CSCI-1680
Layering and Encapsulation

Nick DeMarinis

Based partly on lecture notes by Rodrigo Fonseca, David Mazières, Phil Levis, John Jannotti

Administrivia

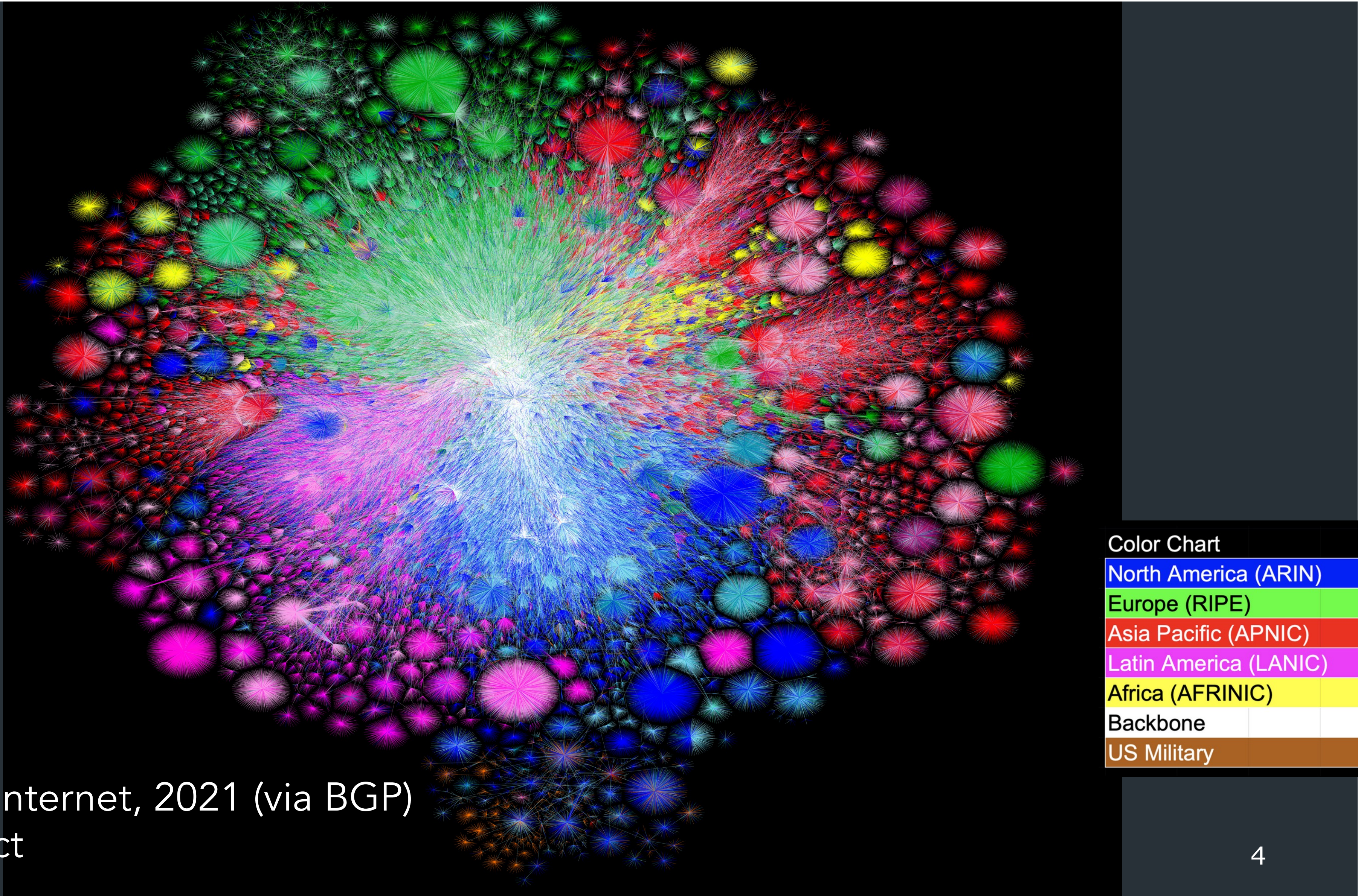
- HW0: Due TODAY by 11:59pm
- Container setup: due by Thursday
  - If you have issues, please fill out the form

- Snowcast out later today (look for Ed post)
  - Gearup Thursday 9/12 5-7pm CIT368 (+Zoom, recorded)

- Milestone due by Monday 9/16 by 11:59pm EDT
  - Warmup and first steps + design doc for the rest

Topics for Today

- Layering and Encapsulation
- Intro to IP, TCP, UDP
- Demo on sockets



Map of the Internet, 2021 (via BGP)
OPTE project

How do we make sense of all this?

How do we make sense of all this?

- *Very* large number of computers
- Diverse of technologies and constraints
- Lots of *multiplexing*
- No single administrative entity

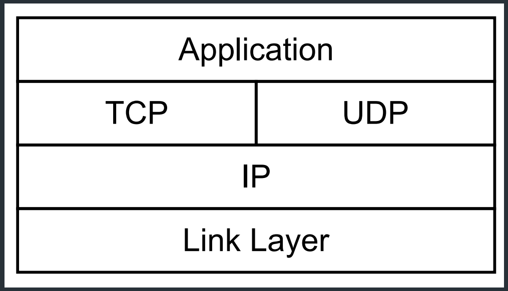
- Evolving demands, protocols, apps => different requirements!

How do we make sense of all this?

- *Very* large number of computers
- Diverse of technologies and constraints
- Lots of *multiplexing*
- No single administrative entity

- Evolving demands, protocols, apps => different requirements!

Layering



<u>Abstraction to the rescue!</u>

Idea: Break problem into separate parts, solve part independently

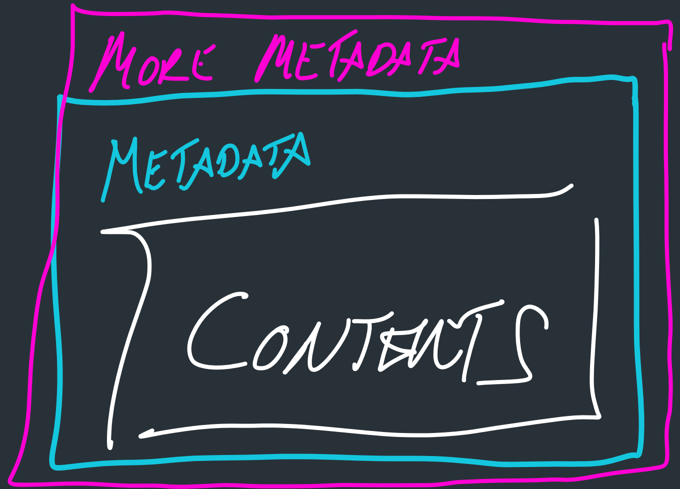
<u>Encapsulate</u> data from “higher layer” inside “lower layer”
=> Lower layer can handle data without caring what’s above it!

An analogy

How to deliver a package?

Metadata that tells us where the data should go ("label" on the package)

Data: content, turns into packets
Metatdata: header



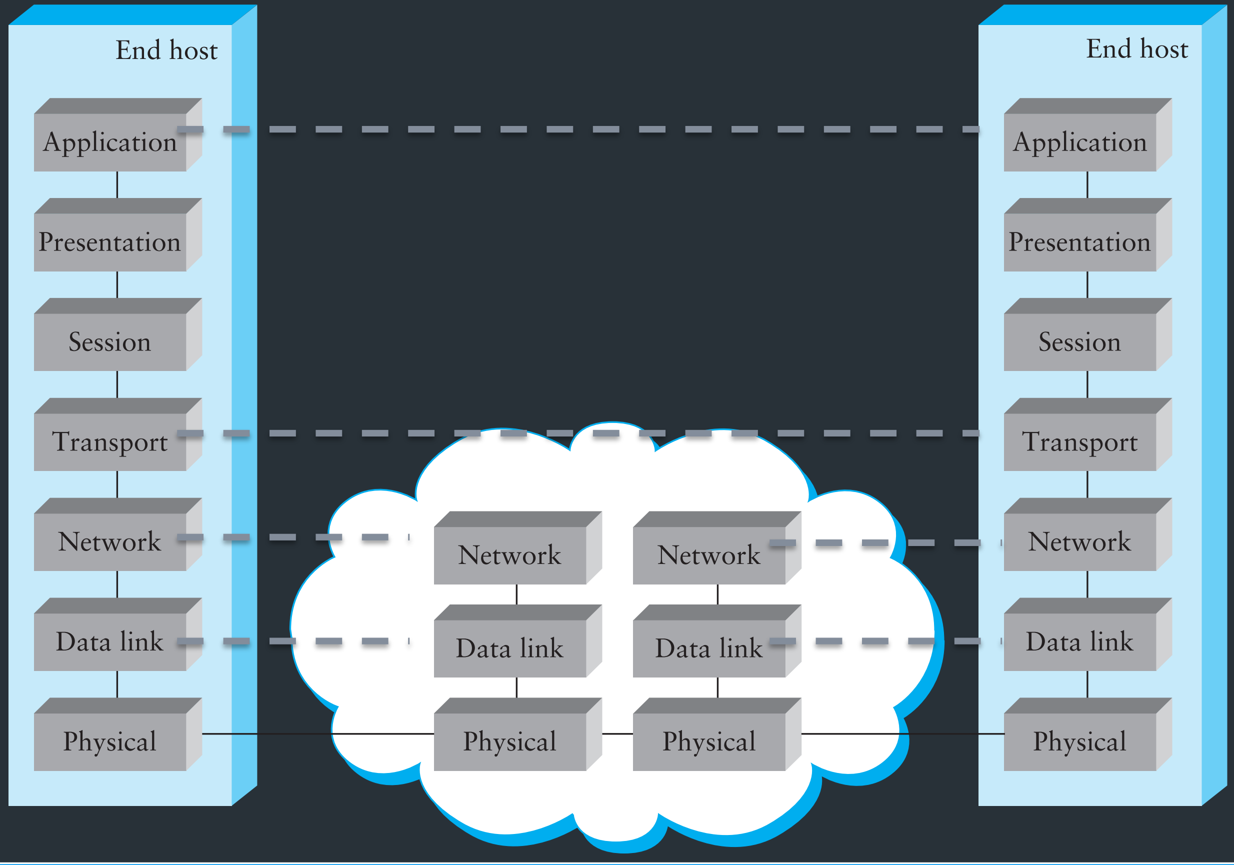
The big complex picture

Application Protocol (L7)

Transport Protocol (L4)

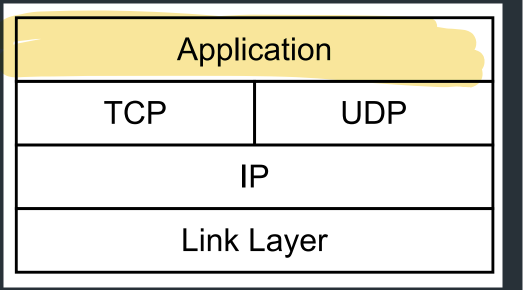
Network Protocol (L3)

Link-Layer Protocol (L2)



"OSI reference model" or "7-layer model"

Applications (Layer 7)



The applications/programs/etc you use every day

Examples:

- HTTP/HTTPS: Web traffic (browser, etc)
- SSH: secure shell
- FTP: file transfer
- DNS (more on this later)
- …

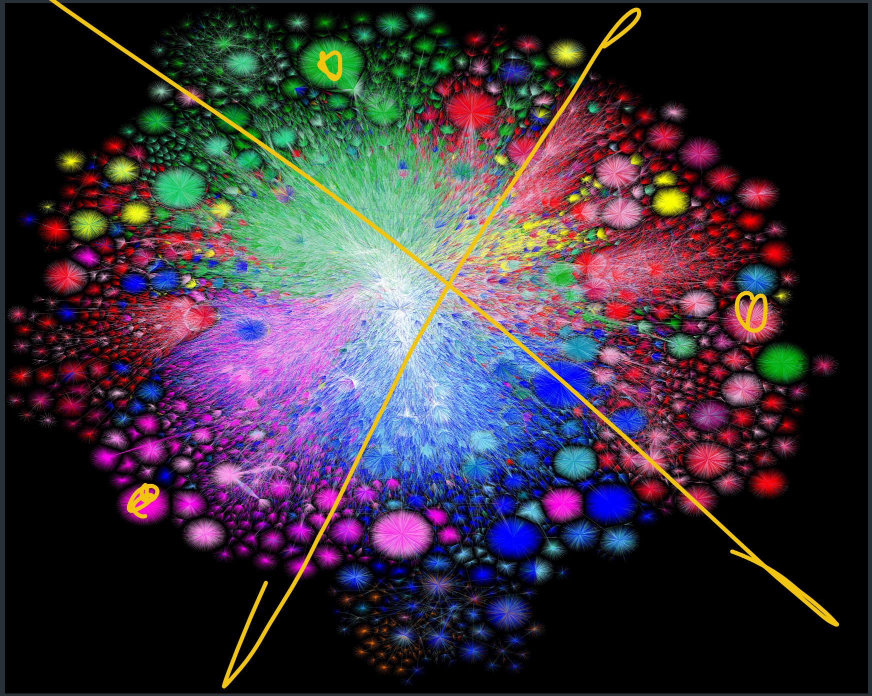
When you're building programs, you usually work here

How to make apps use the network?

```
print("Hello world")
```

↓

```
send("Hello world")
```



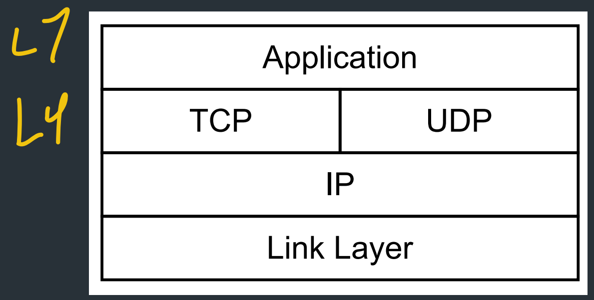
How to make apps use the network?

```
print("Hello world")
```

```
send("Hello world")
```

⇒ Want to send useful messages, not packets
⇒ Don't have to care about <u>how</u> path packet takes to get from A->B, we just want it to get there

Apps rely on: transport layer (layer 4)

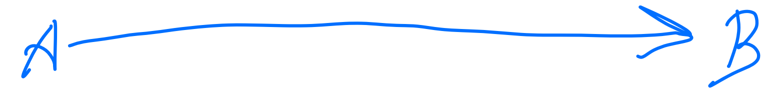


OS provides as "socket interface": API in the OS for making network connection

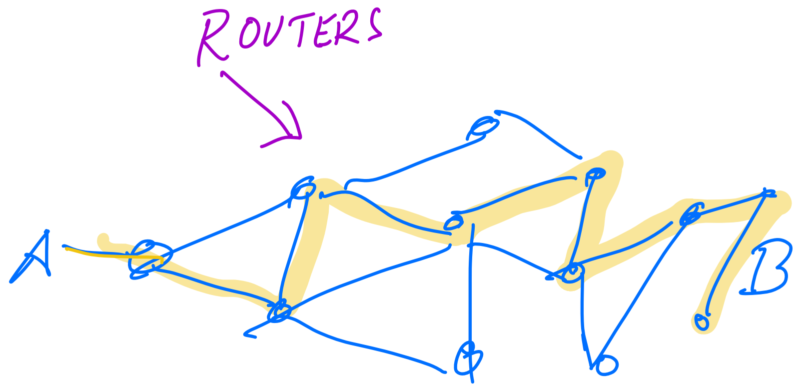
For an app, creates a "pipe" to send/recv data to for from another endpoint
=> Think like a file descriptor

OS keeps track of which sockets belong to which app => multiplexing

L4
TRANSPORT

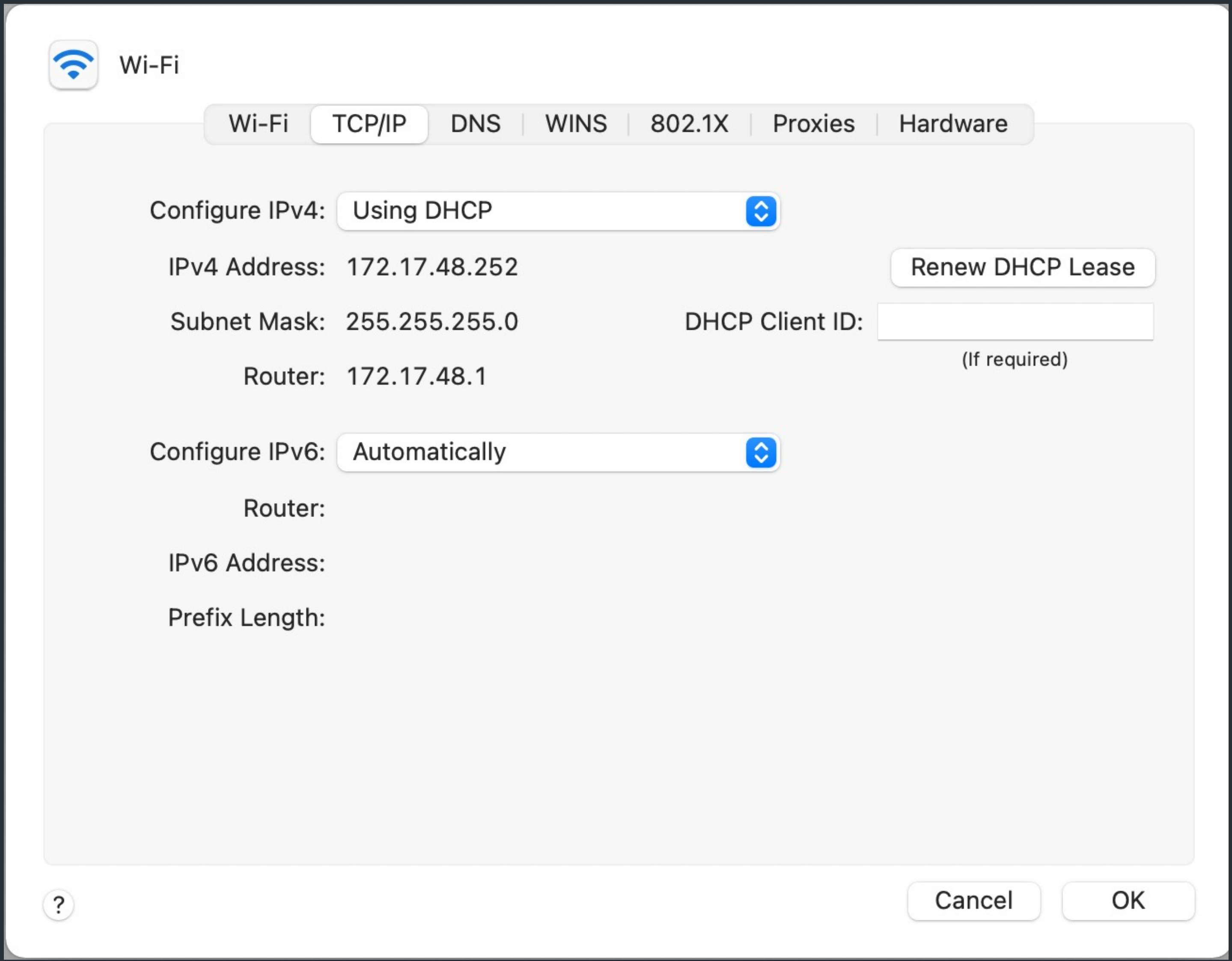


L3
NETWORK



148.128.34.110

128.148.x.x


Wi-Fi
Wi-Fi
TCP/IP
DNS
WINS
802.1X
Proxies
Hardware
Configure IPv4: Using DHCP
IPv4 Address: 172.17.48.252
Renew DHCP Lease
Subnet Mask: 255.255.255.0
DHCP Client ID:
(If required)
Router: 172.17.48.1
Configure IPv6: Automatically
Router:
IPv6 Address:
Prefix Length:
?
Cancel
OK

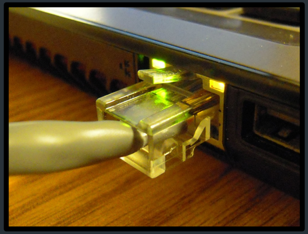
Lower layers

Link layer (L2): Individual links between nodes

=> Ethernet, wifi, cellular, …

Physical layer (L1): how to move bits over link

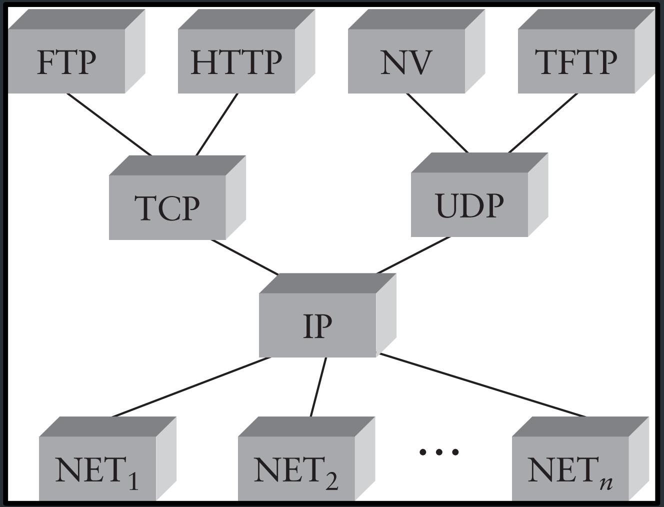
=> Engineering/physics problem



Examples
- Wifi
- Cellular Data
- Ethernet
- Fiber optic
- …

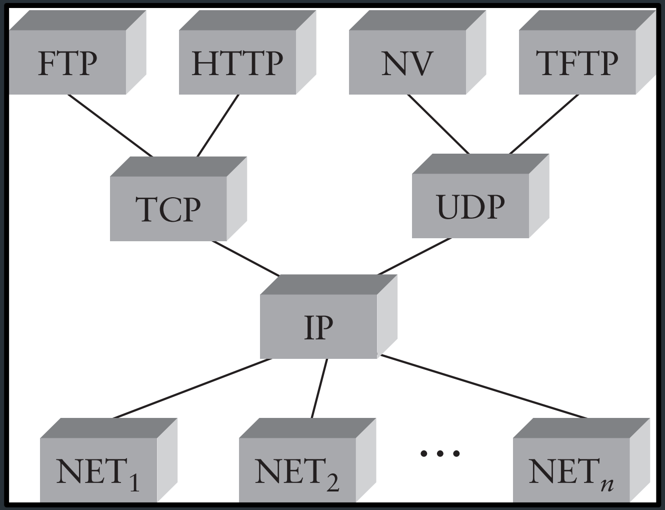
The OS sees links as interfaces
=> Each one probably has a driver that implements that particular protocol

IP as the "narrowing point"



- Applications built using IP
- IP connects many heterogeneous networks

IP as the "narrowing point"



- Applications built using IP
- IP connects many heterogeneous networks

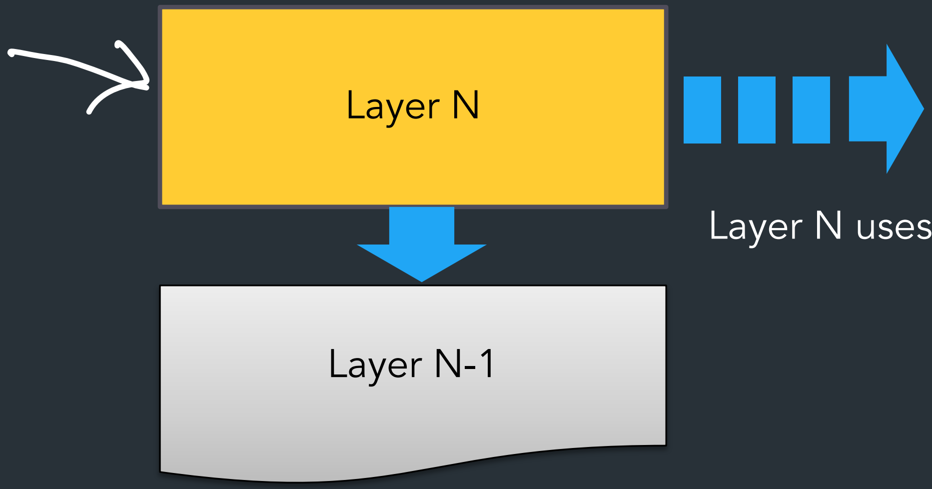
"Hourglass" structure => one (actually two) core abstractions!

What you should take away from this

Each layer is defined by some protocol

IP: IPv4, IPv6

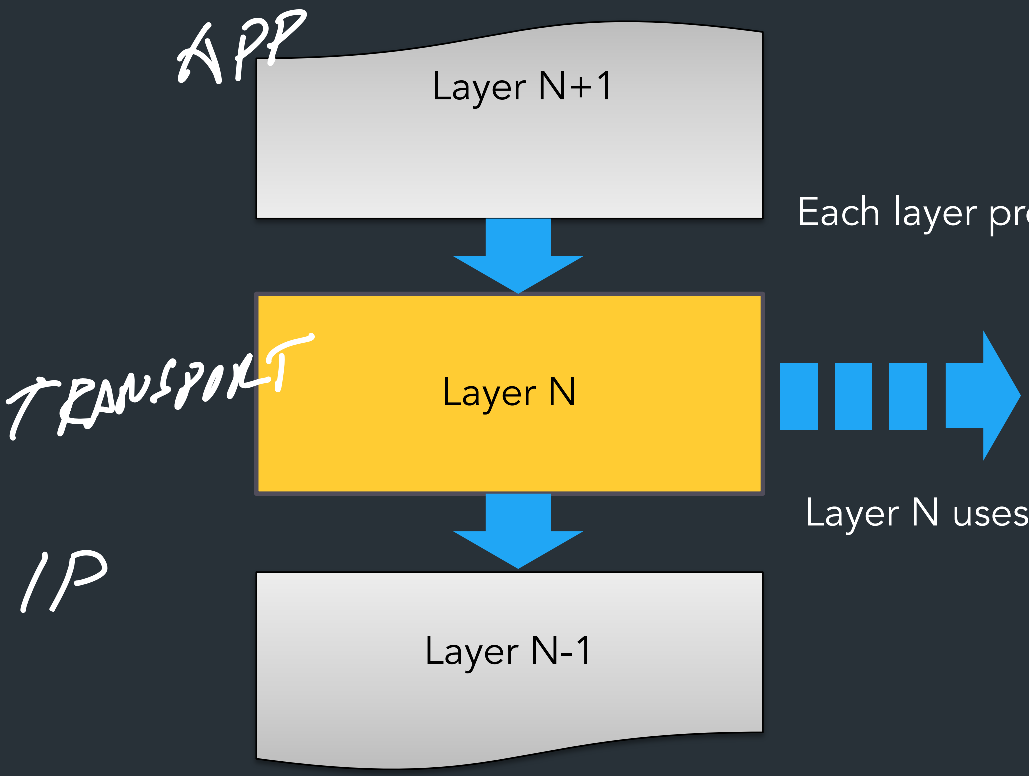
What you should take away from this



Each layer is defined by some protocol

Layer N uses the services provided by N-1 to operate

What you should take away from this



Each layer provides a service for the layers “above” it

Each layer is defined by some protocol

Layer N uses the services provided by N-1 to operate

Why do we do this?

- Helps us manage complexity
- Different implementations at one “layer” use same interface
- Allows independent evolution

To recap

7. Application	Service: user-facing application. (eg. HTTP, SSH, …) Application-defined messages
5. Transport	Service: multiplexing applications TCP: Reliable byte stream UDP: Unreliable messages
3. Network	Service: move packets to any other node in the network IPv4, IPv6 => (Unreliable)
2. Link	Service: move frames to other node across link. (eg. Ethernet, Wifi, …)
1. Physical	Service: move bits to other node across link (Electrical engineering problem)

Where do we handle, eg, security, reliability, fairness?

How/where to handle challenges?

Can decide on how to distribute certain problems

- What services at which layer?
- What to leave out?
- More on this later ("End-to-end principle")

How/where to handle challenges?

Can decide on how to distribute certain problems

- What services at which layer?
- What to leave out?
- More on this later ("End-to-end principle")

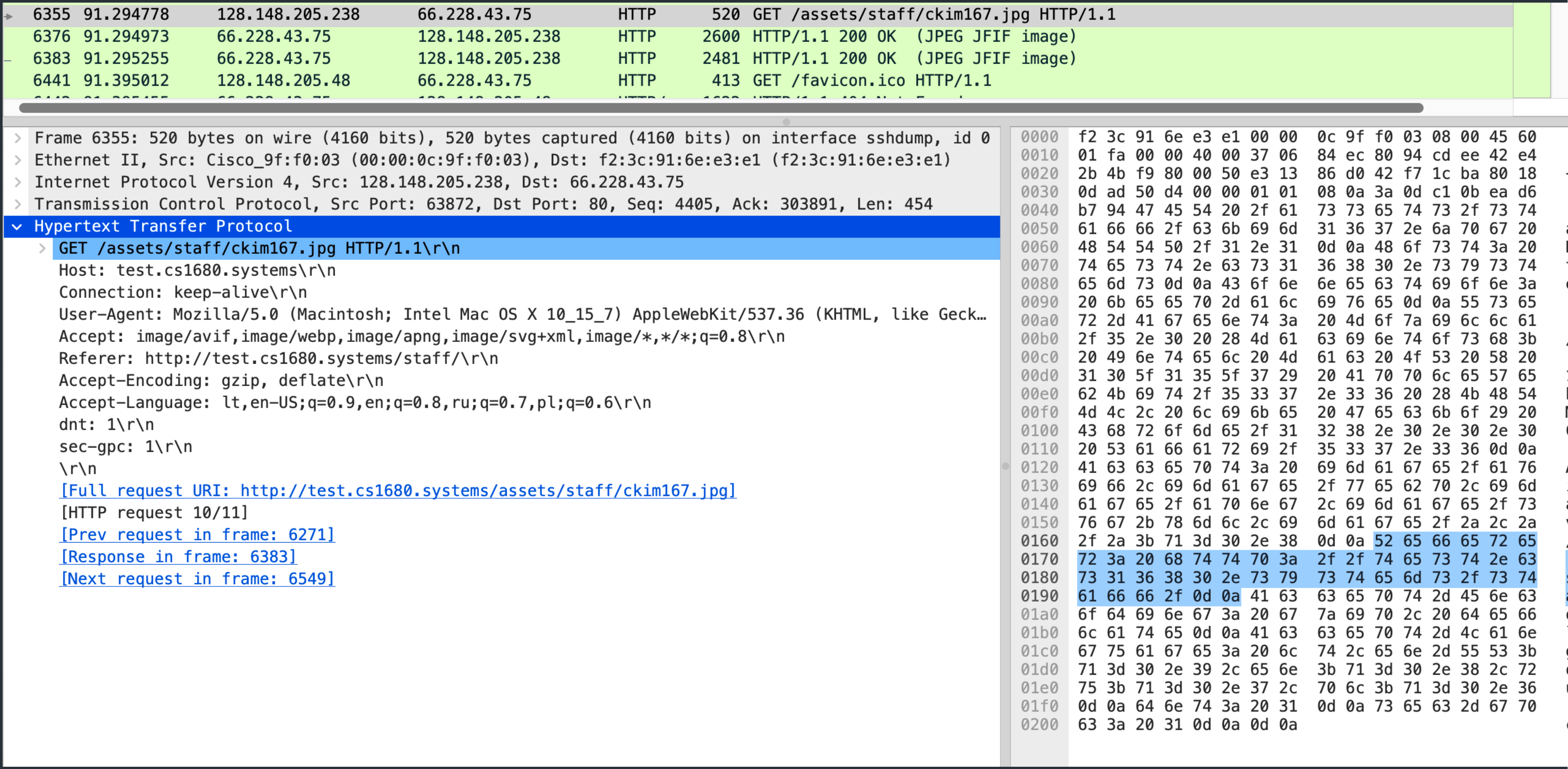
Example: Why bother having (unreliable) UDP, when TCP provides a reliable way to send data?

Get to decide where (and if) to pay the "cost" of certain features

Anatomy of a packet

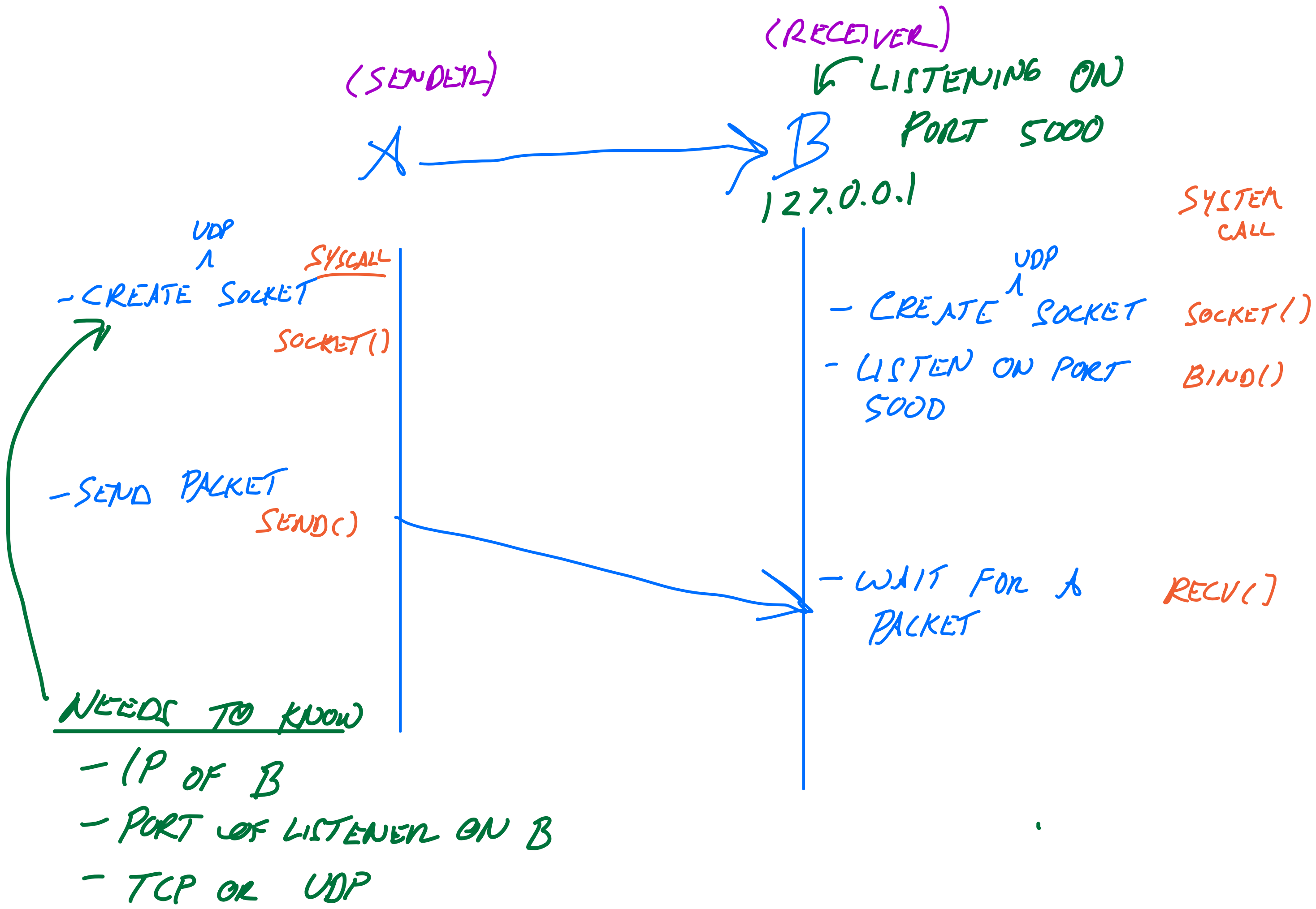
```
> Frame 100: 452 bytes on wire (3616 bits), 452 bytes captured (3616 bits) on interface en0, id 0
> Ethernet II, Src: Apple_15:8e:b8 (f0:18:98:15:8e:b8), Dst: Cisco_c5:2c:a3 (f8:c2:88:c5:2c:a3)
> Internet Protocol Version 4, Src: 172.17.48.252, Dst: 128.148.32.12
> Transmission Control Protocol, Src Port: 52725, Dst Port: 80, Seq: 1, Ack: 1, Len: 386
> Hypertext Transfer Protocol

0000  f8 c2 88 c5 2c a3 f0 18  98 15 8e b8 08 00 45 02   ····,··· ······E·
0010  01 b6 00 00 40 00 40 06  bb 92 ac 11 30 fc 80 94   ····@·@· ····0···
0020  20 0c cd f5 00 50 f1 b0  89 57 ae 46 0c d9 80 18    ····P·· ·W·F····
0030  08 02 b2 50 00 00 01 01  08 0a 36 da 1f 03 69 c9   ···P···· ··6···i·
0040  85 22 47 45 54 20 2f 20  48 54 54 50 2f 31 2e 31   ·"GET /  HTTP/1.1
0050  0d 0a 48 6f 73 74 3a 20  63 73 2e 62 72 6f 77 6e   ··Host:  cs.brown
0060  2e 65 64 75 0d 0a 55 73  65 72 2d 41 67 65 6e 74   .edu··Us er-Agent
0070  3a 20 4d 6f 7a 69 6c 6c  61 2f 35 2e 30 20 28 4d   : Mozill a/5.0 (M
```


6355 91.294778 128.148.205.238 66.228.43.75 HTTP 520 GET /assets/staff/ckim167.jpg HTTP/1.1
6376 91.294973 66.228.43.75 128.148.205.238 HTTP 2600 HTTP/1.1 200 OK (JPEG JFIF image)
6383 91.295255 66.228.43.75 128.148.205.238 HTTP 2481 HTTP/1.1 200 OK (JPEG JFIF image)
6441 91.395012 128.148.205.48 66.228.43.75 HTTP 413 GET /favicon.ico HTTP/1.1
Frame 6355: 520 bytes on wire (4160 bits), 520 bytes captured (4160 bits) on interface sshdump, id 0
Ethernet II, Src: Cisco_9f:f0:03 (00:00:0c:9f:f0:03), Dst: f2:3c:91:6e:e3:e1 (f2:3c:91:6e:e3:e1)
Internet Protocol Version 4, Src: 128.148.205.238, Dst: 66.228.43.75
Transmission Control Protocol, Src Port: 63872, Dst Port: 80, Seq: 4405, Ack: 303891, Len: 454
Hypertext Transfer Protocol
GET /assets/staff/ckim167.jpg HTTP/1.1\r\n
Host: test.cs1680.systems\r\n
Connection: keep-alive\r\n
User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Geck…
Accept: image/avif,image/webp,image/apng,image/svg+xml,image/*,*/*;q=0.8\r\n
Referer: http://test.cs1680.systems/staff/\r\n
Accept-Encoding: gzip, deflate\r\n
Accept-Language: lt,en-US;q=0.9,en;q=0.8,ru;q=0.7,pl;q=0.6\r\n
dnt: 1\r\n
sec-gpc: 1\r\n
\r\n
[Full request URI: http://test.cs1680.systems/assets/staff/ckim167.jpg]
[HTTP request 10/11]
[Prev request in frame: 6271]
[Response in frame: 6383]
[Next request in frame: 6549]
0000 f2 3c 91 6e e3 e1 00 00 0c 9f f0 03 08 00 45 60
0010 01 fa 00 00 40 00 37 06 84 ec 80 94 cd ee 42 e4
0020 2b 4b f9 80 00 50 e3 13 86 d0 42 f7 1c ba 80 18
0030 0d ad 50 d4 00 00 01 01 08 0a 3a 0d c1 0b ea d6
0040 b7 94 47 45 54 20 2f 61 73 73 65 74 73 2f 73 74
0050 61 66 66 2f 63 6b 69 6d 31 36 37 2e 6a 70 67 20
0060 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 20
0070 74 65 73 74 2e 63 73 31 36 38 30 2e 73 79 73 74
0080 65 6d 73 0d 0a 43 6f 6e 6e 65 63 74 69 6f 6e 3a
0090 20 6b 65 65 70 2d 61 6c 69 76 65 0d 0a 55 73 65
00a0 72 2d 41 67 65 6e 74 3a 20 4d 6f 7a 69 6c 6c 61
00b0 2f 35 2e 30 20 28 4d 61 63 69 6e 74 6f 73 68 3b
00c0 20 49 6e 74 65 6c 20 4d 61 63 20 4f 53 20 58 20
00d0 31 30 5f 31 35 5f 37 29 20 41 70 70 6c 65 57 65
00e0 62 4b 69 74 2f 35 33 37 2e 33 36 20 28 4b 48 54
00f0 4d 4c 2c 20 6c 69 6b 65 20 47 65 63 6b 6f 29 20
0100 43 68 72 6f 6d 65 2f 31 32 38 2e 30 2e 30 2e 30
0110 20 53 61 66 61 72 69 2f 35 33 37 2e 33 36 0d 0a
0120 41 63 63 65 70 74 3a 20 69 6d 61 67 65 2f 61 76
0130 69 66 2c 69 6d 61 67 65 2f 77 65 62 70 2c 69 6d
0140 61 67 65 2f 61 70 6e 67 2c 69 6d 61 67 65 2f 73
0150 76 67 2b 78 6d 6c 2c 69 6d 61 67 65 2f 2a 2c 2a
0160 2f 2a 3b 71 3d 30 2e 38 0d 0a 52 65 66 65 72 65
0170 72 3a 20 68 74 74 70 3a 2f 2f 74 65 73 74 2e 63
0180 73 31 36 38 30 2e 73 79 73 74 65 6d 73 2f 73 74
0190 61 66 66 2f 0d 0a 41 63 63 65 70 74 2d 45 6e 63
01a0 6f 64 69 6e 67 3a 20 67 7a 69 70 2c 20 64 65 66
01b0 6c 61 74 65 0d 0a 41 63 63 65 70 74 2d 4c 61 6e
01c0 67 75 61 67 65 3a 20 6c 74 2c 65 6e 2d 55 53 3b
01d0 71 3d 30 2e 39 2c 65 6e 3b 71 3d 30 2e 38 2c 72
01e0 75 3b 71 3d 30 2e 37 2c 70 6c 3b 71 3d 30 2e 36
01f0 0d 0a 64 6e 74 3a 20 31 0d 0a 73 65 63 2d 67 70
0200 63 3a 20 31 0d 0a 0d 0a

Example: communicating via UDP

UDP EXAMPLE



UDP: Unreliable transport protocol
=> Just send the packets, doesn't care if they actually arrive
=> Sender won't know if the packet reached the destination

Extra content if you want to read more...

Transport: UDP and TCP

UDP and TCP: most popular protocols atop IP
  - Both use 16-bit *port* number & 32-bit IP address
  - Applications *bind* a port & receive traffic on that port
- UDP – User (unreliable) Datagram Protocol
  - Send *packets* to a port (… and not much else)
  - Sent packets may be dropped, reordered, even duplicated
- TCP – Transmission Control Protocol
  - Provides illusion of reliable 'pipe' or 'stream' between two processes anywhere on the network
  - Handles congestion and flow control

Uses of TCP

- Most applications use TCP
  - Easier to program (reliability is convenient)
  - Automatically avoids congestion (don’t need to worry about taking down the network
- Servers typically listen on well-known ports:
  - SSH: 22
  - SMTP (email): 25
  - HTTP (web): 80, 443

Uses of UDP

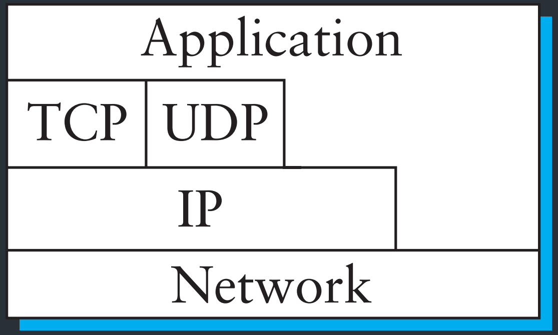
In general, when you have concerns other than a reliable "stream" of packets:

- When latency is critical (late messages don't matter)
- When messages fit in a single packet
- When you want to build your own (un)reliable protocol!

Examples

- DNS (port 53)
- Streaming multimedia/gaming (sometimes)

A note on layering



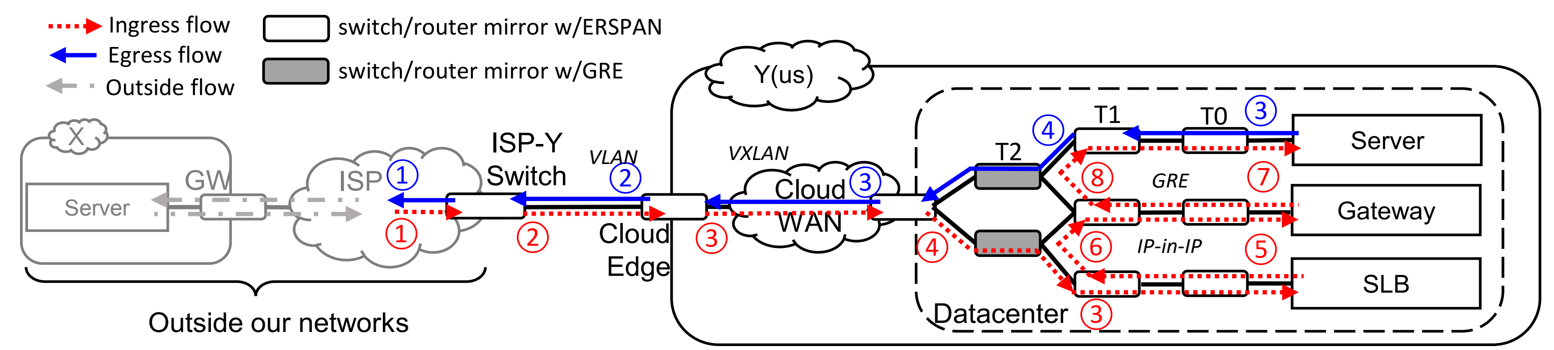
Strict layering not *required*

- TCP/UDP "cheat" to detect certain errors in IP-level information like address
- Overall, allows evolution, experimentation

One more thing…

- Layering defines interfaces well
  - What if I get an Ethernet frame, and send it as the payload of an IP packet across the world?
- Layering can be recursive
  - Each layer agnostic to payload!
- Many examples
  - Tunnels: e.g.,
    VXLAN is ETH over UDP (over IP over ETH again…)
  - Our IP assignment: IP on top of UDP "links"

Example



Number	Header Format										
	Headers Added after Mirroring			Mirrored Headers							
①	ETHERNET	IPV4	ERSPAN	ETHERNET						IPV4	TCP
②	ETHERNET	IPV4	ERSPAN	ETHERNET					802.1Q	IPV4	TCP
③	ETHERNET	IPV4	ERSPAN	ETHERNET		IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
④	ETHERNET	IPV4	GRE			IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑤	ETHERNET	IPV4	ERSPAN	ETHERNET	IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑥	ETHERNET	IPV4	GRE		IPV4	IPV4	UDP	VXLAN	ETHERNET	IPV4	TCP
⑦	ETHERNET	IPV4	ERSPAN	ETHERNET		IPV4	GRE		ETHERNET	IPV4	TCP
⑧	ETHERNET	IPV4	GRE			IPV4	GRE		ETHERNET	IPV4	TCP

* This is just an example, do not worry about the details, or the specific protocols!

From: Yu et al., A General, Easy to Program and Scalable Framework for Analyzing In-network Packet Traces, NSDI 2019

How do we use these protocols?

Using TCP/IP

How can applications use the network?

- *Sockets* API.
  - Originally from BSD, widely implemented (*BSD, Linux, Mac OS, Windows, …)
  - Important to know and do once
  - Higher-level APIs build on them

- After basic setup, it's a lot like working with files

Sockets: Communication Between Machines

- Network sockets are file descriptors too
- Datagram sockets (eg. UDP): unreliable message delivery
  - Send atomic messages, which may be reordered or lost
- Stream sockets (TCP): bi-directional pipes
  - *Stream* of bytes written on one end, read on another
  - Reads may not return full amount requested, must re-read

System calls for using TCP

Client	Server
	`socket` – make socket
	`bind` – assign address, port
	`listen` – listen for clients
`socket` – make socket	
`bind*` – assign address	
`connect` – connect to listening socket	
	`accept` – accept connection

- This call to bind is optional, connect can choose address & port.

Socket Naming

- TCP & UDP name *communication endpoints*
  - IP address specifies host (128.148.32.110)
  - 16-bit port number demultiplexes within host
  - Well-known services listen on standard ports (*e.g.* ssh – 22, http – 80, mail – 25)
  - Clients connect from arbitrary ports to well known ports
- A connection is named by 5 components
  - Protocol, local IP, local port, remote IP, remote port

Dealing with Data

- Many messages are binary data sent with precise formats

- Data usually sent in Network byte order (Big Endian)
  - Remember to always convert!
  - In C, this is htons(), htonl(), ntohs(), ntohl()