CS162
Operating Systems and Systems Programming
Lecture 22

Distributed File Systems & Internet

Professor Natacha Crooks
https://cs162.org/

Slides based on prior slide decks from David Culler, Ion Stoica, John Kubiatowicz, Alison Norman and Lorenzo Alvisi

Recall: What is a Distributed System?

A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

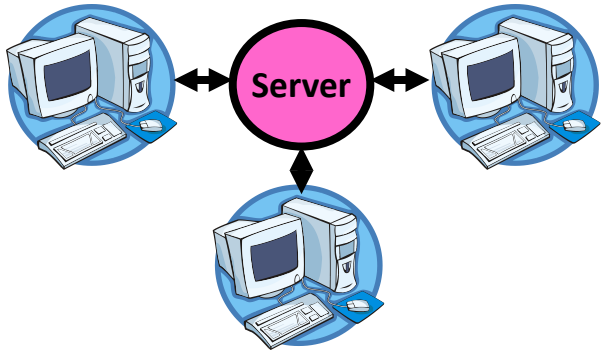
Leslie Lamport,
The Godfather of Distributed Systems

Recall: Centralised vs Distributed Systems

Considered a single computer! All computation was done on the local computer in isolation

The world is a large distributed system

Recall: Two types of distributed systems

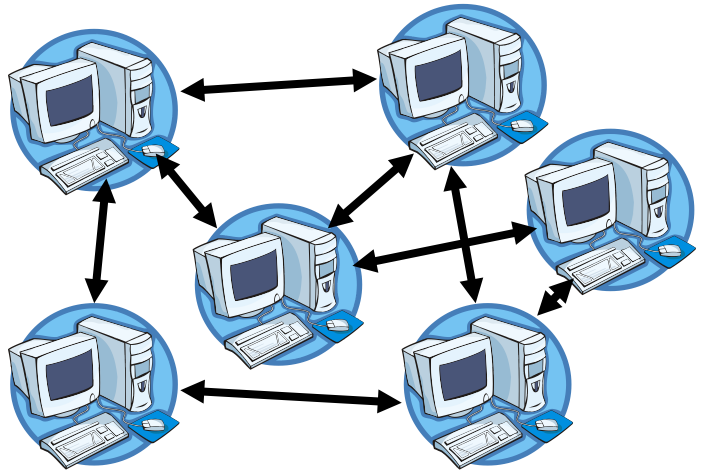


Client/Server Model

One or more server provides *services* to clients

Clients makes *remote procedure calls* to server

Server serves *requests* from clients



Peer-to-Peer Model

Each computer acts as a peer

No hierarchy or central point of coordination

All-way communication between peers through *gossiping*

Recall: The promise of distributed systems

Availability

Proportion of time system is in functioning condition
=> One machine goes down, use another

Fault-tolerance

System has well-defined behaviour when fault occurs
=> Store data in multiple locations

Scalability

Ability to add resources to system to support more work
⇒Just add machines when need more storage/processing power

Transparency

The ability of the system to mask its complexity behind a simple interface

Remote Procedure Call (RPC)

Raw messaging is a bit too low-level for programming

- Must wrap up information into message at source
- Must decide what to do with message at destination
- May need to sit and wait for multiple messages to arrive
- And must deal with machine representation by hand

Remote Procedure Call (RPC)

Another option: Remote Procedure Call (RPC)

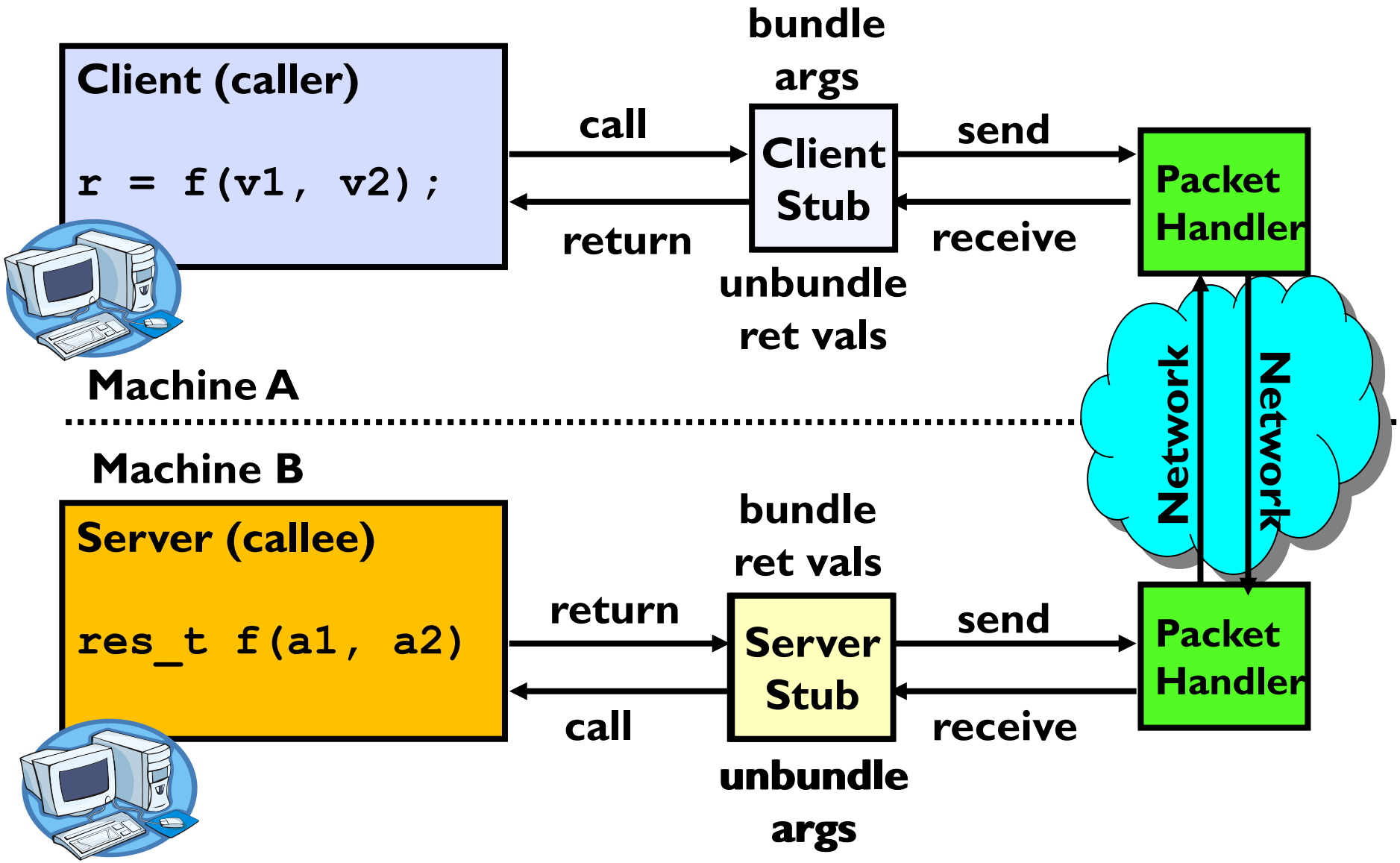
- Calls a procedure on a remote machine
- Idea: Make communication look like an ordinary function call
- Automate all of the complexity of translating between representations
- Client calls:

```
remoteFileSystem→Read("rutabaga");
```

- Translated automatically into call on server:

```
fileSys→Read("rutabaga");
```

RPC Information Flow



RPC Implementation

Request-response message passing (under covers!)

"Stub" provides glue on client/server

- Client stub is responsible for "marshalling" arguments and "unmarshalling" the return values
- Server-side stub is responsible for "unmarshalling" arguments and "marshalling" the return values.

Marshalling involves (depending on system)

- Converting values to a canonical form, serializing objects, copying arguments passed by reference, etc.

RPC Details (1/3)

Equivalence with regular procedure call
- Parameters ⇔ Request Message
- Result ⇔ Reply message
- Name of Procedure: Passed in request message
- Return Address: mbox2 (client return mail box)

Stub generator: Compiler that generates stubs
- Input: interface definitions in an "interface definition language (IDL)"
  - » Contains, among other things, types of arguments/return
- Output: stub code in the appropriate source language
  - » Code for client to pack message, send it off, wait for result, unpack result and return to caller
  - » Code for server to unpack message, call procedure, pack results, send them off

RPC Details (2/3)

Cross-platform issues:

- What if client/server machines are different architectures/ languages?
  - » Convert everything to/from some canonical form
  - » Tag every item with an indication of how it is encoded (avoids unnecessary conversions)

How does client know which mbox (destination queue) to send to?

- Need to translate name of remote service into network endpoint (Remote machine, port, possibly other info)
- Binding: the process of converting a user-visible name into a network endpoint
  - » This is another word for "naming" at network level
  - » Static: fixed at compile time
  - » Dynamic: performed at runtime

RPC Details (3/3)

- Dynamic Binding
  - Most RPC systems use dynamic binding via name service
    - » Name service provides dynamic translation of service → mbox
  - Why dynamic binding?
    - » Access control: check who is permitted to access service
    - » Fail-over: If server fails, use a different one

- What if there are multiple servers?
  - Could give flexibility at binding time
    - » Choose unloaded server for each new client
  - Could provide same mbox (router level redirect)
    - » Choose unloaded server for each new request
    - » Only works if no state carried from one call to next

- What if multiple clients?
  - Pass pointer to client-specific return mbox in request

Problems with RPC: Non-Atomic Failures

Different failure modes in dist. system than on a single machine

Consider many different types of failures
- User-level bug causes address space to crash
- Machine failure, kernel bug causes all processes on same machine to fail
- Some machine is compromised by malicious party

Can easily result in inconsistent view of the world
- Did my cached data get written back or not?
- Did server do what I requested or not?

Answer? Distributed transactions/2PC

Problems with RPC: Performance

RPC is *not* performance transparent:
- Cost of Procedure call « same-machine RPC « network RPC
- Overheads: Marshalling, Stubs, Kernel-Crossing, Communication

Programmers must be aware that RPC is not free
- Caching can help, but may make failure handling complex

CS162 Students Vs Elon Musk

Do you believe this?

Elon Musk @elonmusk · 1d

Btw, I'd like to apologize for Twitter being super slow in many countries. App is doing >1000 poorly batched RPCs just to render a home timeline!

Elon Musk
@elonmusk

Replying to @sampullara

I was told ~1200 RP
engineers at Twitter
microservices. The

Popeska @Popeska · Nov 14

Replying to @elonmusk and @sampullara

The batching is done server-side where the initial client request doesn't matter - GraphQL resolves the request across the microservices and then sends it back. The internal graph is the same regardless of request origin

19 47 1,018

Same app in US takes ~2 secs to refresh (too long), but ~20 secs in India, due to bad batching/verbose comms. Actually useful data transferred is low.

7:22 AM · Nov 14, 2022 · Twitter for iPhone

Topic roadmap

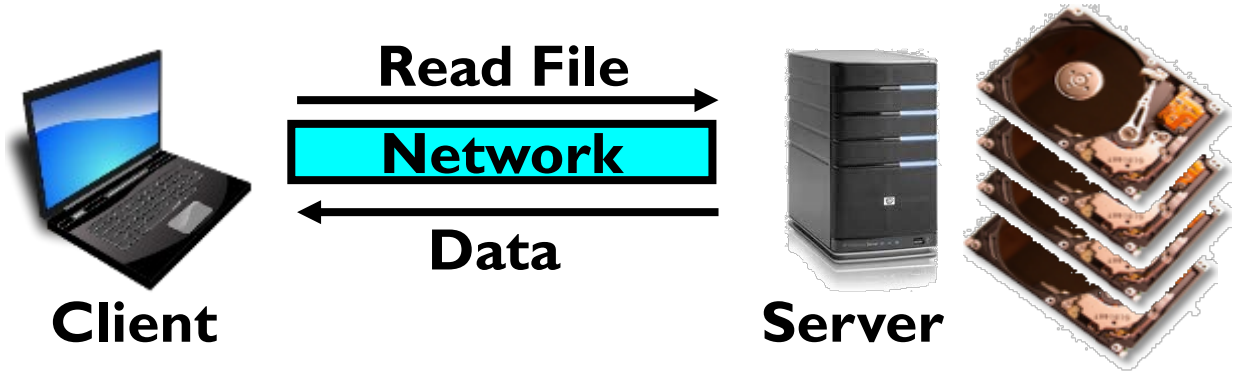
Distributed File Systems

Peer-To-Peer System:
The Internet

Distributed Data Processing

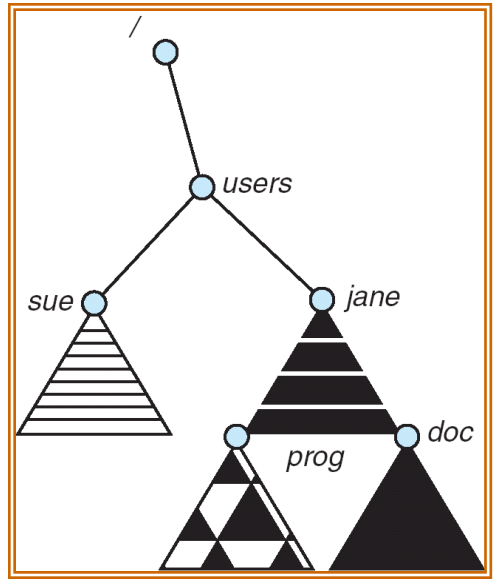
Coordination
(Atomic Commit and
Consensus)

Distributed File Systems



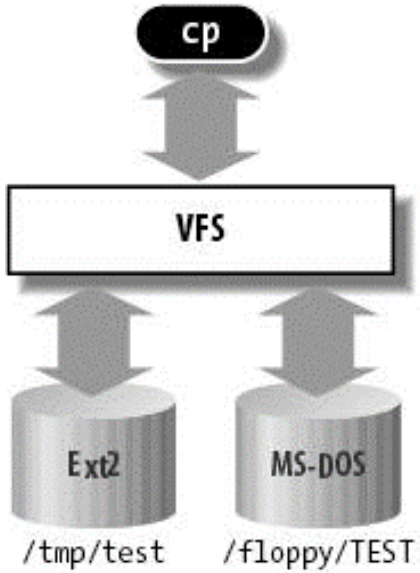
Transparent access to files stored on a remote disk

Mount remote files into your local file system
- Directory in local file system refers to remote files
- e.g., `/users/jane/prog/foo.c` on laptop actually refers to `/prog/foo.c` on `crooks.cs.berkeley.edu`



Naming Choices:
- [Hostname,localname]: Filename includes server
- A global name space: Filename unique in "world"

Virtual Filesystem Switch

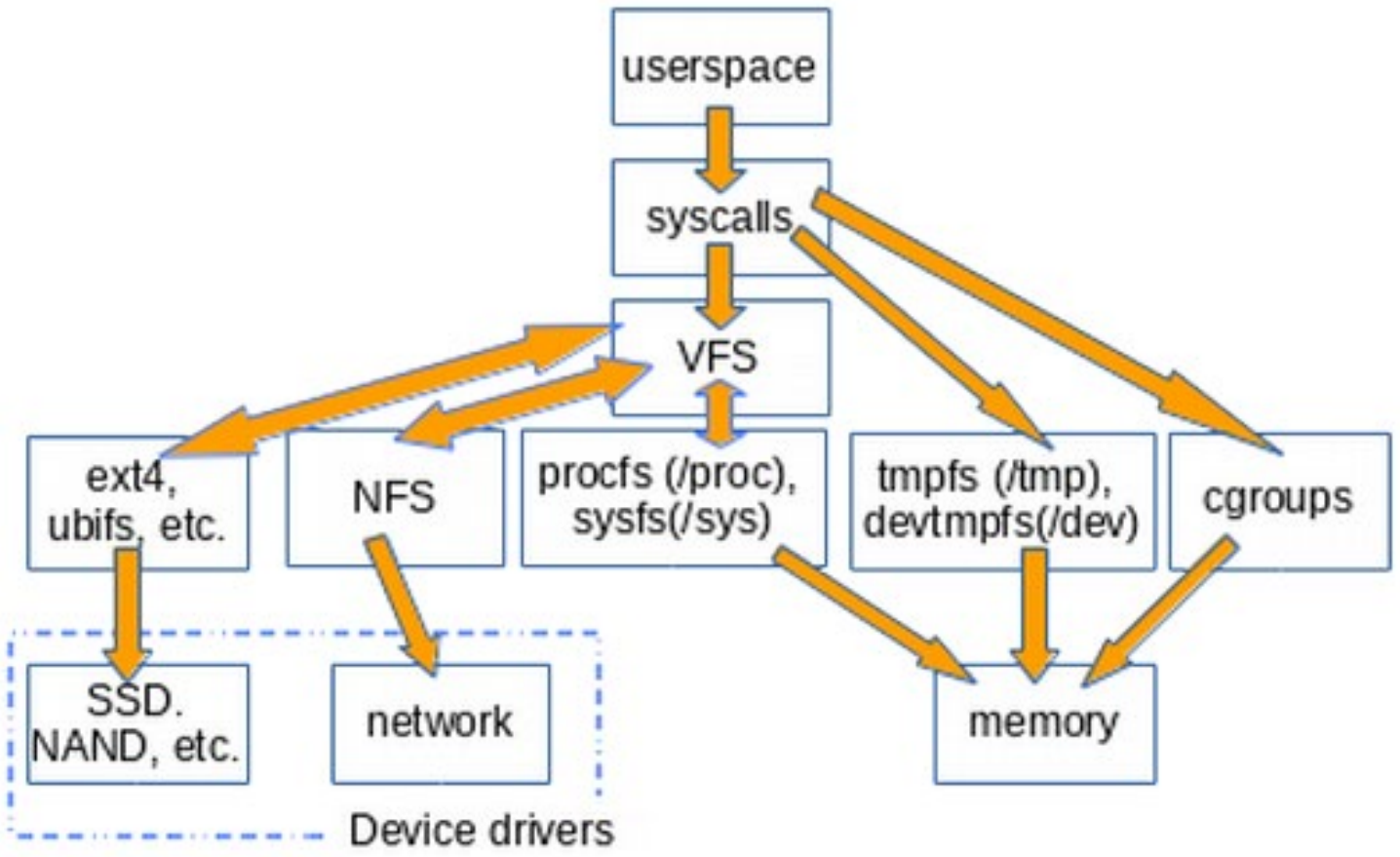



```
inf = open("/floppy/TEST", O_RDONLY, 0);
outf = open("/tmp/test",
       O_WRONLY|O_CREAT|O_TRUNC, 0600);
do {
    i = read(inf, buf, 4096);
    write(outf, buf, i);
} while (i);
close(outf);
close(inf);
```

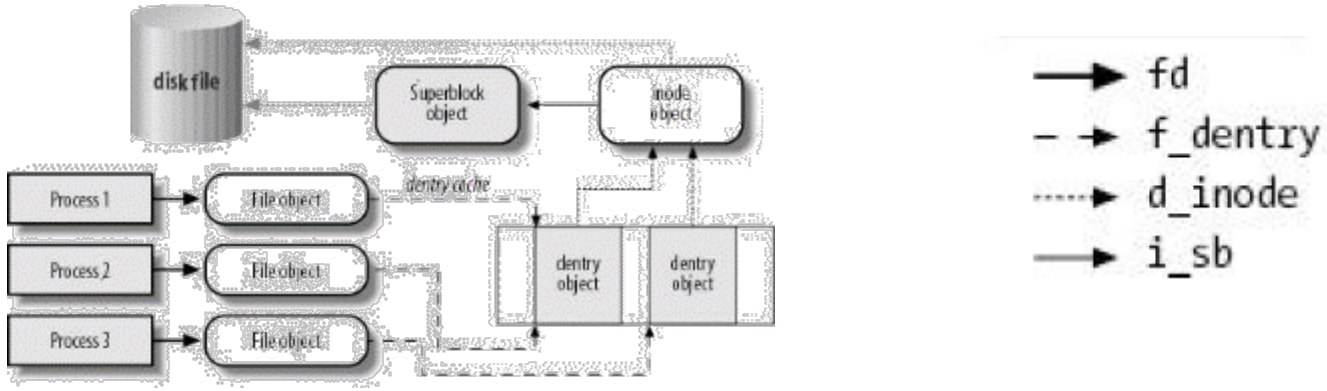
Virtual abstraction of file system
- Provides virtual superblocks, inodes, files, etc
- Compatible with a variety of local and remote file systems

VFS allows the same system call interface (the API) to be used for different types of file systems
- The API is to the VFS interface, rather than any specific type of file system

Example Linux mouting tree



VFS Common File Model in Linux

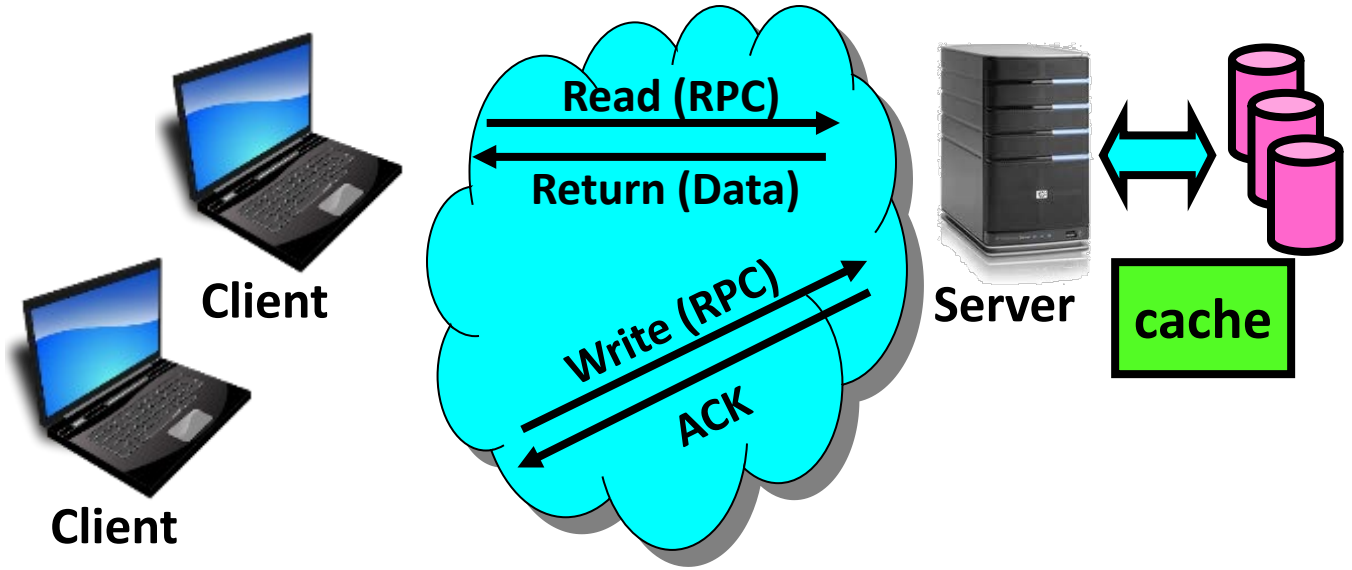


Four primary object types for VFS:

- superblock object: represents a specific mounted filesystem
- inode object: represents a specific file
- dentry object: represents a directory entry
- file object: represents open file associated with process

May need to fit the model by faking it

Simple Distributed File System

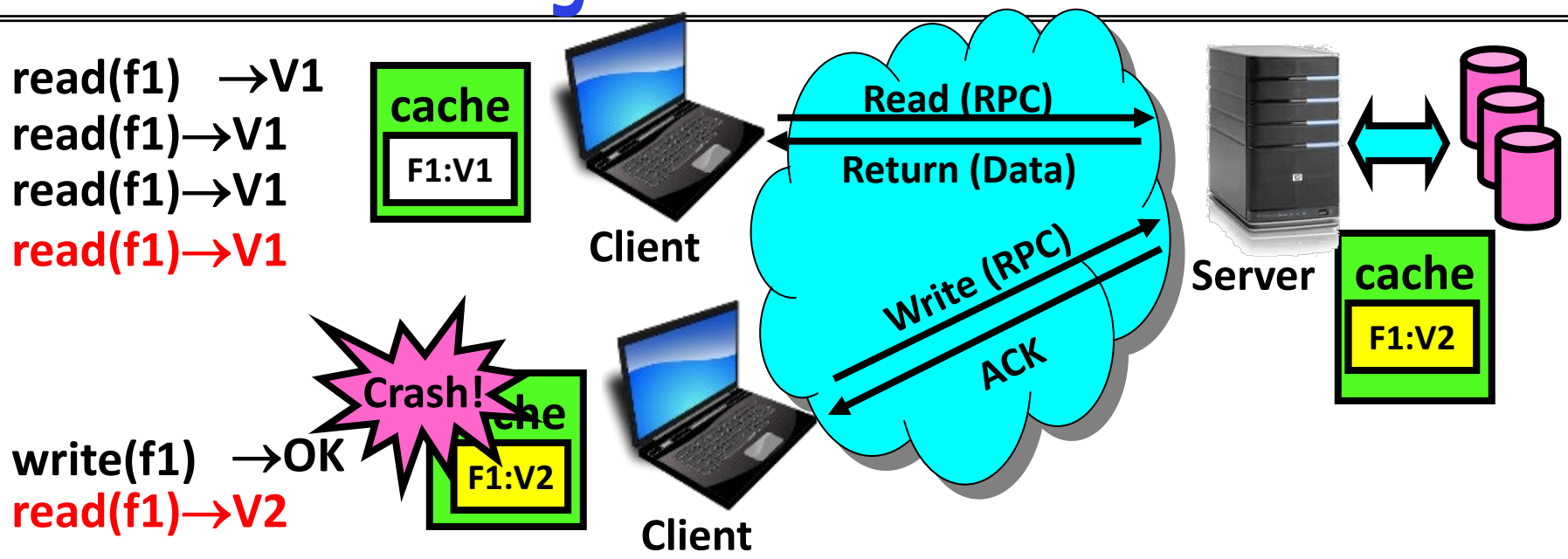


Remote Disk: Reads and writes forwarded to server. Use Remote Procedure Calls (RPC) to translate file system calls into remote requests

Advantage: Server provides consistent view of file system to multiple clients

Problems? Performance!
- Going over network is slower than going to local memory
- Lots of network traffic/not well pipelined
- Server can be a bottleneck

Use of caching to reduce network load



Idea: Use caching to reduce network load
- In practice: use buffer cache at source and destination

Advantage: if open/read/write/close can be done locally, don't need to do any network traffic...fast!

Problems:
- Failure:
  - » Client caches have data not committed at server
- Cache consistency! Client caches not consistent with server/each other

Dealing with Failures

What if server crashes? Can client wait until it comes back and just continue making requests?

- Changes in server's cache but not in disk are lost

What if there is shared state across RPC's?

- Client opens file, then does a seek
- Server crashes
- What if client wants to do another read?

Similar problem: What if client removes a file but server crashes before acknowledgement?

Stateless Protocol

Stateless Protocol: A protocol in which all information required to service a request is included with the request

Idempotent Operations – repeating an operation multiple times is same as executing it just once (e.g., storing to a mem addr.)

Client: timeout expires without reply, just run the operation again (safe regardless of first attempt)

Recall HTTP: Also a stateless protocol

- Include cookies with request to simulate a session

Case Study: Network File System (NFS)

It's an open world!

Three Layers for NFS system

- UNIX file-system interface: open, read, write, close calls + file descriptors
- VFS layer: distinguishes local from remote files
  - » Calls the NFS protocol procedures for remote requests
- NFS service layer: bottom layer of the architecture
  - » Implements the NFS protocol

NFS Continued

NFS Protocol: RPC for file operations on server
- Reading/searching a directory
- manipulating links and directories
- accessing file attributes/reading and writing files

Write-through caching: Modified data committed to server's disk before results are returned to the client
- lose some of the advantages of caching
- time to perform write() can be long
- Need some mechanism for readers to eventually notice changes! (more on this later)

NSF2: Every write persisted.
NSF3: open-to-close consistency

NFS Continued

NFS servers are stateless; each request provides all arguments require for execution

- E.g. reads include information for entire operation, such as `ReadAt(inumber,position)`, not `Read(openfile)`
- No need to perform network open() or close() on file – each operation stands o
- n its own

Idempotent: Performing requests multiple times has same effect as performing them exactly once

- Example: Read and write file blocks: just re-read or re-write file block – no other side effects

- Example: What about "remove"? NFS does operation twice and second time returns an advisory error

NFS Continued

NFSPROC_GETATTR	file handle returns: attributes
NFSPROC_SETATTR	file handle, attributes returns: –
NFSPROC_LOOKUP	directory file handle, name of file/dir to look up returns: file handle
NFSPROC_READ	file handle, offset, count data, attributes
NFSPROC_WRITE	file handle, offset, count, data attributes
NFSPROC_CREATE	directory file handle, name of file, attributes –
NFSPROC_REMOVE	directory file handle, name of file to be removed –
NFSPROC_MKDIR	directory file handle, name of directory, attributes file handle
NFSPROC_RMDIR	directory file handle, name of directory to be removed –
NFSPROC_READDIR	directory handle, count of bytes to read, cookie returns: directory entries, cookie (to get more entries)

NFS Continued

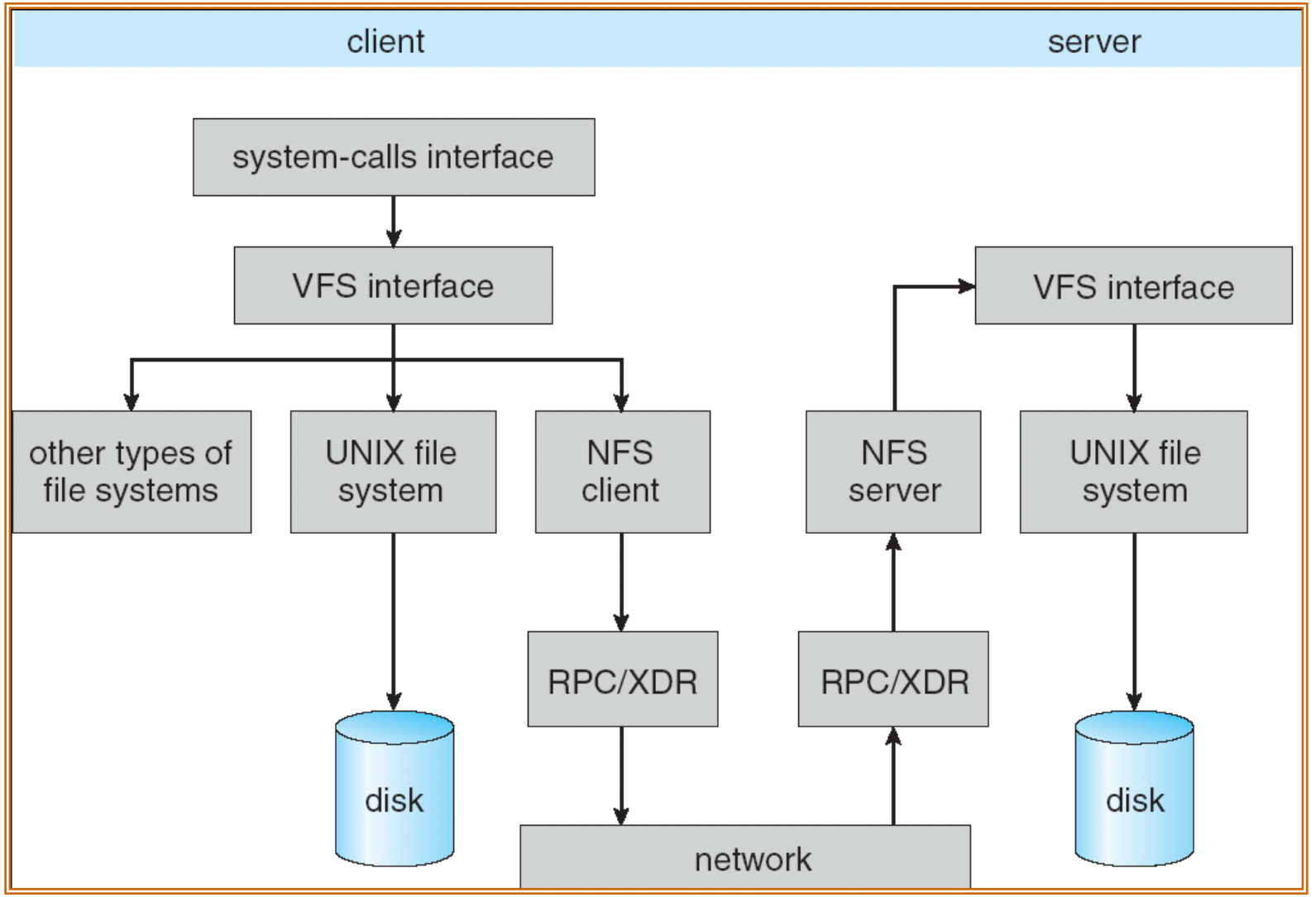
Client	Server
fd = open("/foo", ...);	
Send LOOKUP (rootdir FH, "foo")	
	Receive LOOKUP request
	look for "foo" in root dir
	return foo's FH + attributes
Receive LOOKUP reply	
allocate file desc in open file table	
store foo's FH in table	
store current file position (0)	
return file descriptor to application	
read(fd, buffer, MAX);	
Index into open file table with fd	
get NFS file handle (FH)	
use current file position as offset	
Send READ (FH, offset=0, count=MAX)	
	Receive READ request
	use FH to get volume/inode num
	read inode from disk (or cache)
	compute block location (using offset)
	read data from disk (or cache)
	return data to client
Receive READ reply	
update file position (+bytes read)	
set current file position = MAX	
return data/error code to app	
read(fd, buffer, MAX);	
Same except offset=MAX and set current file position = 2*MAX	

NFS Continued

Failure Model: Transparent to client system

- Is this a good idea? What if you are in the middle of reading a file and server crashes?
- Options (NFS Provides both):
  - » Hang until server comes back up (next week?)
  - » Return an error. (Of course, most applications don't know they are talking over network)

NFS Architecture

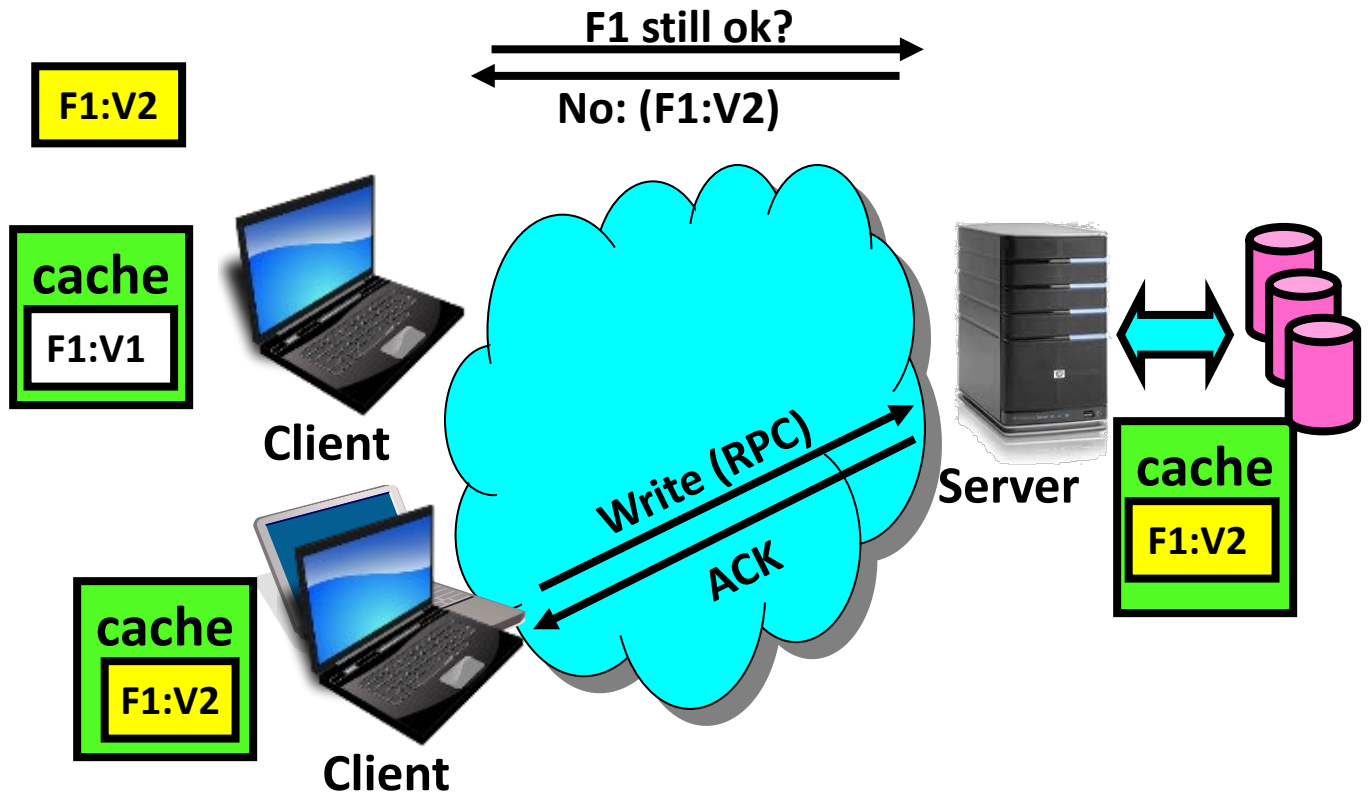


NFS Cache consistency

NFS protocol: weak consistency
- Client polls server periodically to check for changes
  - » Polls server if data hasn't been checked in last 3-30 seconds (exact timeout it tunable parameter).
  - » Thus, when file is changed on one client, server is notified, but other clients use old version of file until timeout.

What if multiple clients write to same file?
- » In NFS, can get either version (or parts of both)
- » Completely arbitrary!

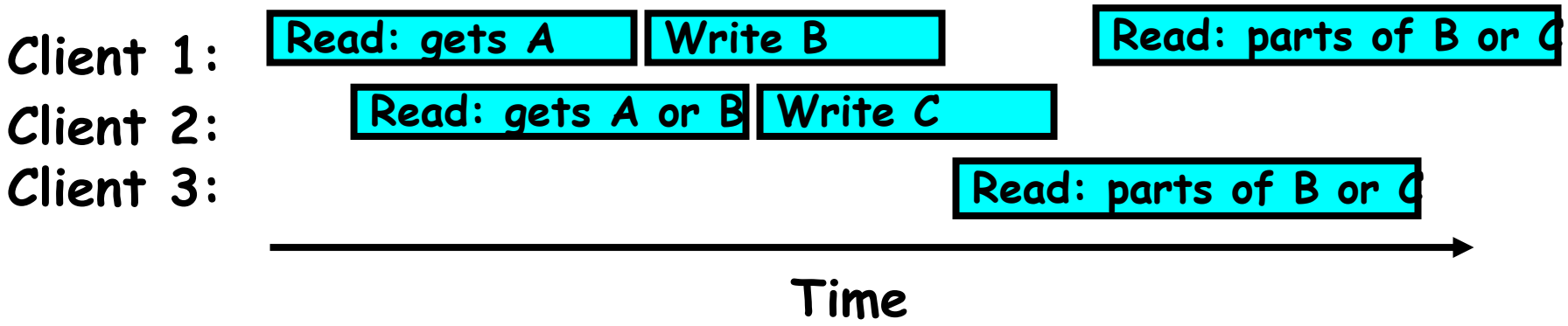
NFS Cache consistency



Sequential Ordering Constraints

What sort of cache coherence might we expect?
- i.e. what if one CPU changes file, and before it's done, another CPU reads file?

Example: Start with file contents = "A"



Sequential Ordering Constraints

What would we actually want?

- Assume we want distributed system to behave exactly the same as if all processes are running on single system
  - » If read finishes before write starts, get old copy
  - » If read starts after write finishes, get new copy
  - » Otherwise, get either new or old copy

- For NFS:
  - » If read starts more than 30 seconds after write, get new copy; otherwise, could get partial update

NFS Pros and Cons

NFS Pros:

- Simple, Highly portable

NFS Cons:

- Sometimes inconsistent!
- Doesn't scale to large # clients
  - » Must keep checking to see if caches out of date
  - » Server becomes bottleneck due to polling traffic

Case study: The Internet

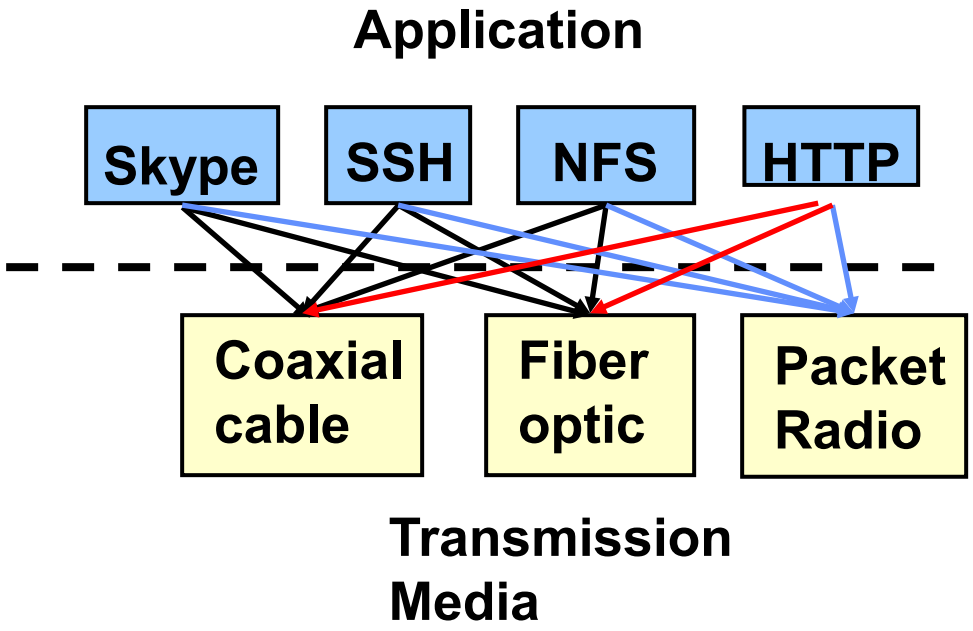
The Internet is the largest distributed system that exists!

Many different applications
- Email, web, P2P, etc.

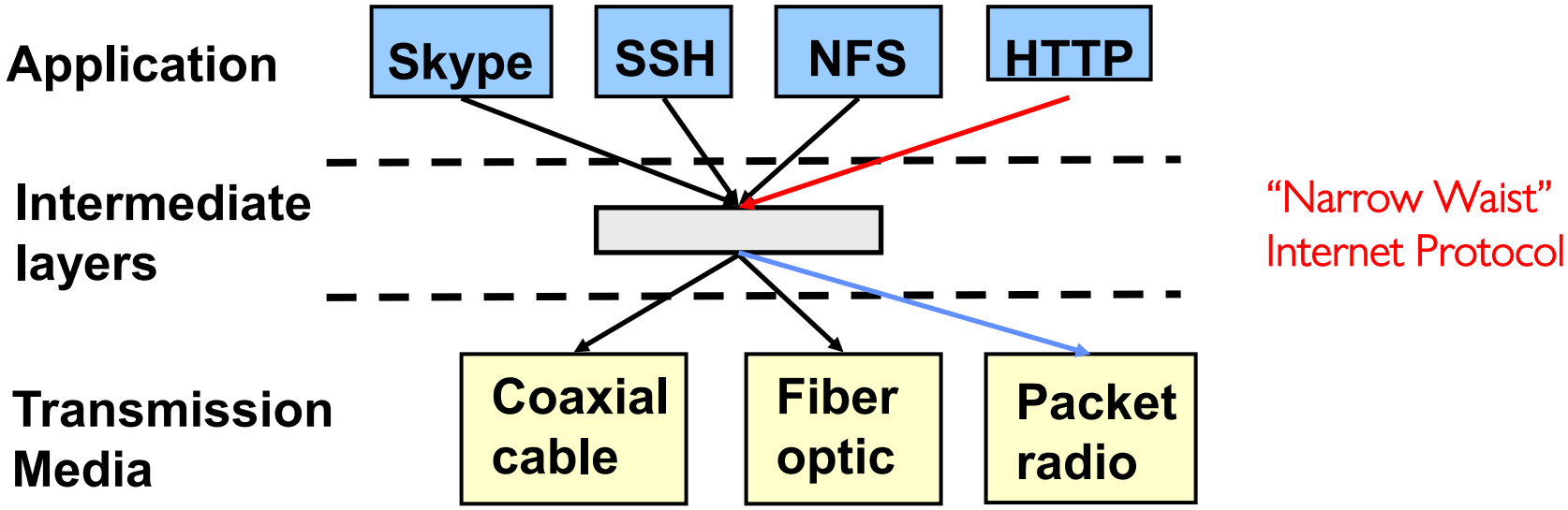
Many different operating systems and devices

Many different network styles and technologies
- Wireless, wired, optical

How do we organize this mess
- Layering & end-to-end principle



The Internet: Layers, Layers, Layers

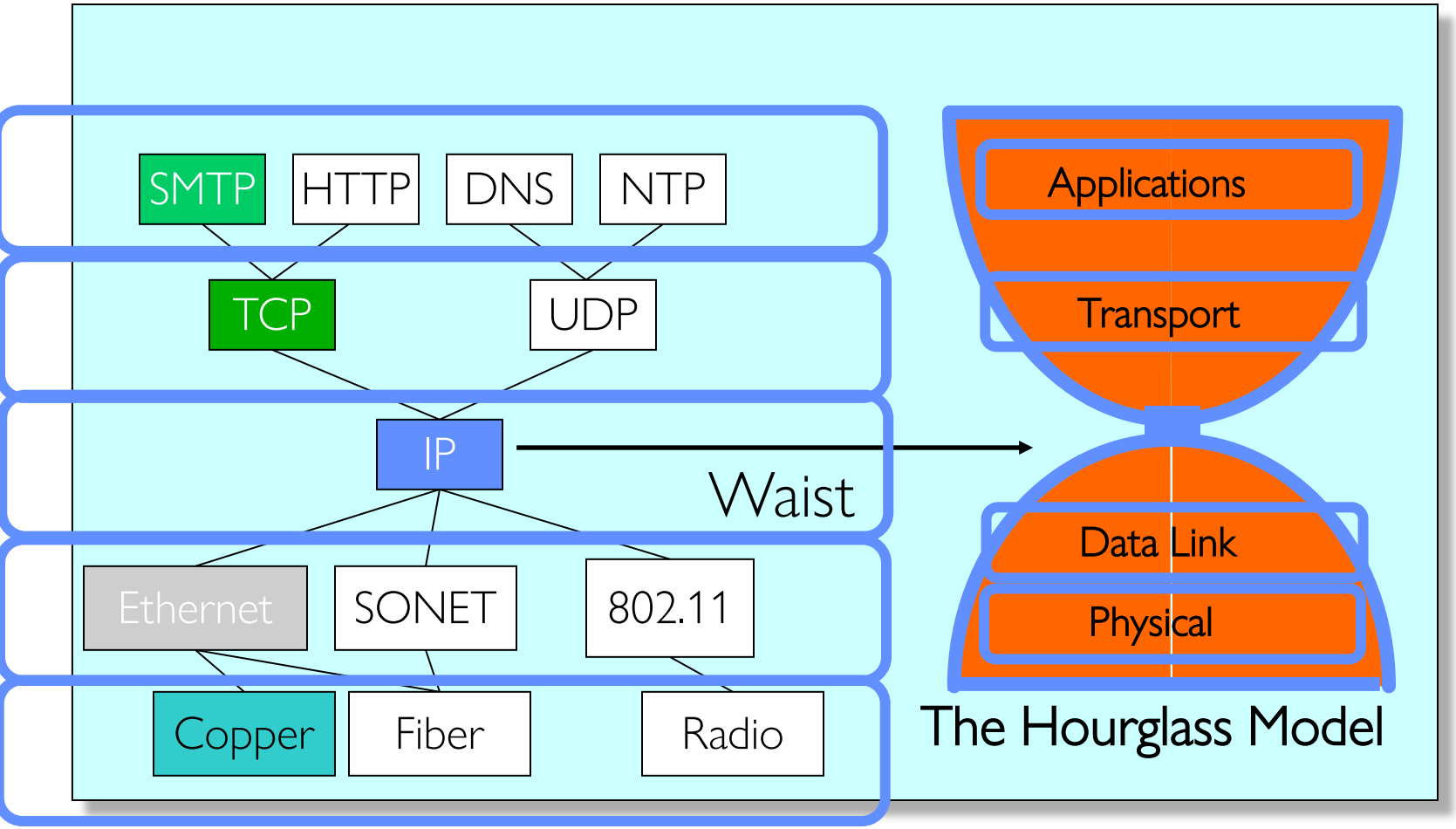


Introduce intermediate layers that provide set of abstractions for various network functionality & technologies

- A new app/media implemented only once

Goal: Reliable communication channels on which to build distributed applications

The Internet: The *hourglass*



"Narrow waist" facilitates interoperability

Layers "abstract" away hardware so that upper layers are agnostic to lower layers

=> Sound familiar?

The Internet: Implications of Hourglass

Single Internet-layer module (IP):

Allows arbitrary networks to interoperate
- Any network technology that supports IP can exchange packets

Allows applications to function on all networks
- Applications that can run on IP can use any network

Supports simultaneous innovations above and below IP
- But changing IP itself, i.e., IPv6, very involved

The Internet: Drawbacks of Layering

Layer N may duplicate layer N-1 functionality
- E.g., error recovery to retransmit lost data

Layers may need same information
- E.g., timestamps, maximum transmission unit size

Layering can hurt performance
- E.g., hiding details about what is really going on

Some layers are not always cleanly separated
- Inter-layer dependencies for performance reasons
- Some dependencies in standards (header checksums)

End-To-End Argument

Hugely influential paper:

- "End-to-End Arguments in System Design" by Saltzer, Reed, and Clark ('84)

"Sacred Text" of the Internet

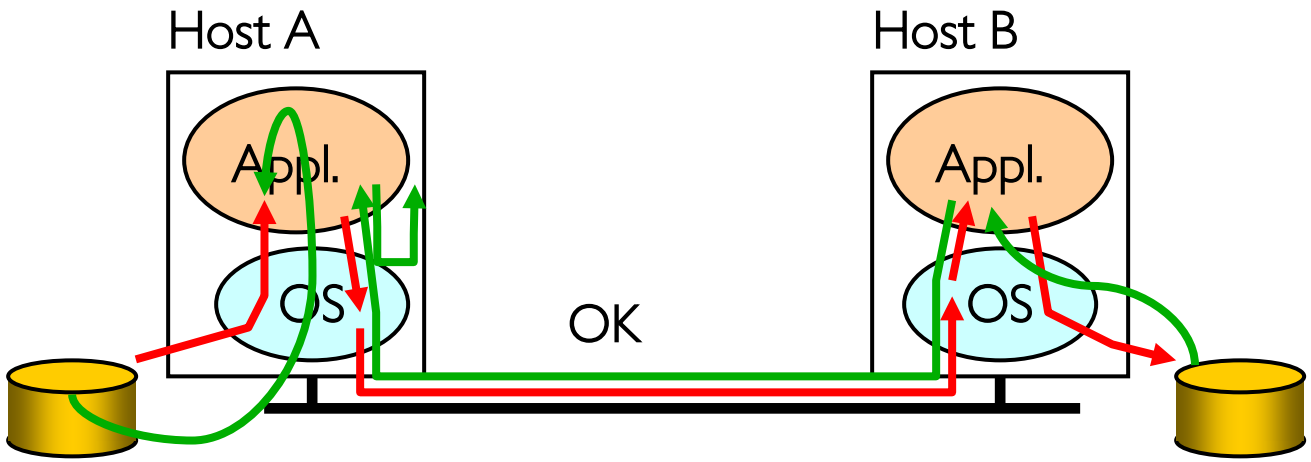
- Endless disputes about what it means
- Everyone cites it as supporting their position

Simple Message: Some types of network functionality can only be correctly implemented end-to-end

- Reliability, security, etc.

Hosts cannot rely on the network help to meet requirement, so must implement it themselves

Example: Reliable File Transfer



Solution 1: make each step reliable, and then concatenate them

Solution 2: end-to-end check and try again if necessary

Discussion

Solution 1 is incomplete

- What happens if memory is corrupted?
- Receiver has to do the check anyway!

Solution 2 is complete

- Full functionality can be entirely implemented at application layer with no need for reliability from lower layers

Is there any need to implement reliability at lower layers?

- Well, it could be more efficient

End-to-End Principle

Implementing complex functionality in the network:

- Doesn't always reduce host implementation complexity
- Does increase network complexity
- Probably imposes delay and overhead on all applications, even if they don't need functionality

However, implementing in network can enhance performance in some cases

  - e.g., very lossy link

Conservative Interpretation of E2E

Don't implement a function at the lower levels of the system unless it can be completely implemented at this level

Or: Unless you can relieve the burden from hosts, don't bother

Moderate Interpretation

Think twice before implementing functionality in the network

If hosts can implement functionality correctly, implement it in a lower layer only as a performance enhancement

But do so only if it does not impose burden on applications that do not require that functionality

This is the interpretation we are using