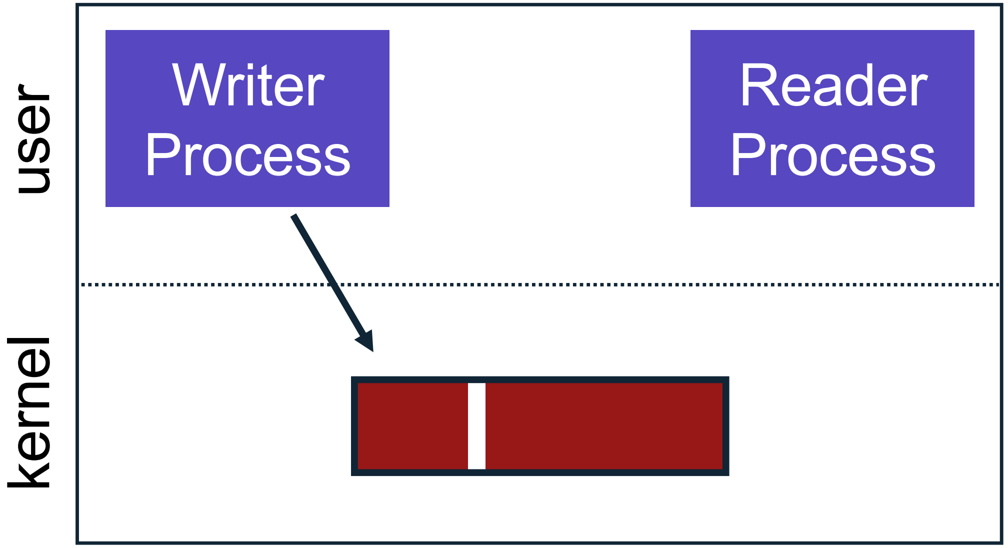
write waits for space

Pipe

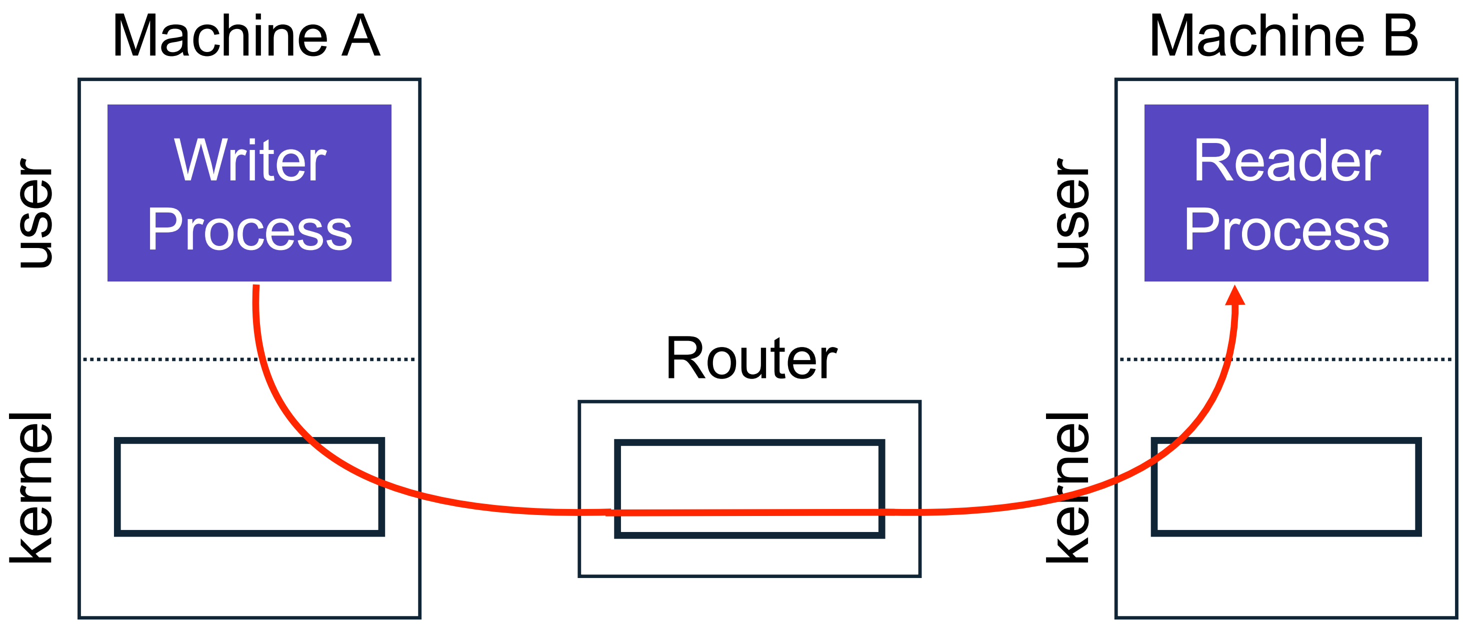


write waits for space

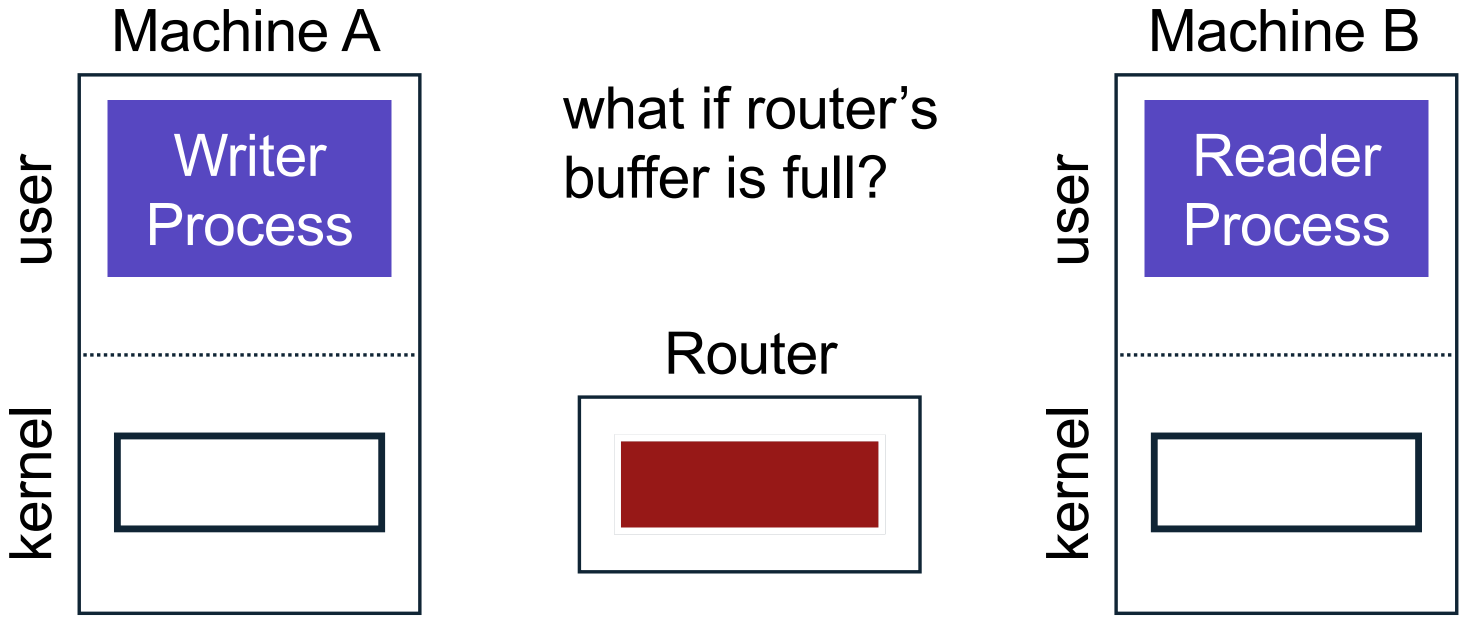
Pipe



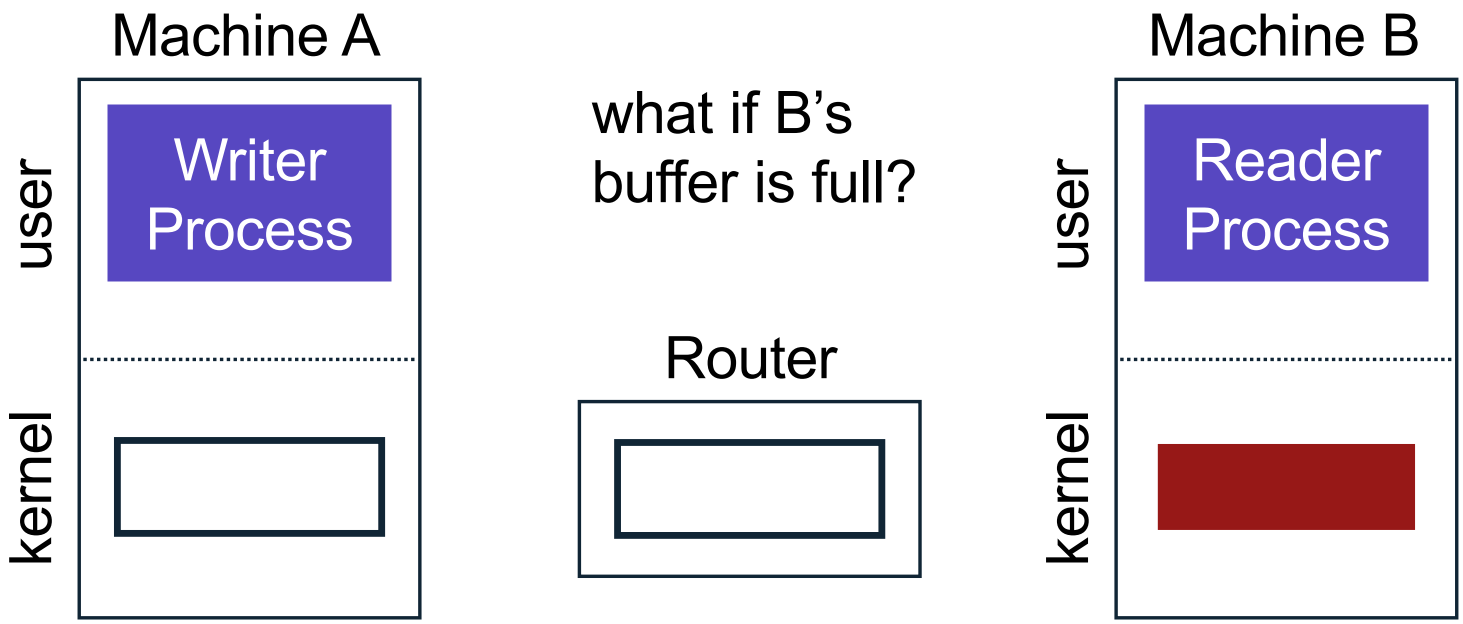
Network Socket



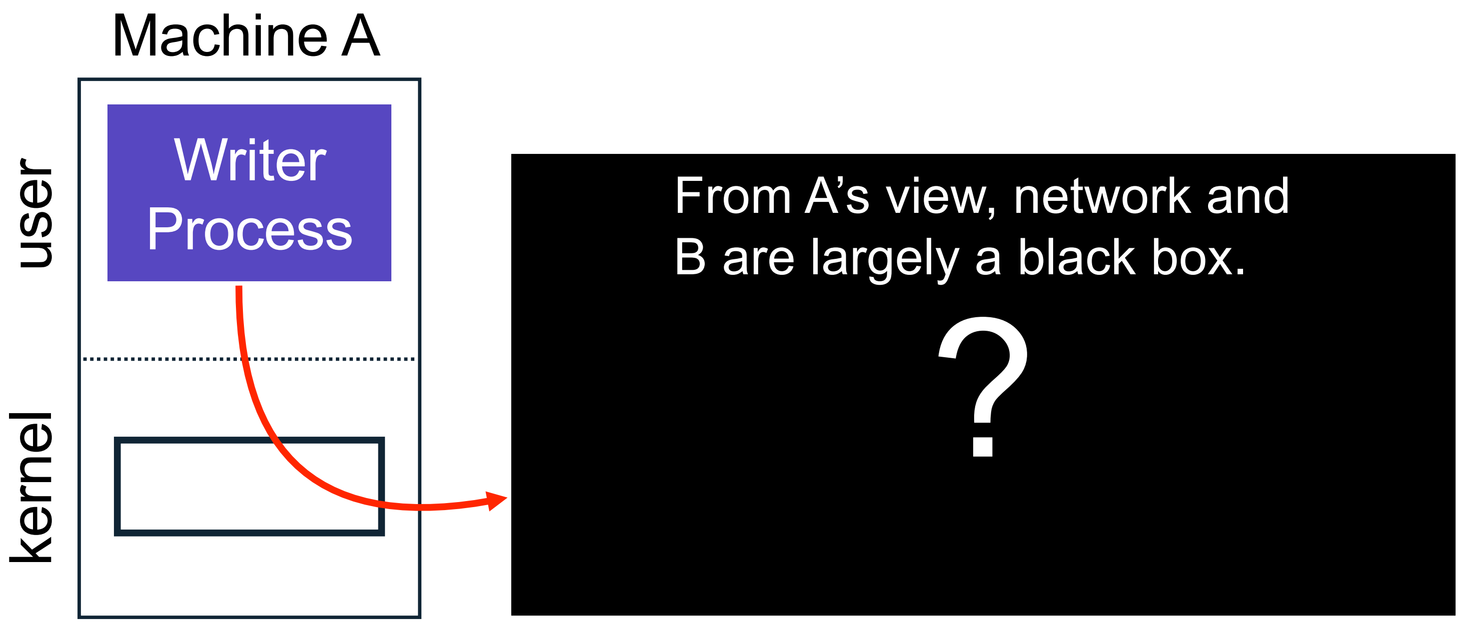
Network Socket

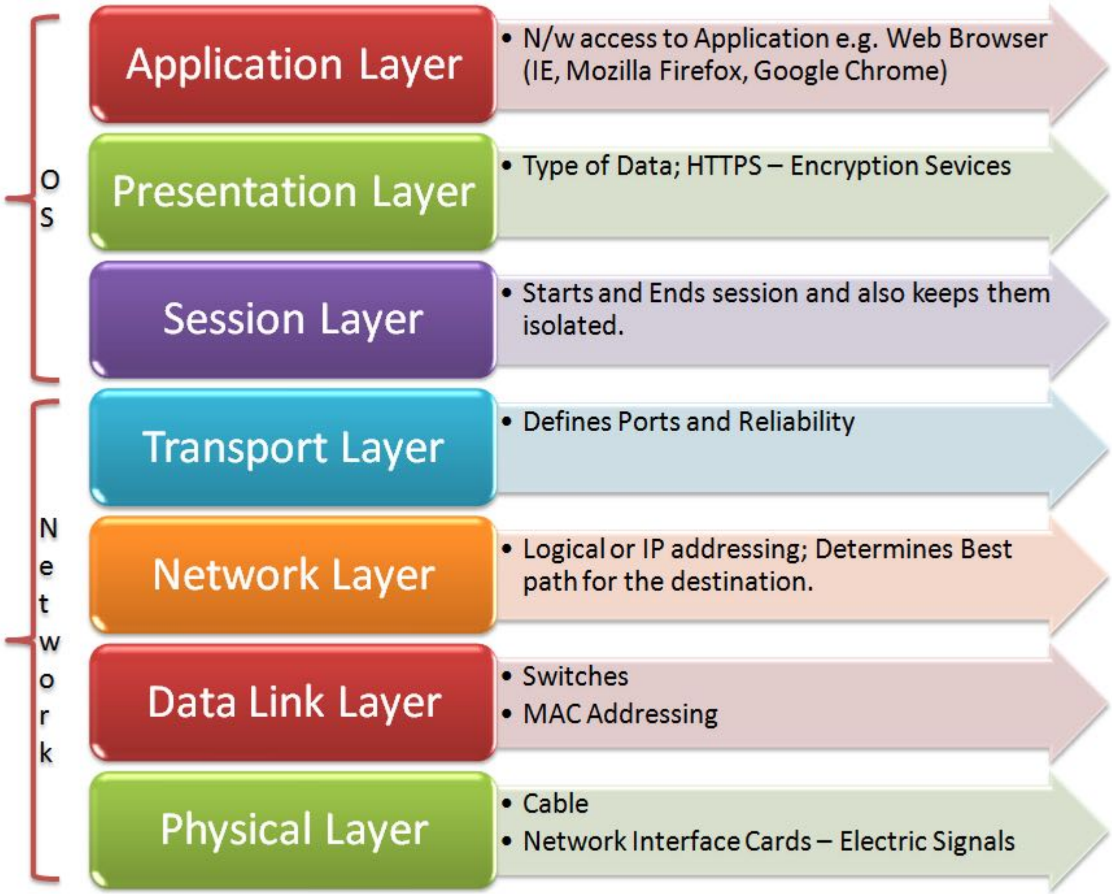


Network Socket



Network Socket



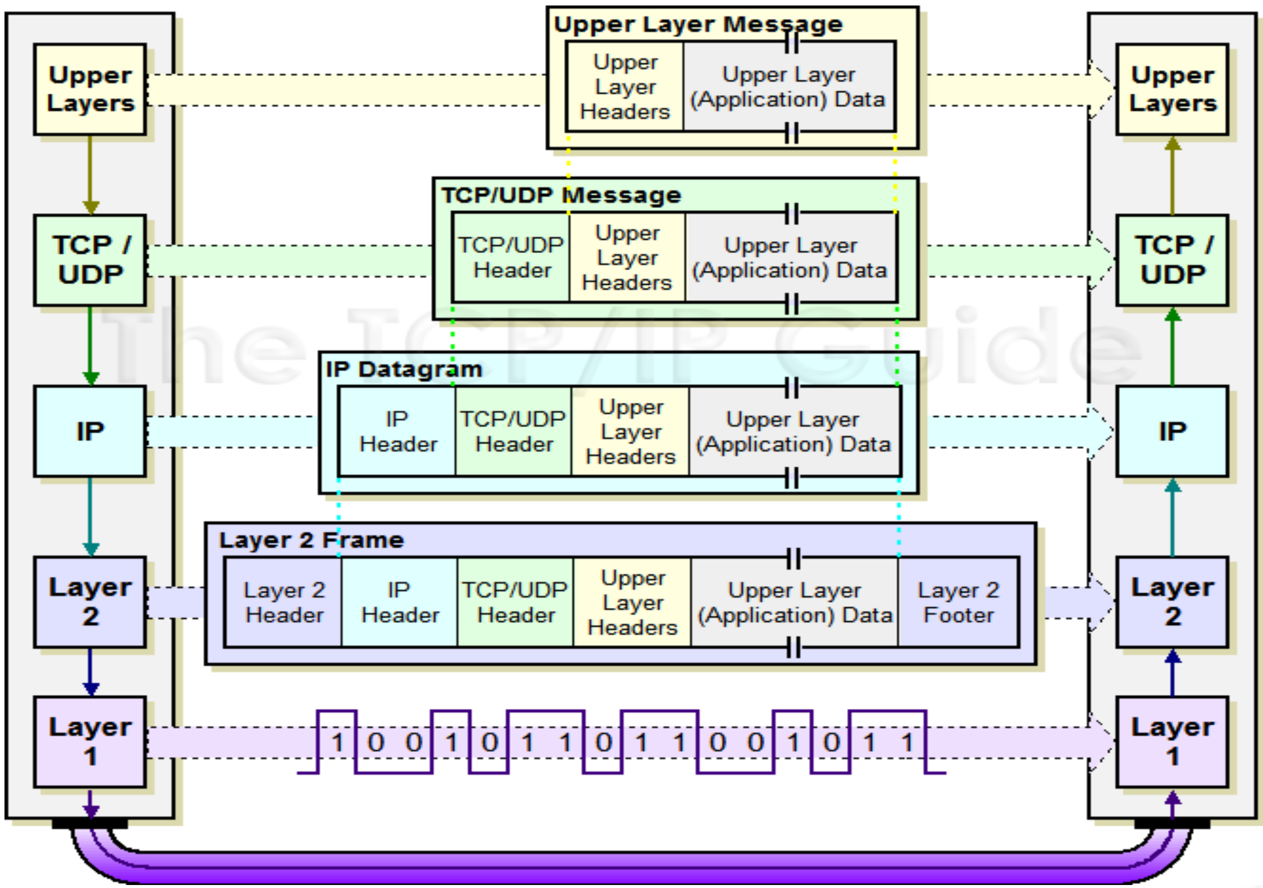


Network Architecture & Design

OSI Model Overview

THE UNIVERSITY of ADELAIDE

Network Encapsulation / De-encapsulation



http://www.tcpipguide.com

Internet Protocol (IP)

IPv4 (32-bit)

IP Address Classes				
Class	Purpose	High-Order Bits	Address Range	Maximum Number of Hosts
A	Large networks	0	1 to 126	16, 777, 214 (2^24) - 2
B	Medium networks	10	128 to 191	65534 (2^16) - 2
C	Small networks	110	192 to 223	254 (2^8) - 2
D	Multicast	1110	224 to 239	N/A
E	Experimental	1111	240 to 254	N/A

The address range 127.0.0.0 is a loopback network used for testing and troubleshooting.

CIDR Notation - 158.6.12.1/22

Internet Protocol (IP)

RFC 1918 sets aside addresses for internal use only

Addresses not routable on Internet

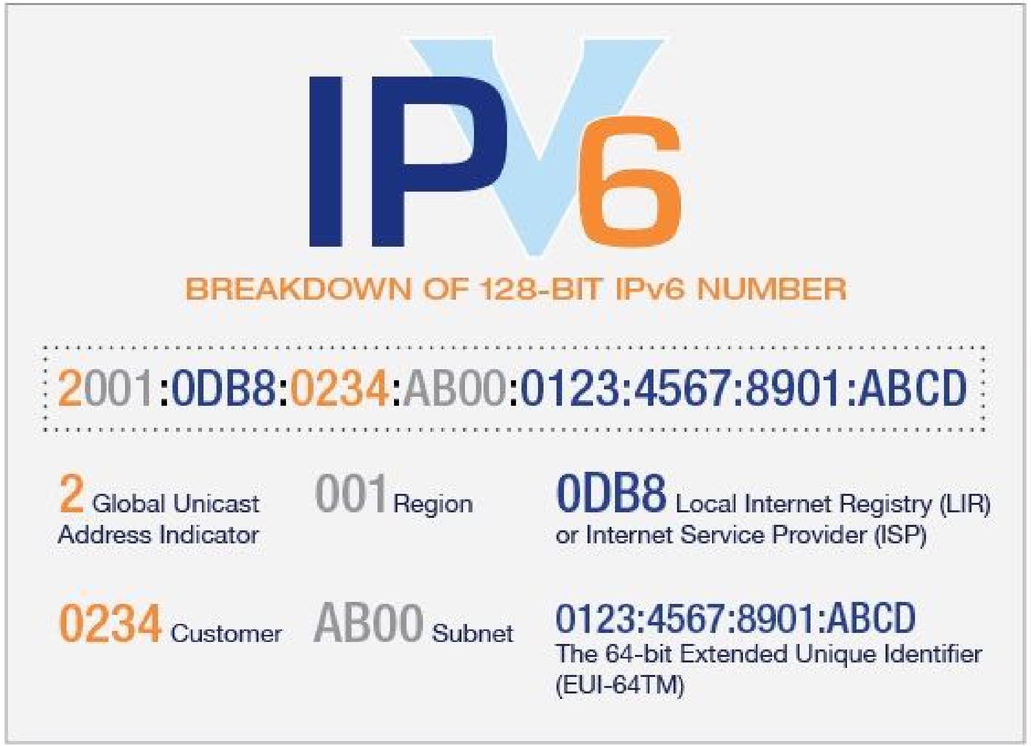
Often associated with Network Address Translation (NAT) scenarios

Class	Private Address Range
A	10.0.0.0 to 10.255.255.255
B	172.16.0.0 to 172.31.255.255
C	192.168.0.0 to 192.168.255

THE UNIVERSITY of ADELAIDE

Internet Protocol (IP)

5×10^{28} (roughly 2^{95}) addresses for each of the roughly 6.5 billion (6.5×10^{9}) people alive today



THE UNIVERSITY *of* ADELAIDE

Communication Overview

Raw messages: UDP

Reliable messages: TCP

Remote procedure call: RPC

User Datagram Protocol (UDP)

Provides “best effort” delivery

Connectionless

- No error detection or correction
- Do not use sequencing
- No flow control mechanisms,
- Do not use a pre-established session
- Considered unreliable

Has little overhead (and no flow control), so it is faster than TCP

- Streaming data

Header

- Source port / Destination port
- Message length / Checksum

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

Communication Overview

~~Raw messages: UDP~~

Reliable messages: TCP

Remote procedure call: RPC

Reliable Messages: Layering strategy

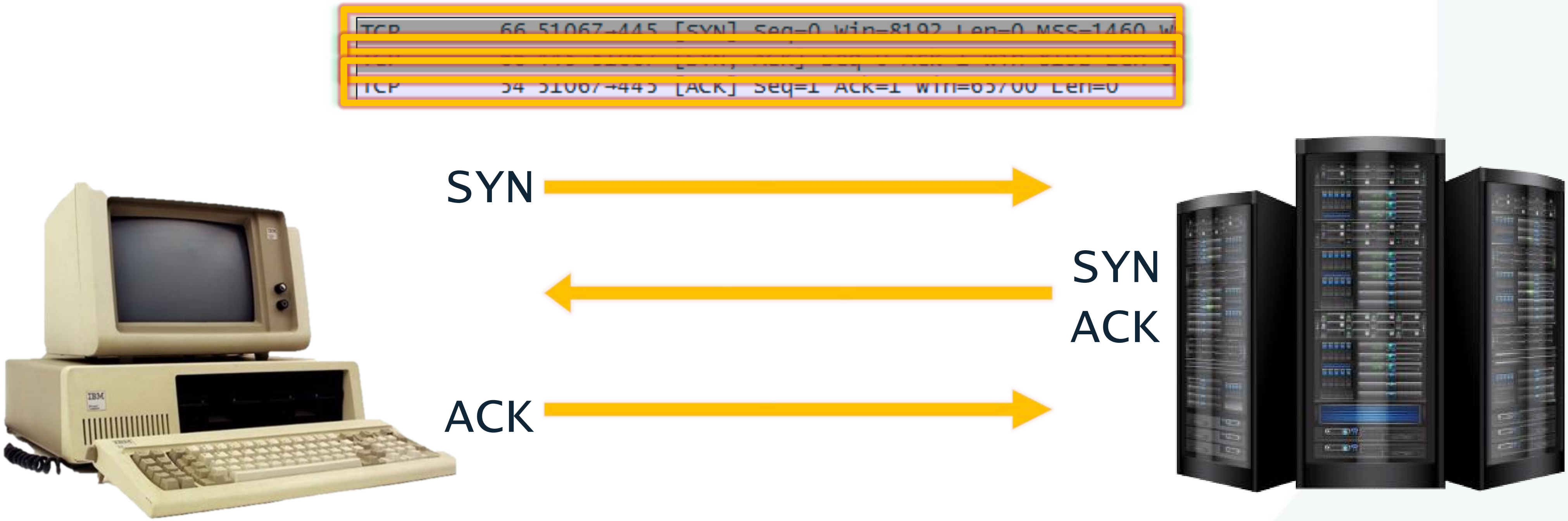
TCP: Transmission Control Protocol

Using software, build reliable, logical connections over unreliable connections

Techniques:

- acknowledgment (ACK)

TCP 3-way Handshake



Technique #1: ACK

Sender	Receiver
[send message]	
	[recv message]
	[send ack]
[recv ack]	

Sender knows message was received

ACK

Sender

[send message]

Receiver

Sender doesn't receive ACK…
What to do?

Technique #2: Timeout

Sender	Receiver
[send message] [start timer]	
… waiting for ack …	
[timer goes off] [send message]	
	[recv message] [send ack]
[recv ack]	

Lost ACK: Issue 1

How long to wait?

Too long?

System feels unresponsive

Too short?

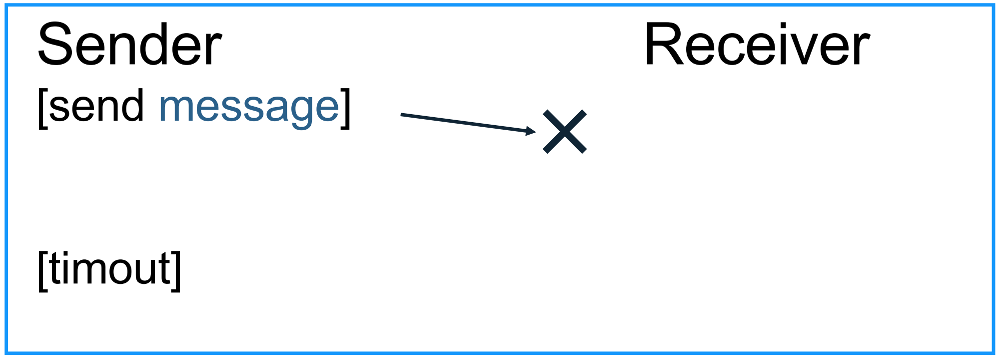
Messages needlessly re-sent

Messages may have been dropped due to overloaded server. Resending makes overload worse!

Lost Ack: Issue 2

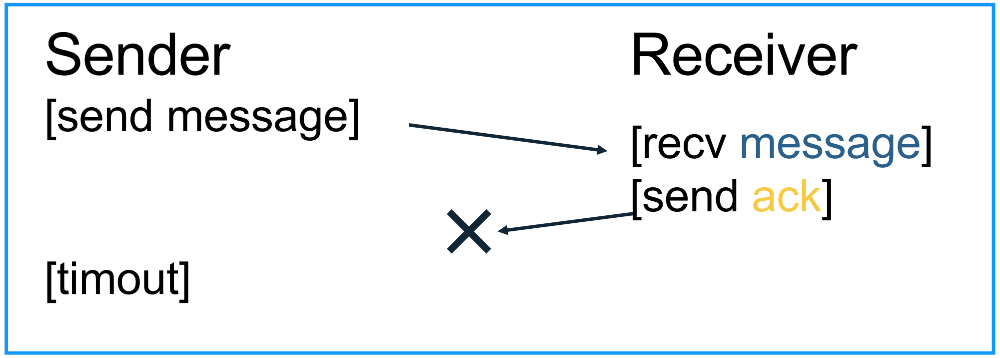
What does a lost ack really mean?

Case 1



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

Case 2



ACK: message received exactly once

No ACK: message may or may not have been received

What if message is command to increment counter?

THE UNIVERSITY of ADELAIDE

Reliable Messages: Layering Strategy

Using software, build reliable, logical connections over unreliable connections

Techniques:

- acknowledgment
- timeout
- remember sent messages

Technique #3: Receiver Remembers Messages

Sender | Receiver
[send message] → [recv message]
[send ack] ✕

[timout]
[send message] → [ignore message]
[send ack]
[recv ack]

how does receiver know to ignore?

THE UNIVERSITY of ADELAIDE

Sequence numbers

Sequence numbers

- senders gives each message an increasing unique seq 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
- Msg K is already buffered

THE UNIVERSITY of ADELAIDE

TCP

TCP: Transmission Control Protocol

Most popular protocol based on seq nums

Buffers messages so arrive in order

Timeouts are adaptive

Communications Overview

~~Raw messages: UDP~~

~~Reliable messages: TCP~~

Remote procedure call: RPC

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

Very common abstraction

RPC

Machine A

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

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

Machine B

```
int foo(char*msg) {
    …
}

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

What it feels like for programmer

RPC

Machine A

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

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

Machine B

```
int foo(char*msg) {
    …
}

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

Actual Calls

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

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: Pointers

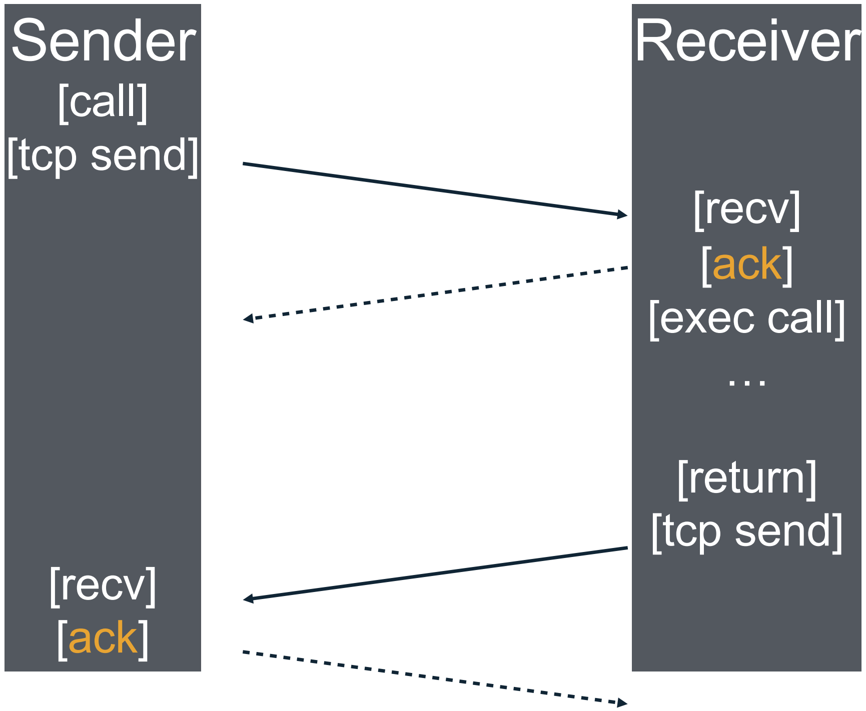
Why are pointers problematic?

The address passed from client not valid on server

Solutions?

- smart RPC package: follow pointers and copy data

RPC over UDP



Why wasteful?

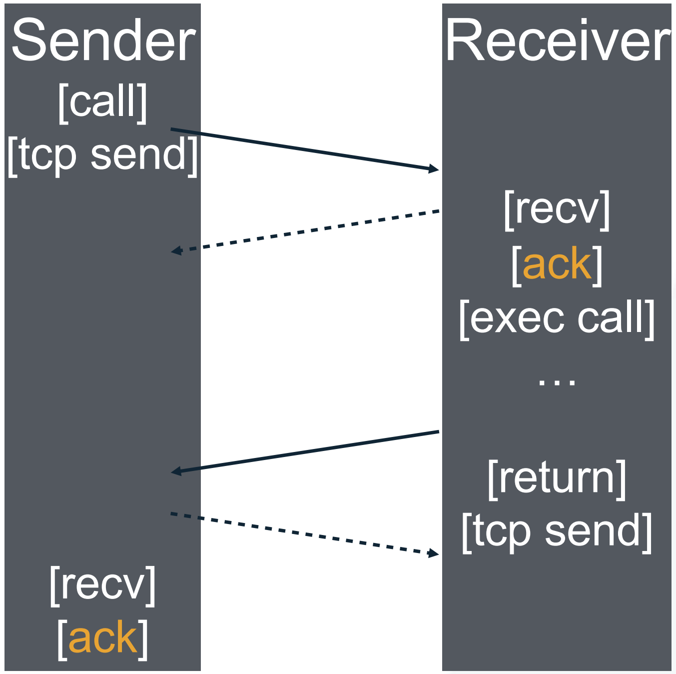
RPC over UDP

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

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

COMP SCI 3004 Operating Systems

NFS & AFS

make history.

THE UNIVERSITY of ADELAIDE

Overview

Architecture

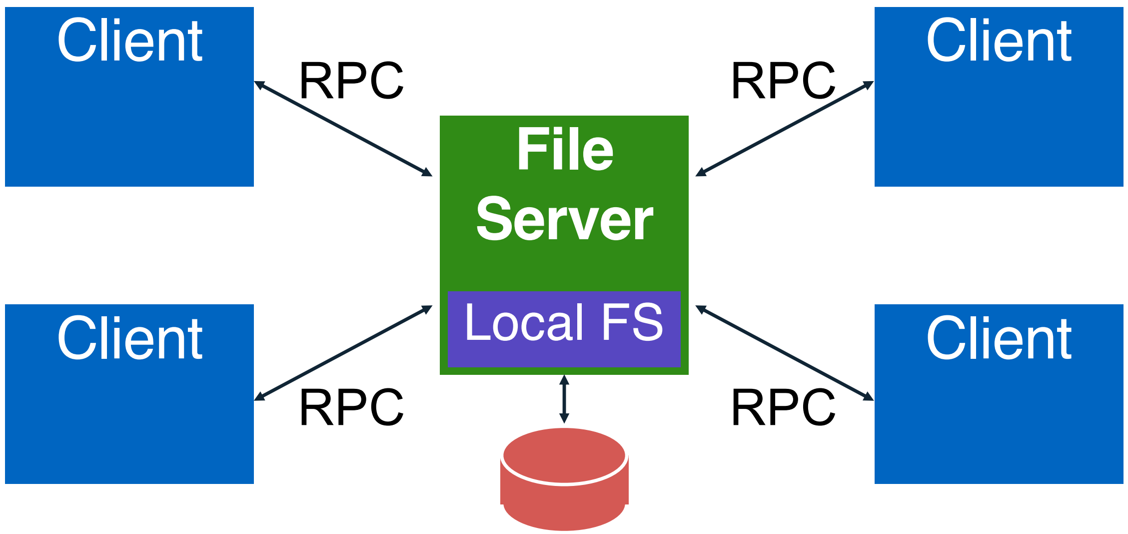
Network API

Write Buffering

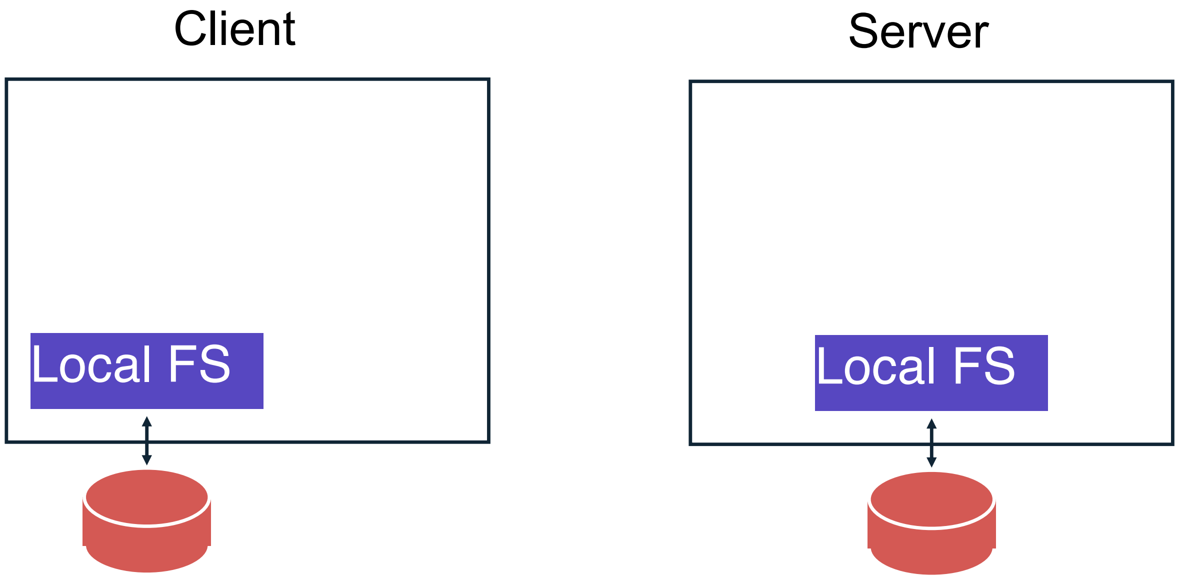
Cache

THE UNIVERSITY of ADELAIDE

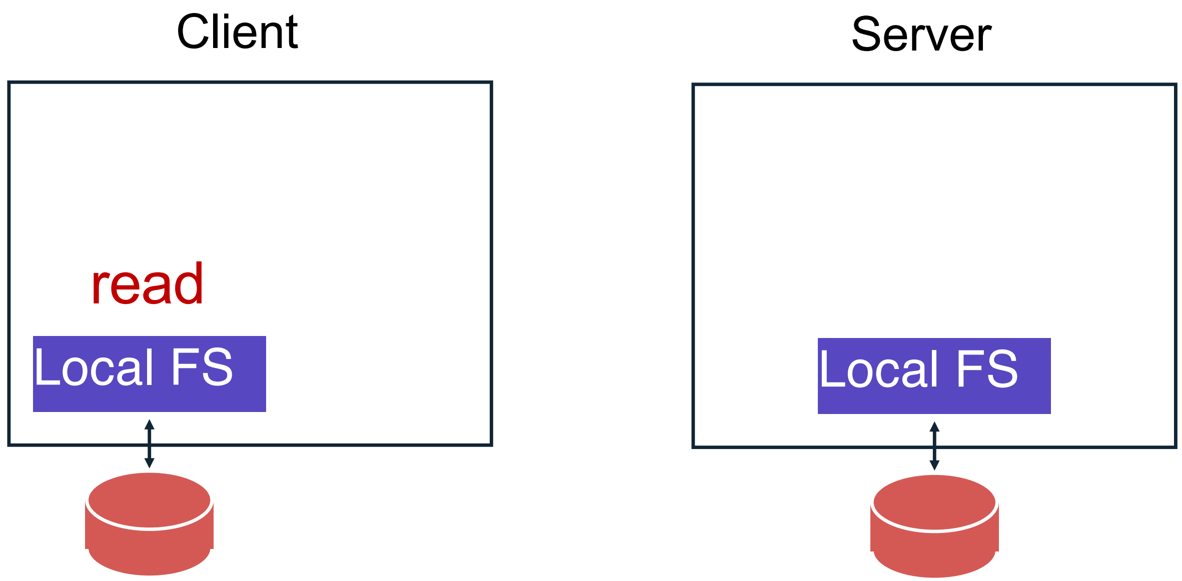
NFS Architecture



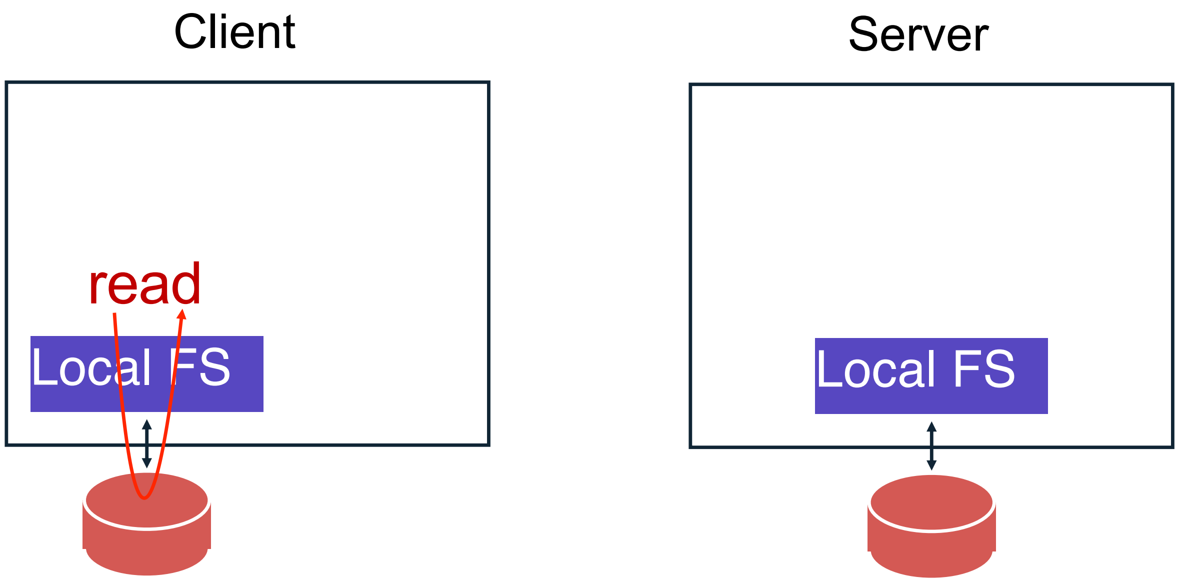
General Strategy: Export FS



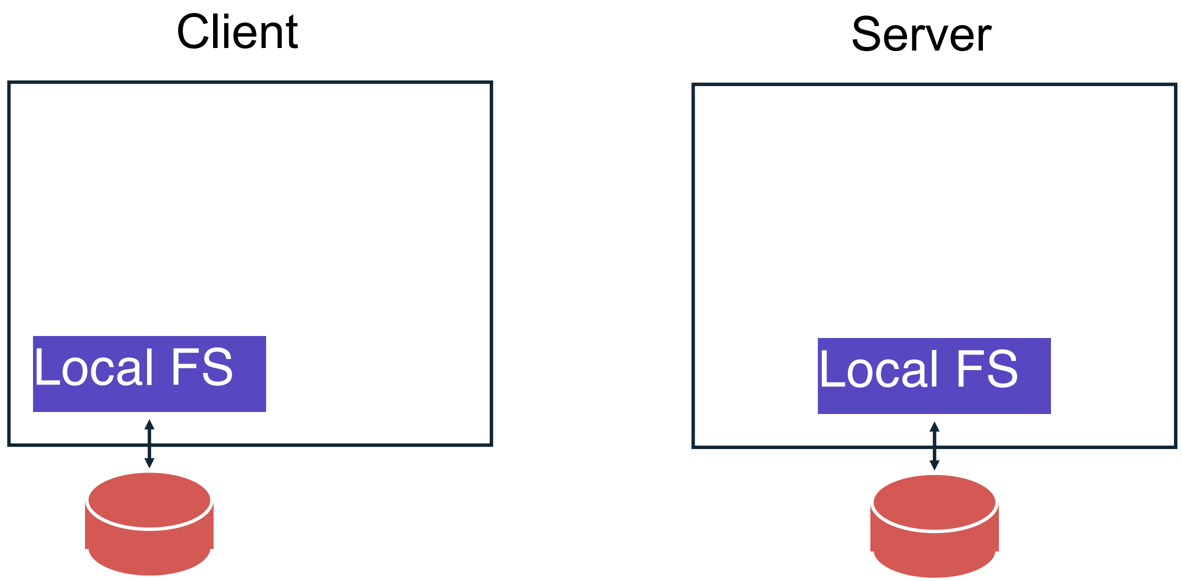
General Strategy: Export FS



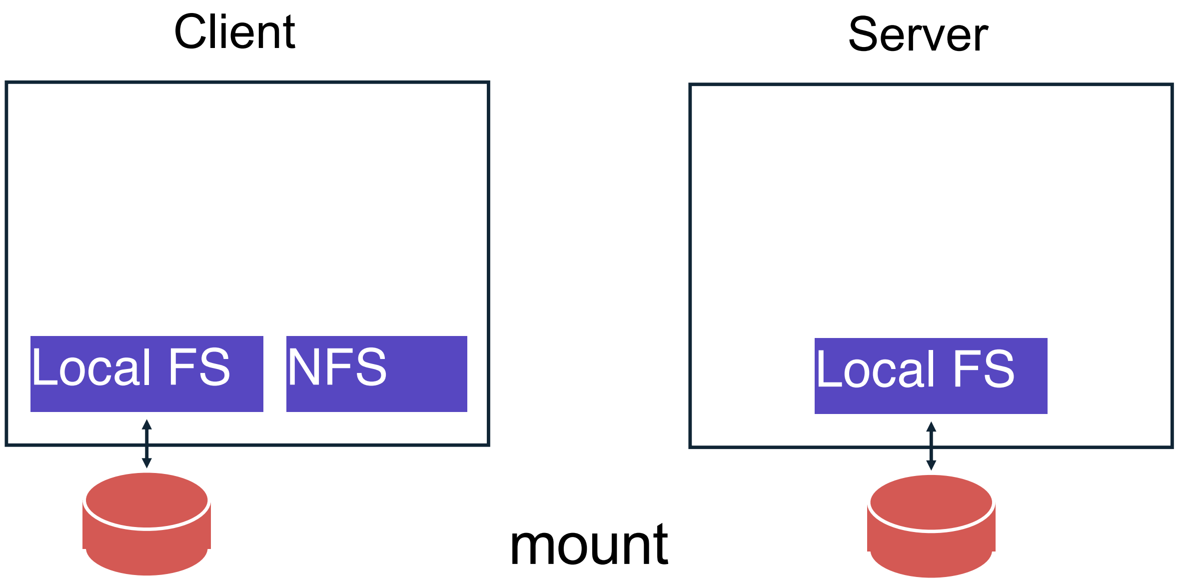
General Strategy: Export FS

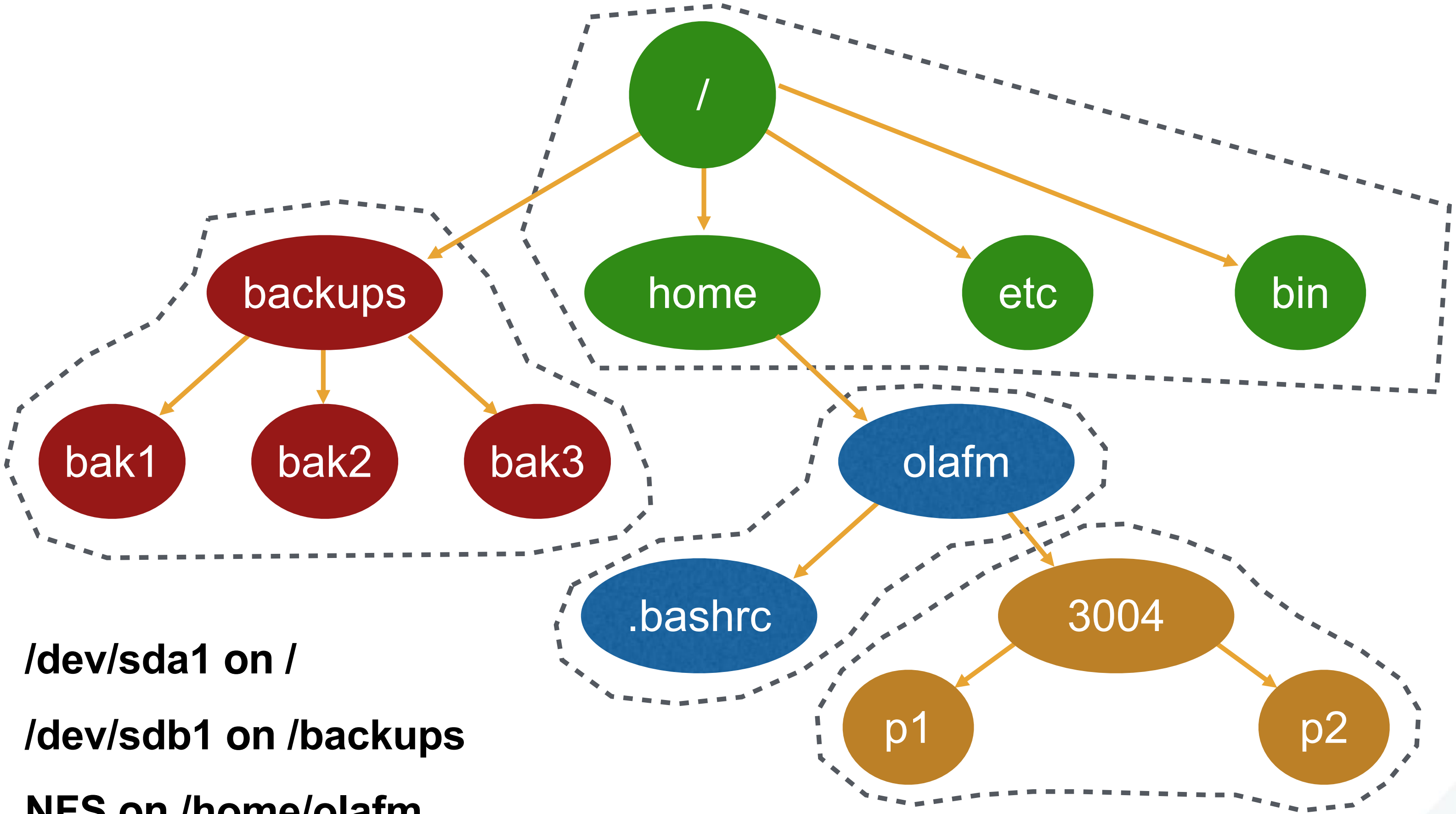


General Strategy: Export FS



General Strategy: Export FS



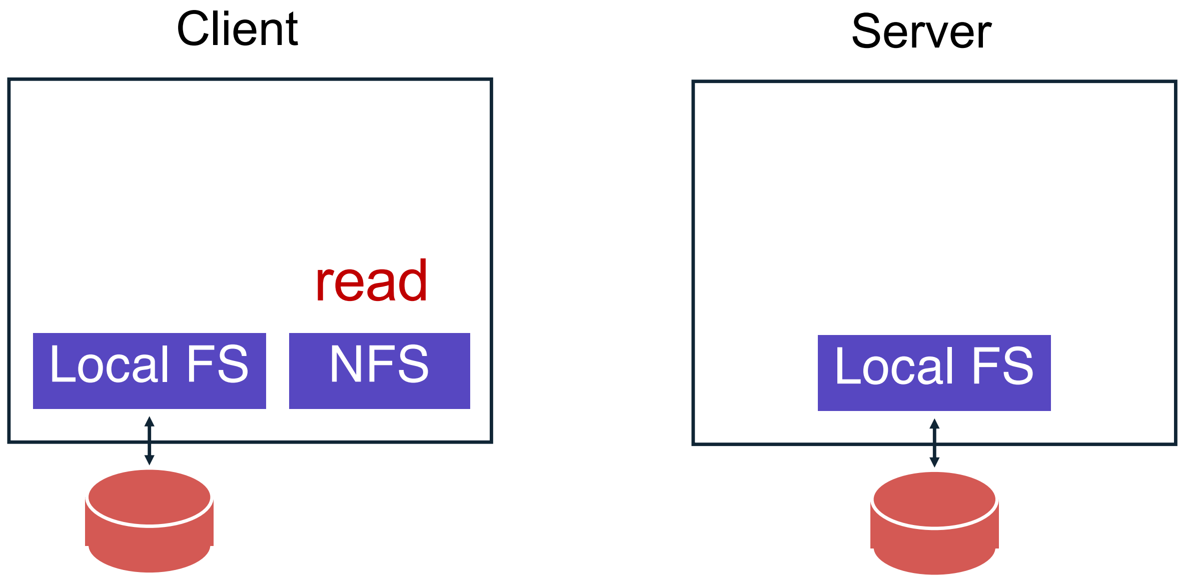


/dev/sda1 on /

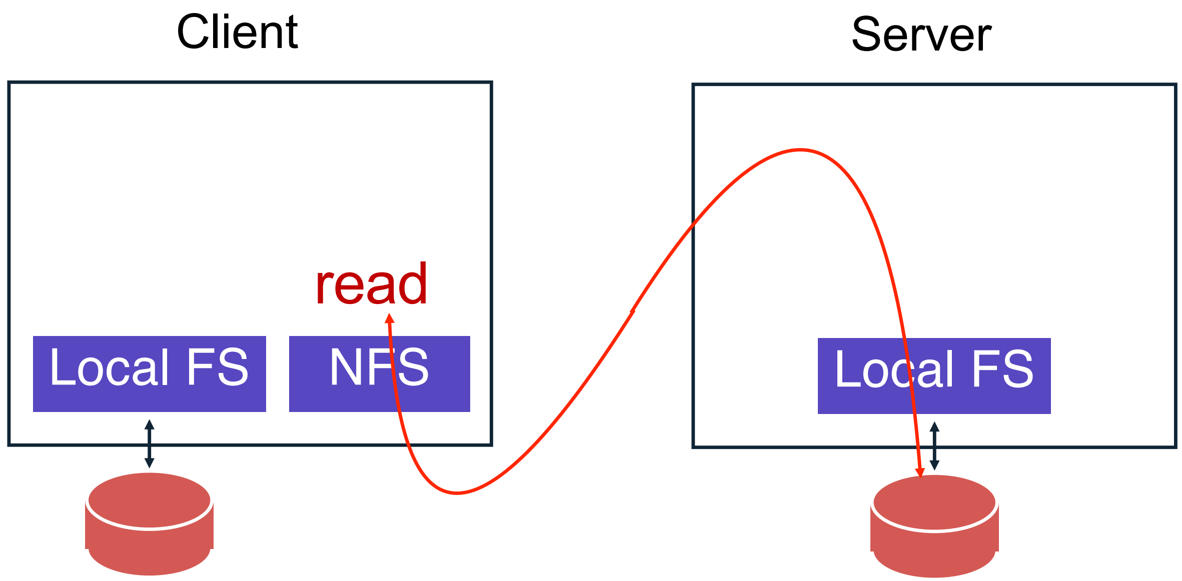
/dev/sdb1 on /backups

NFS on /home/olafm

General Strategy: Export FS



General Strategy: Export FS



Strategy 1 Problems

What about crashes?

```
int fd = open("foo", O_RDONLY);
read(fd, buf, MAX);
read(fd, buf, MAX);
…
read(fd, buf, MAX);
```

Server crash!
nice if acts like a "slow read"

Imagine server crashes and reboots during reads…

THE UNIVERSITY of ADELAIDE

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

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 each time. Server need not remember anything from clients.

Pros? Server can crash and reboot transparently to clients.

Cons? Too many path lookups.

THE UNIVERSITY of ADELAIDE

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);
```

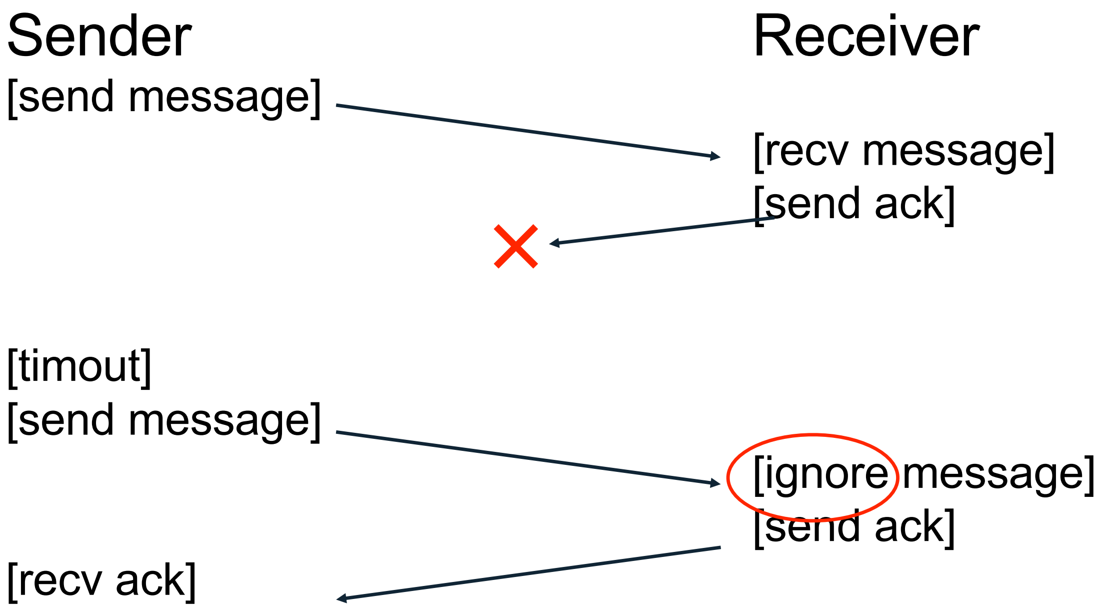
Problem with append()?

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

Problem: Why is it difficult to not replay append()?

THE UNIVERSITY of ADELAIDE

Replica Suppression is Stateful



TCP suppresses repeated message

Problem: TCP is stateful
If server crashes, forgets which RPC’s have been executed!

Idempotent Operations

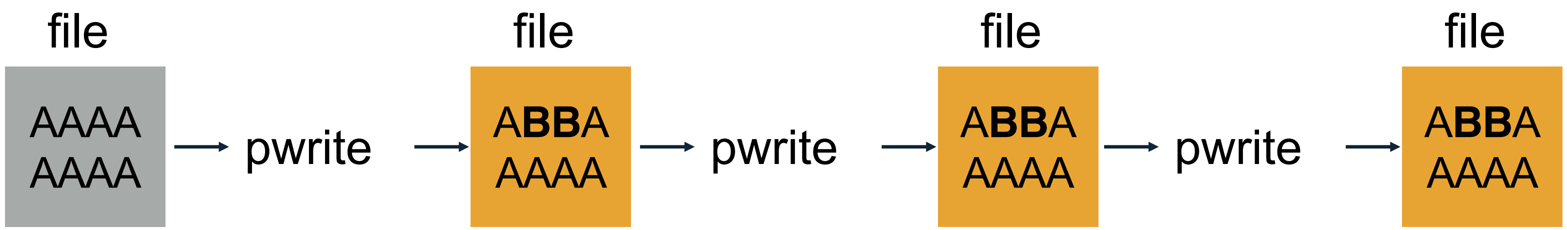
Solution:

Design API so no harm to executing function more than once

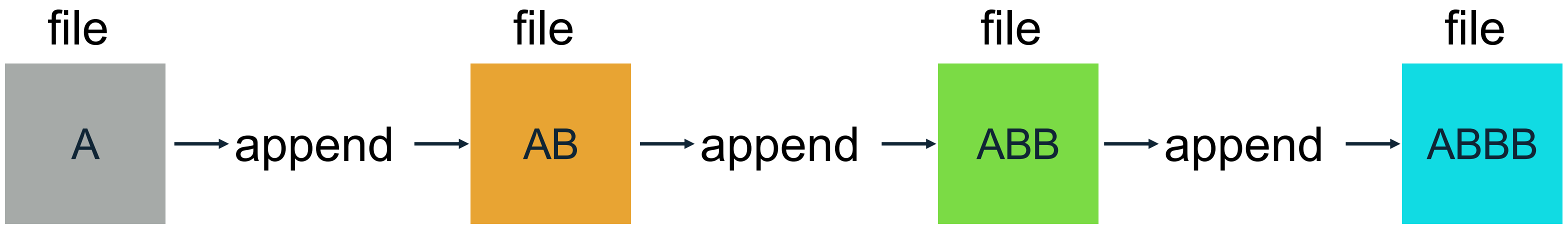
If f() is idempotent, then:

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

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 #>
```

THE UNIVERSITY *of* ADELAIDE

Strategy 5: client logic

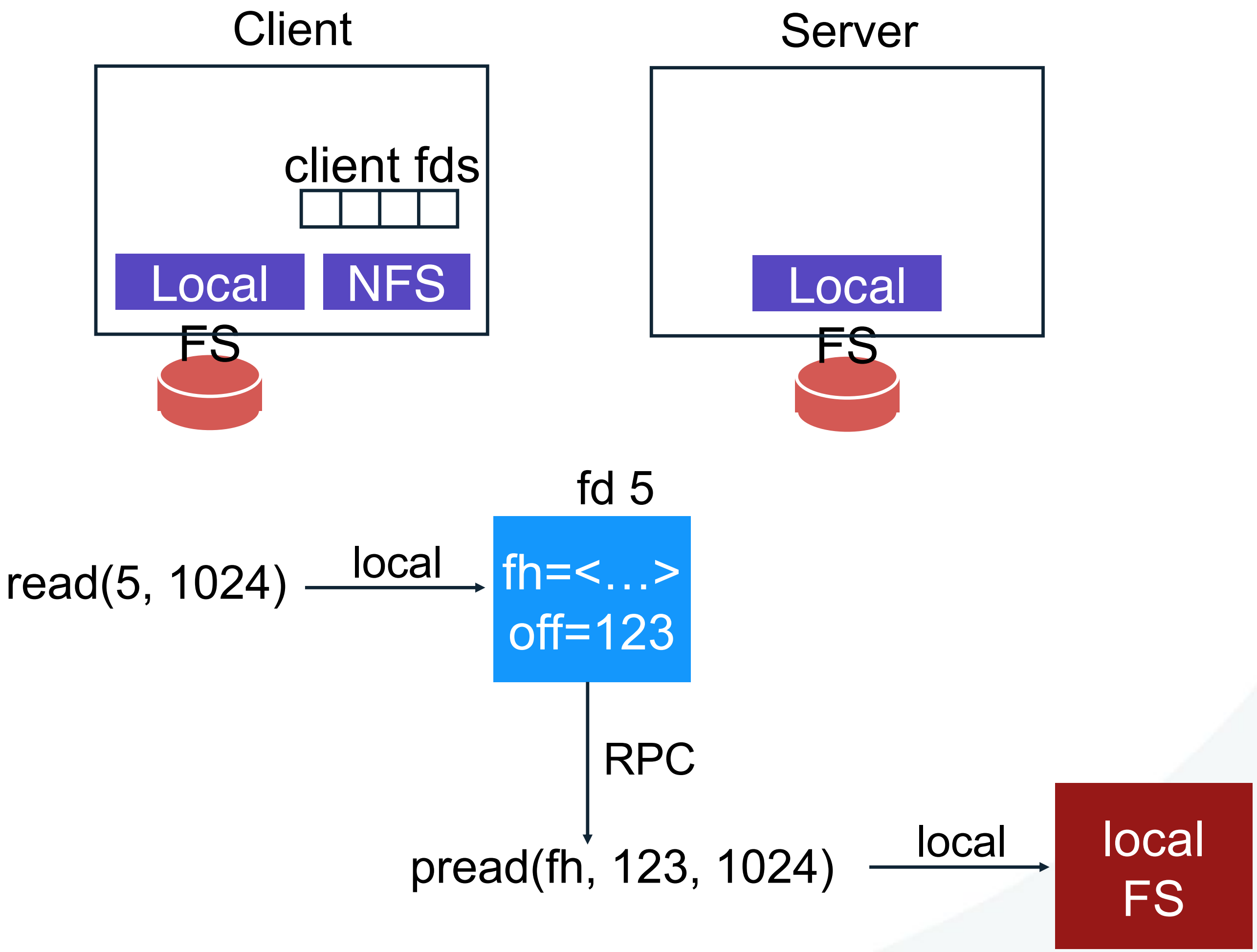
Build normal UNIX API on client side on top of idempotent, RPC-based API

Client open() creates a local fd object

It contains:

- file handle
- offset

File Descriptors



Overview

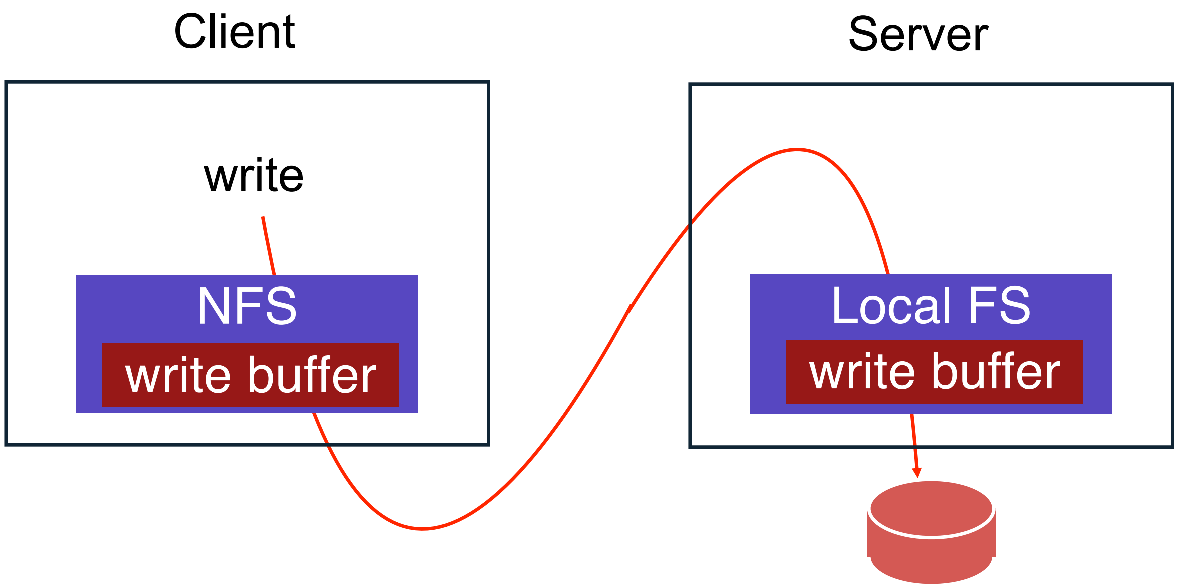
~~Architecture~~

~~Network API~~

Write Buffering

Cache

Write Buffers



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

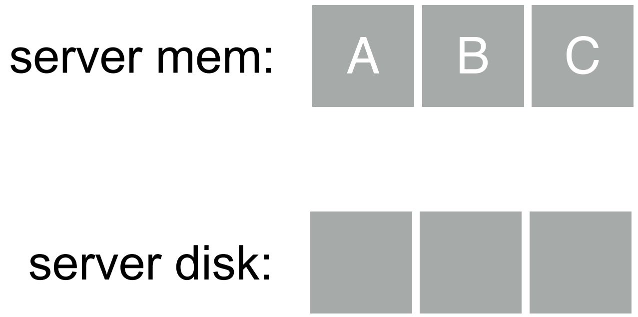
Server Write Buffer Lost

```
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

THE UNIVERSITY of ADELAIDE

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

THE UNIVERSITY of ADELAIDE

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: | X | B | C |

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: X B C

server acknowledges write before write is pushed to disk

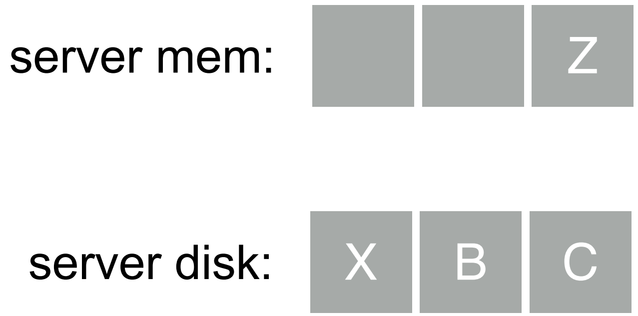
THE UNIVERSITY of ADELAIDE

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 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 |

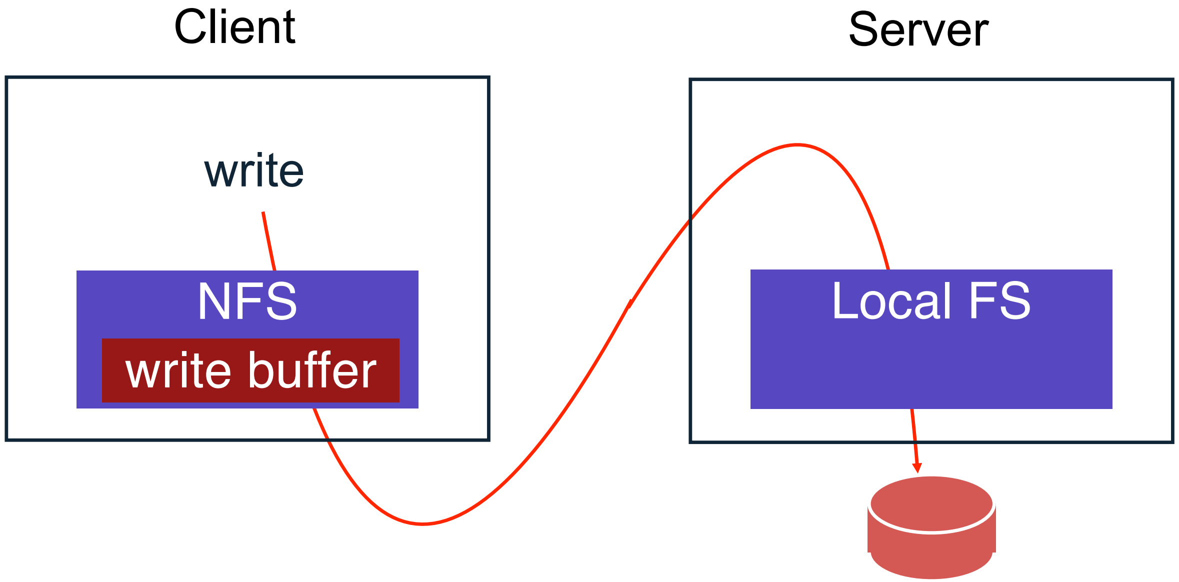
server disk: | X | B | Z |

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

Solutions????

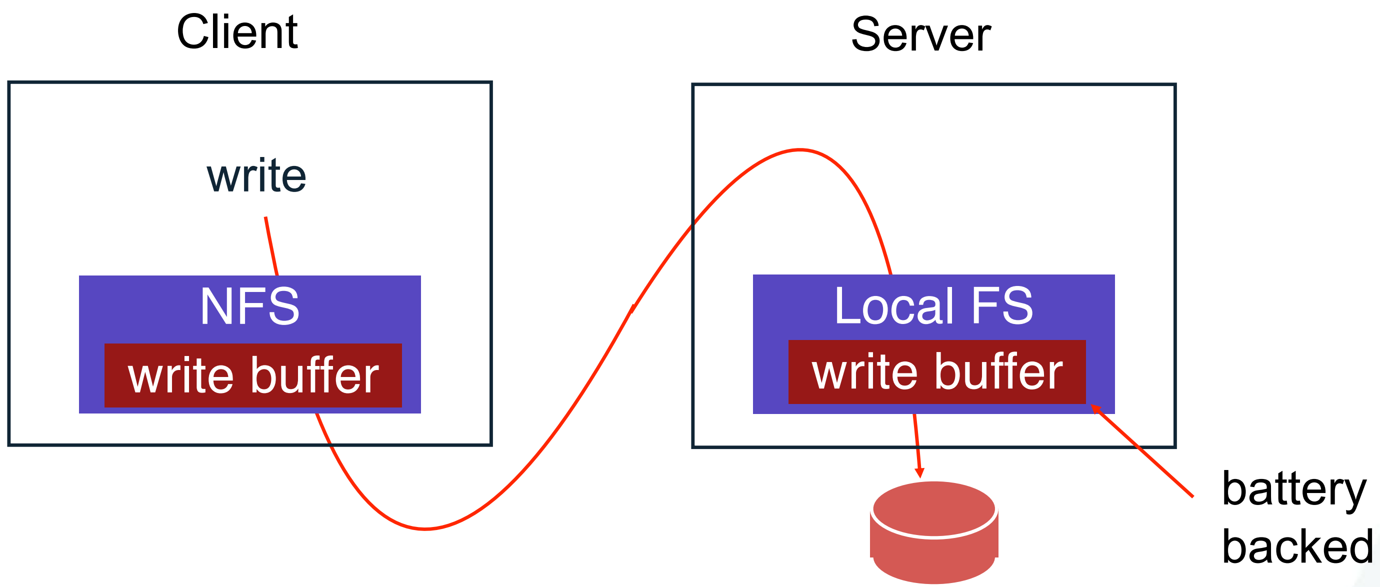
THE UNIVERSITY *of* ADELAIDE

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

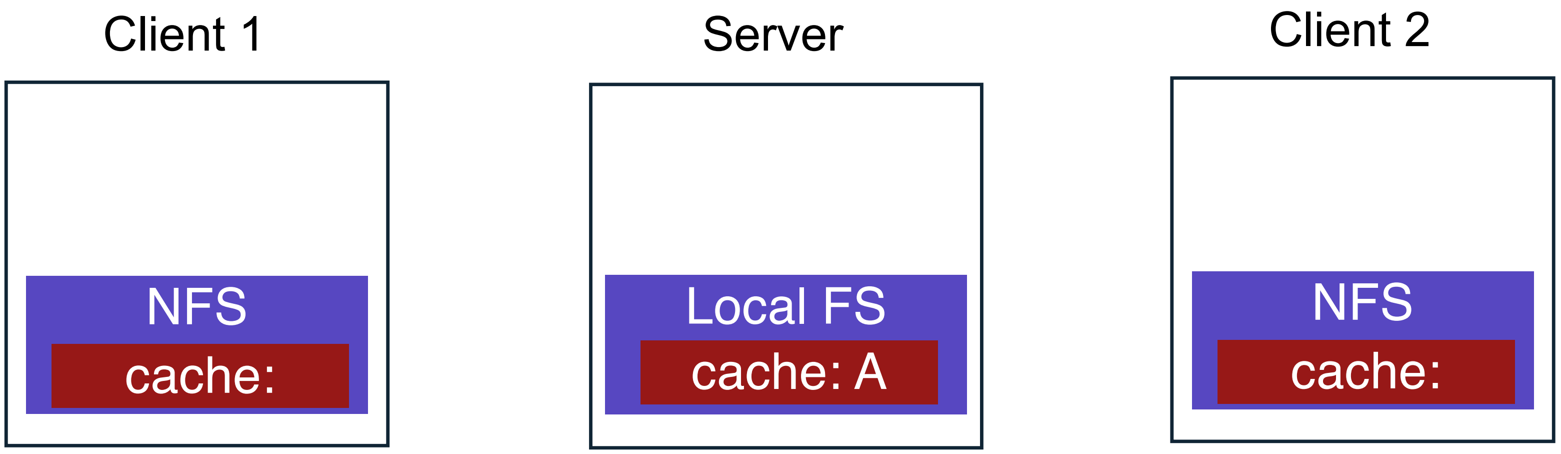
Cache Consistency

NFS can cache data in three places:

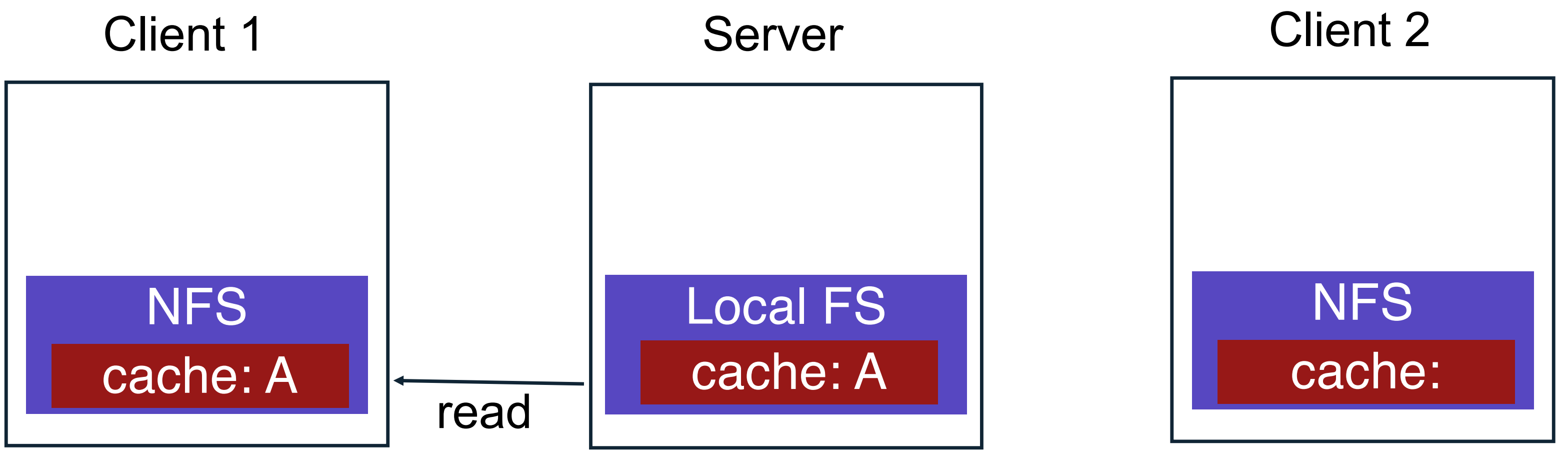
- server memory
- client disk
- client memory

How do you make sure all versions are in sync?

Distributed Cache

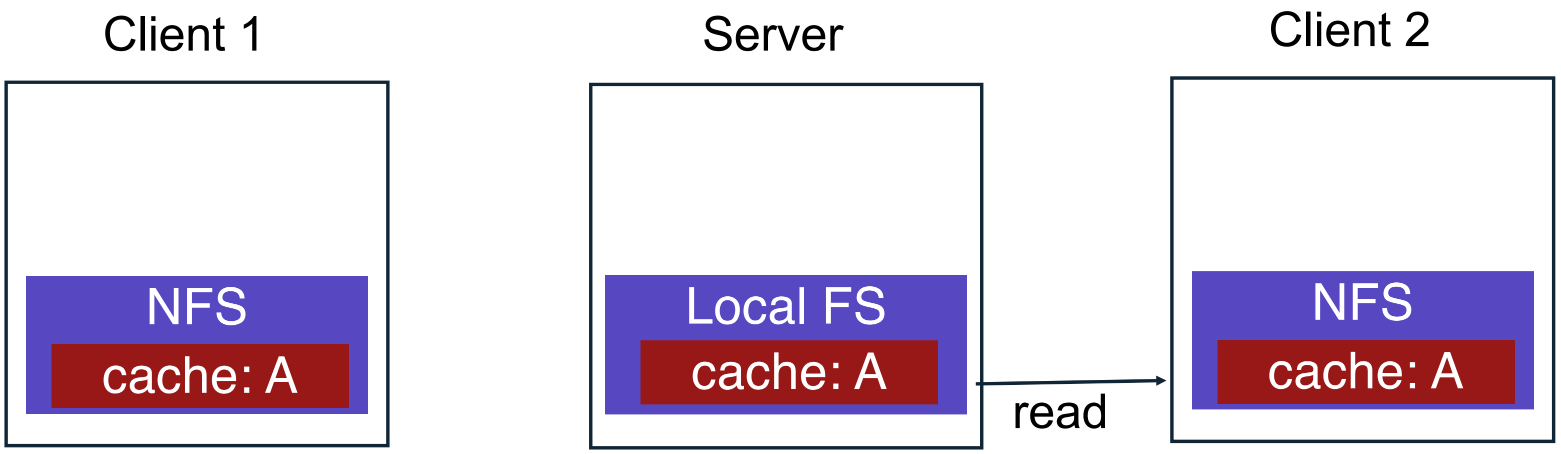


Cache



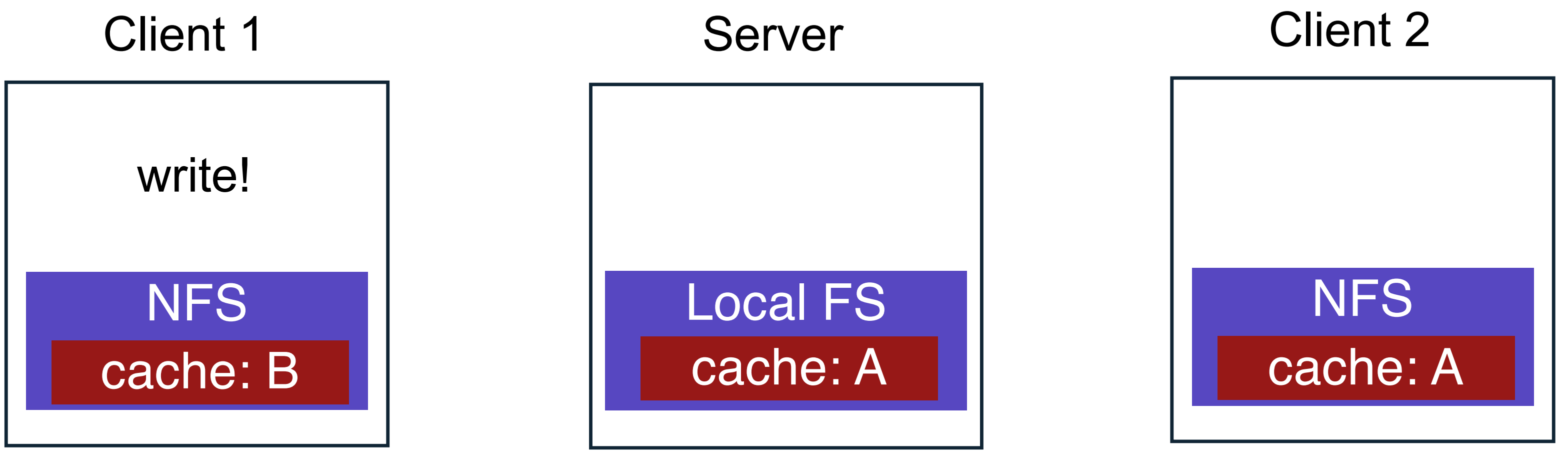
Client 1
Server
Client 2
NFS
cache: A
read
Local FS
cache: A
NFS
cache:

Cache



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

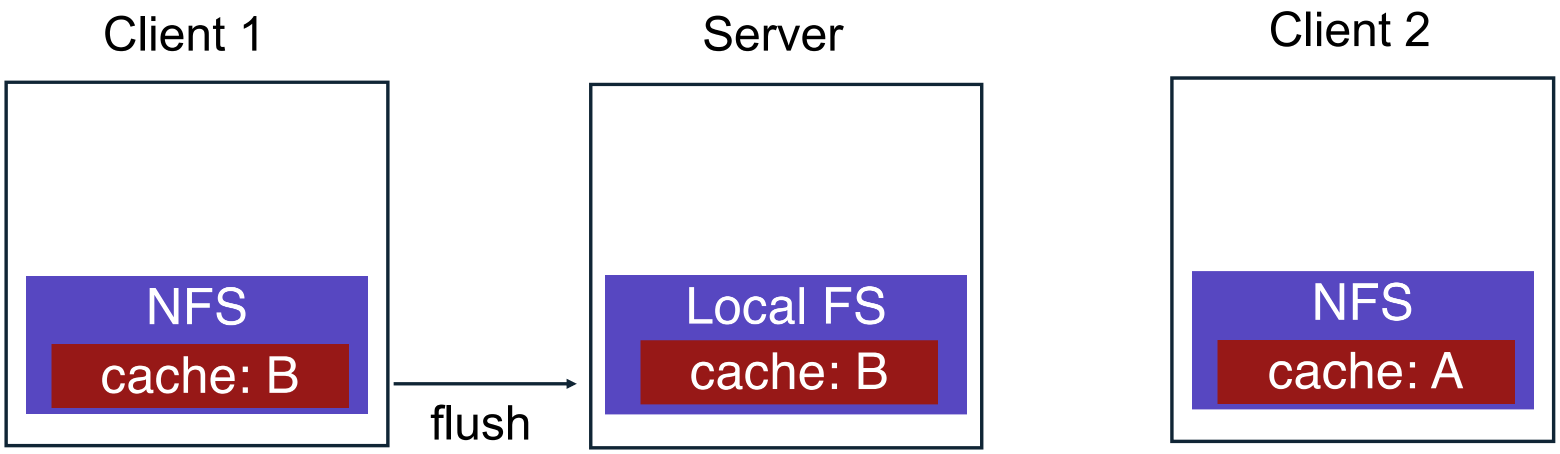
Cache



“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)

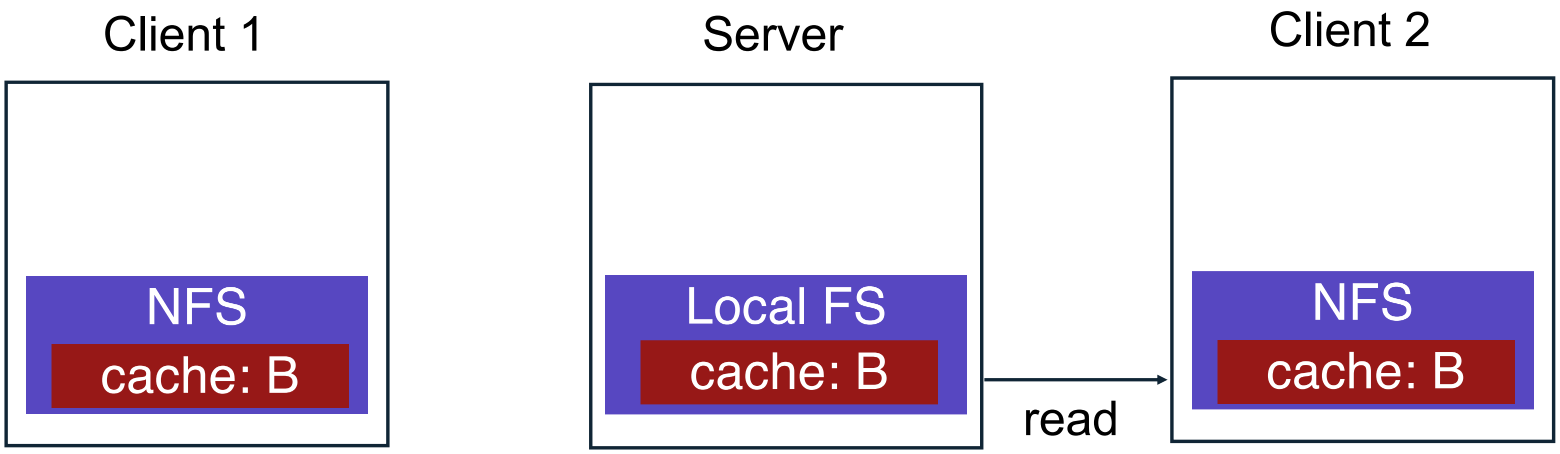
Cache



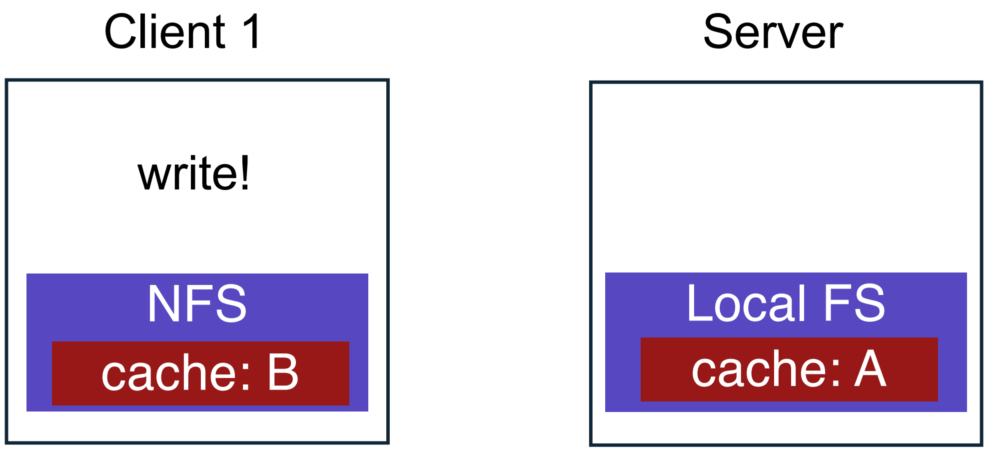
“Stale Cache” problem:
client 2 doesn’t have latest version

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

Cache


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

Problem 1: Update Visibility



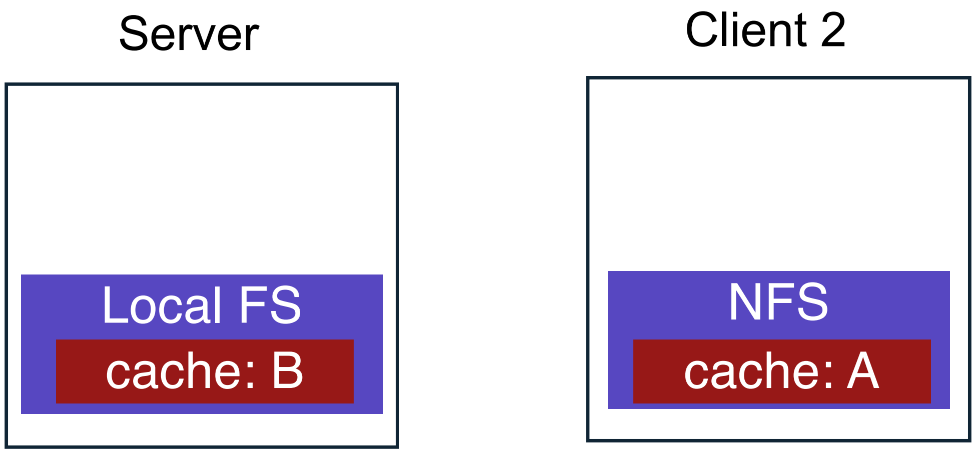
When client buffers a write, how can server (and other clients) see update?

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?

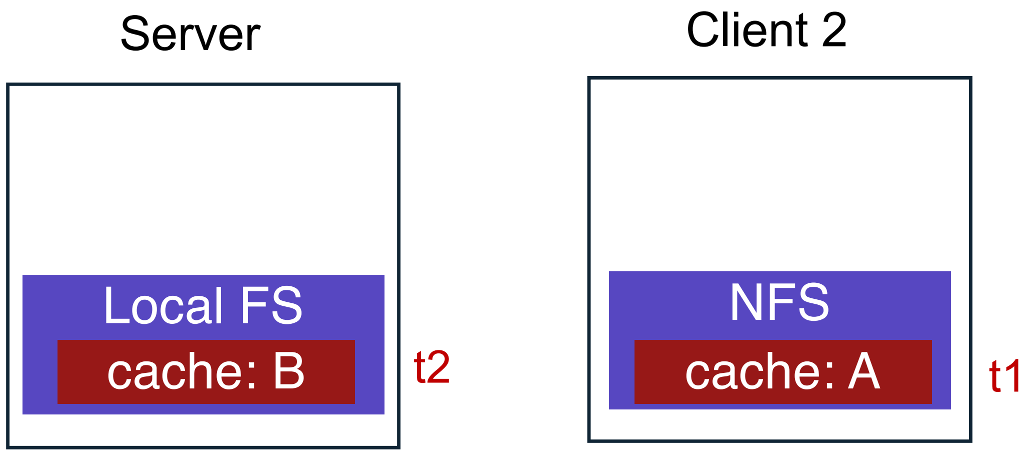
One possible solution:

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

- get’s last modified timestamp for this file (t2) (not block…)
- compare to cache timestamp
- refetch data block if changed since timestamp (t2 > t1)

THE UNIVERSITY of ADELAIDE

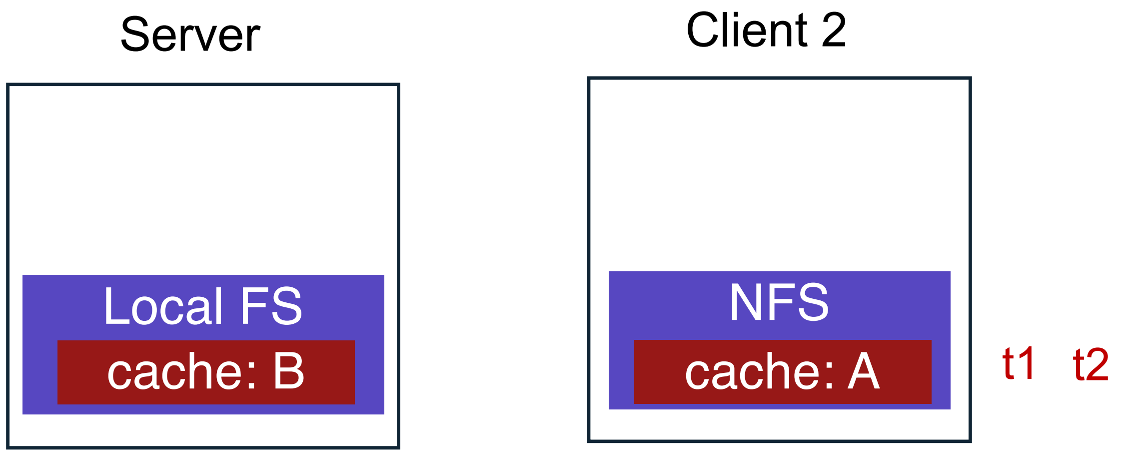
Measure then Build

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

THE UNIVERSITY of ADELAIDE

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

THE UNIVERSITY of ADELAIDE

AFS Goals

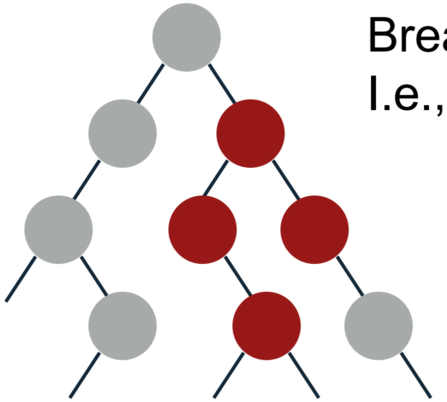
Primary goal: scalability! (many clients per server)

More reasonable semantics for concurrent file access

AFS Design

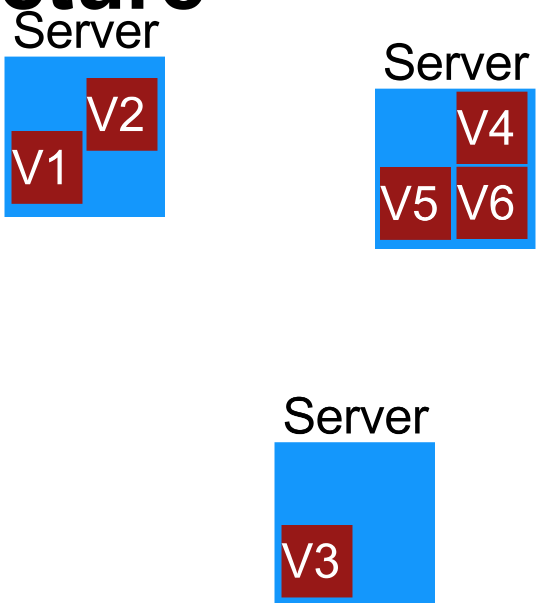
NFS: Server exports local FS

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



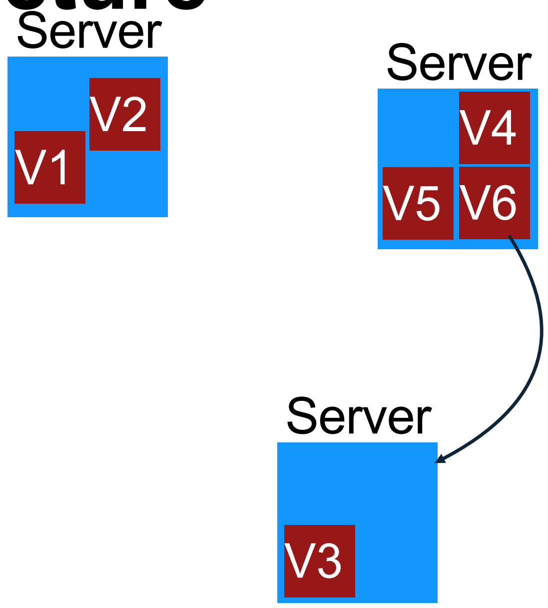
Break directory tree into “volumes”
I.e., partial sub trees

Volume Architecture



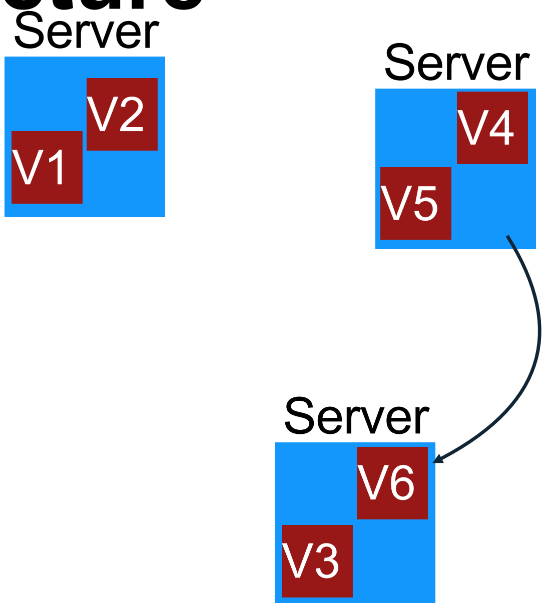
collection of servers store different volumes that together form directory tree

Volume Architecture



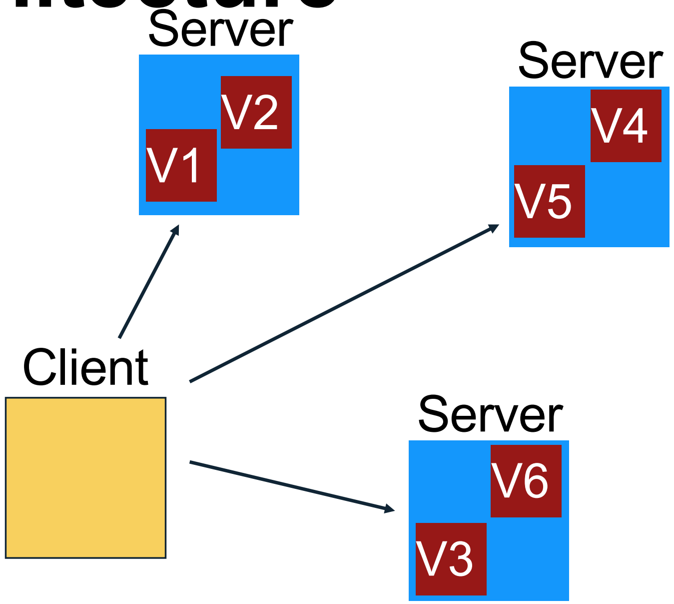
volumes may be moved by an administrator.

Volume Architecture



volumes may be moved by an administrator.

Volume Architecture



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

Communication via RPC
Servers store data in local file systems

THE UNIVERSITY of ADELAIDE

AFS Cache Consistency

Update visibility

Stale cache

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

THE UNIVERSITY *of* ADELAIDE

Cache Consistency

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

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

Stale Cache

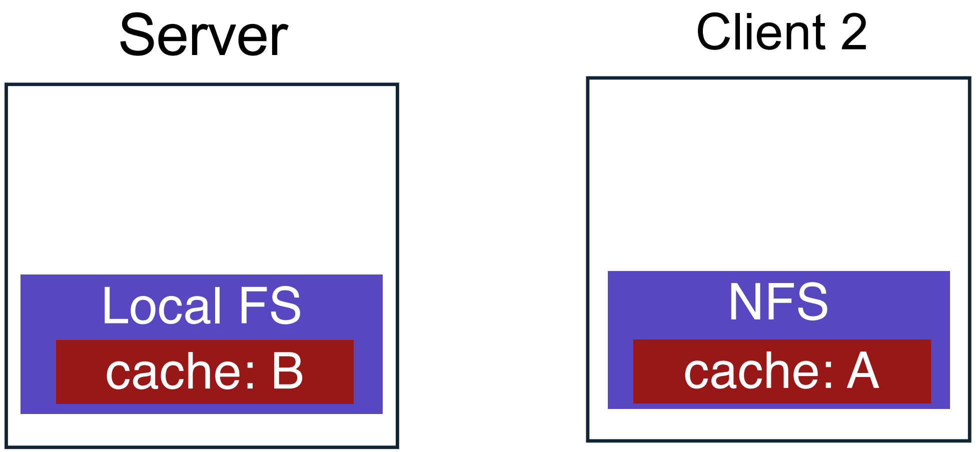
NFS rechecks cache entries compared to the server before using them, assuming the check has not been done “recently”

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



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!

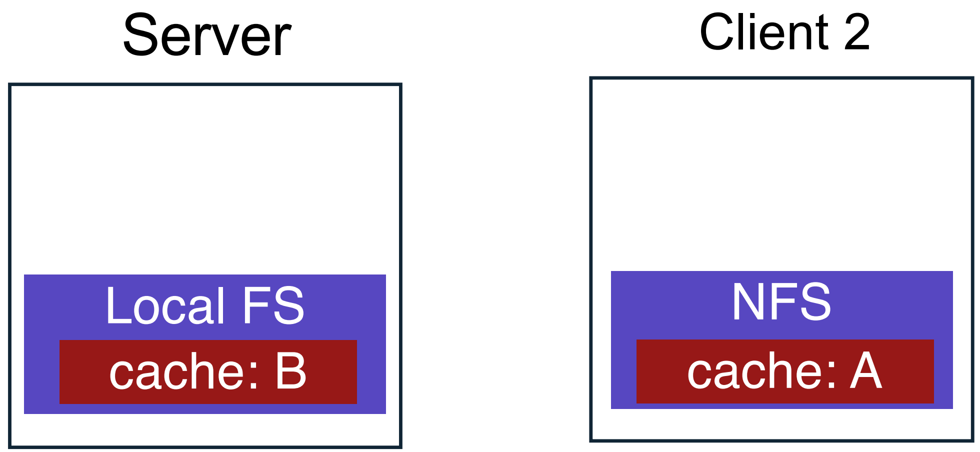
Callbacks: Dealing with STATE

What if client crashes?

What if server runs out of memory?

What if server crashes?

Client Crash



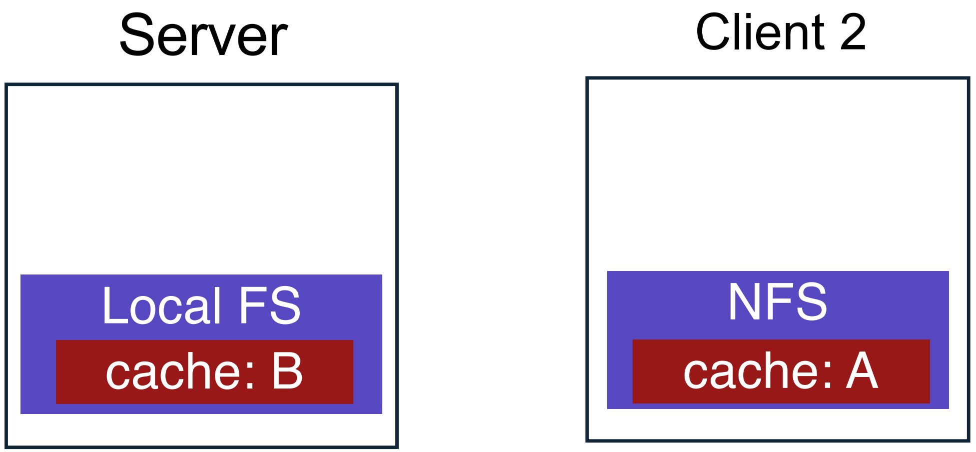
What should client do after reboot?
(remember cached data can be on disk too…)

Concern? may have missed notification that cached copy changed

Option 1: evict everything from cache

Option 2: ??? recheck entries before using

Low Server Memory



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)

Content Distribution Networks (CDN)

A CDN is a highly-distributed platform of servers that helps minimize delays in loading web page content by reducing the physical distance between the server and the user. This helps users around the world view the same high-quality content without slow loading times.

Provides low latency, high performance, and high availability

Provides for geographic and logical load-balancing

Could help mitigate DDoS attacks

With the emergence of cloud computing, CDNs have become a continual trend involving all layers of cloud computing: SaaS, e.g. Google Docs. IaaS, e.g. Amazon. PaaS, e.g. Google App Engine.

Thank you for participating in Operating Systems!

Good luck in the Exam!

THE UNIVERSITY *of* ADELAIDE

PRV12105 CRICOS 00123M

make history.

THE UNIVERSITY of ADELAIDE

This item may include material that has been copied and communicated under the Statutory Licence pursuant to s113P of the Copyright Act 1968 for the educational purposes of the University of Adelaide. Any further copying or communication of this material may be the subject of copyright protection.