CSCI-1680
Layering and Encapsulation

Nick DeMarinis

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

Administrivia

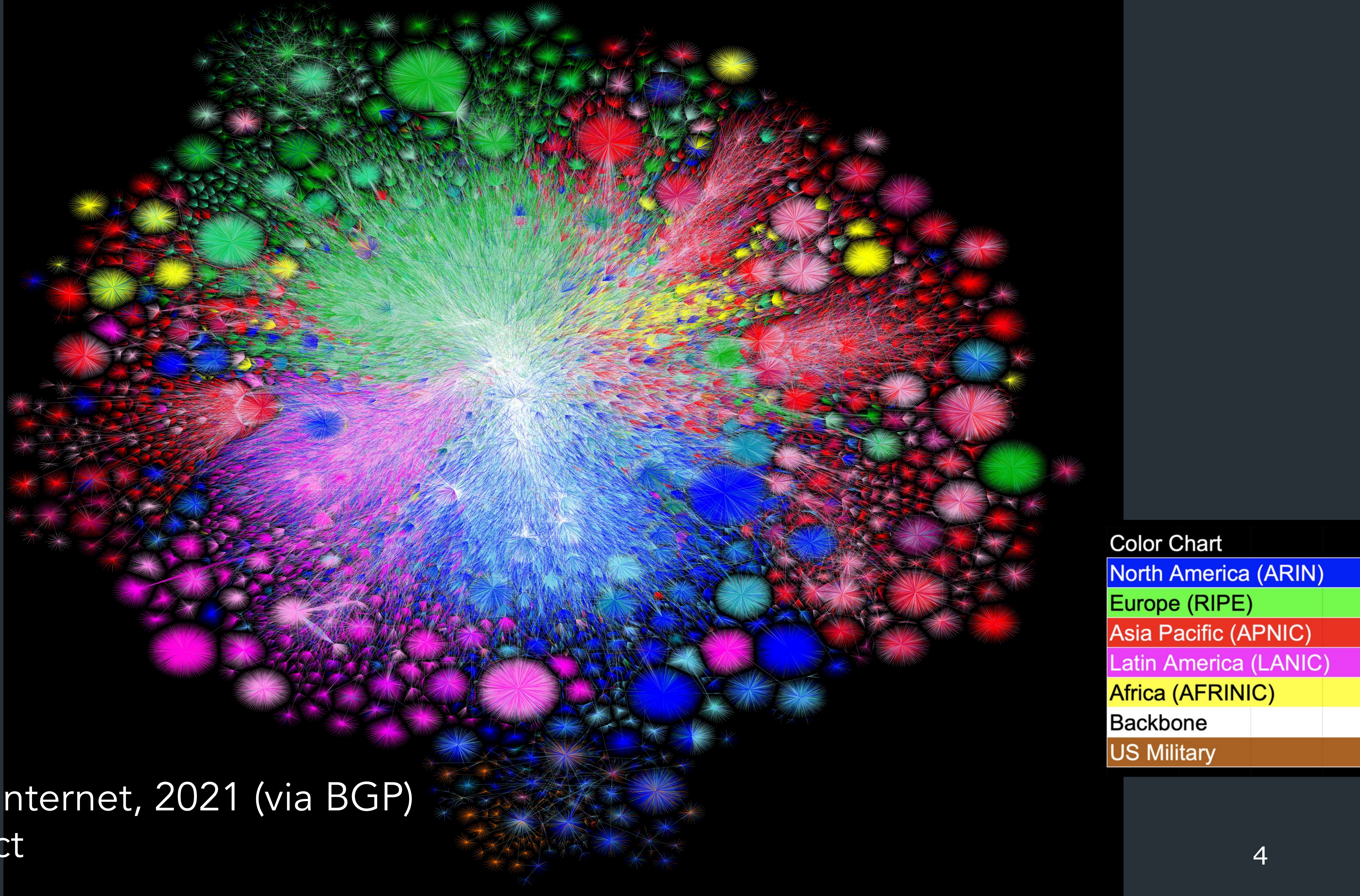
- HW0: Due TODAY by 11:59pm
- Container setup: due by Thursday
  - If you have issues, please fill out the form

- Snowcast out later today (look for Ed post)
  - Gearup Thursday 9/14 5-7pm CIT368 (+Zoom, recorded)

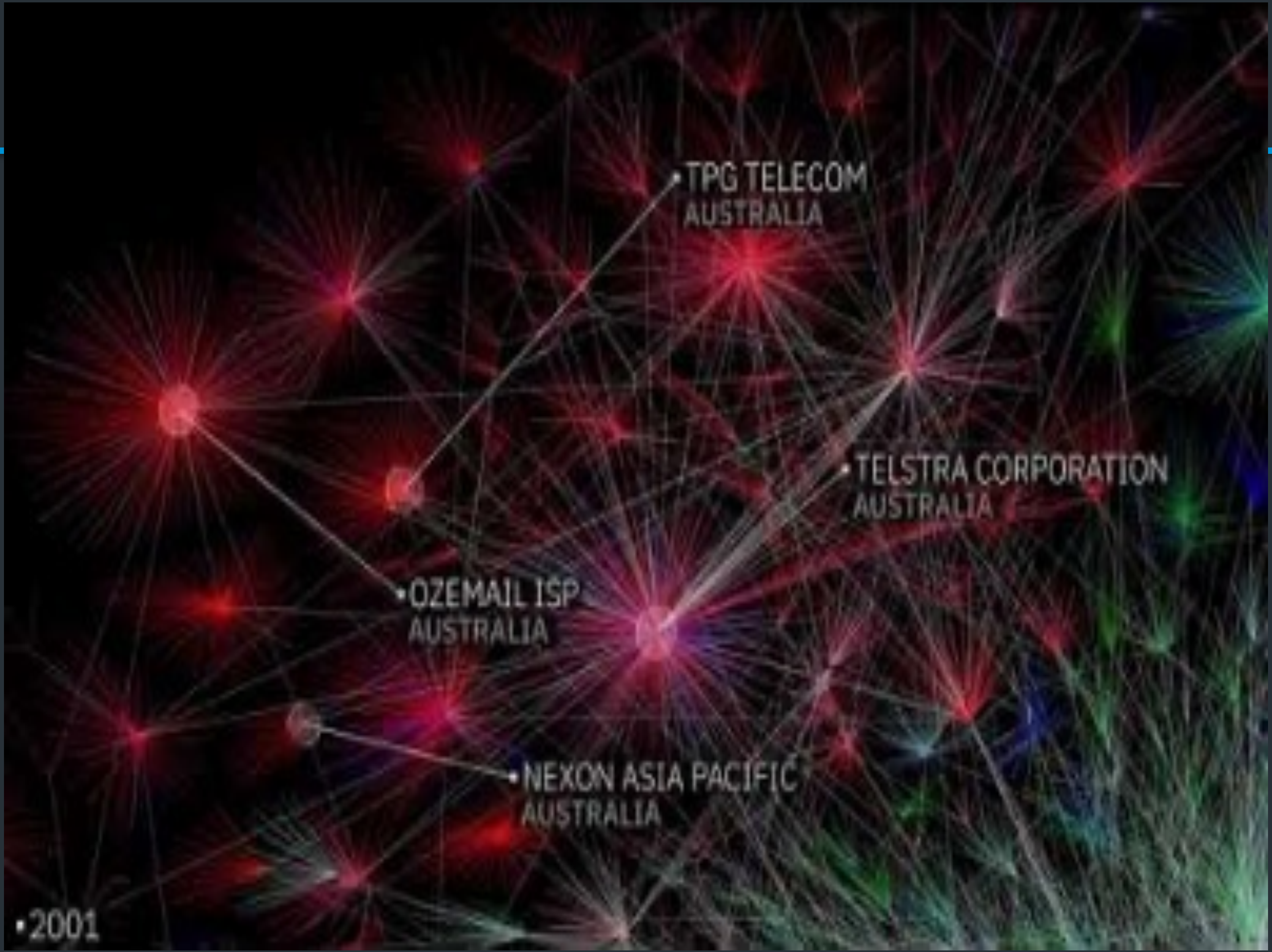
- Milestone due by Tuesday, 9/19 by 11:59pm EDT
  - Warmup and first steps + design doc for the rest

Topics for Today

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



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

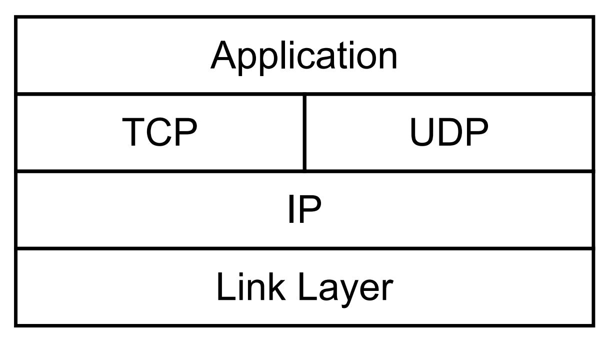


OPTE Internet map, 1997-2021: https://youtu.be/DdaElt6oP6w

How do we make sense of all this?

- *Very* large number of computers
- Incredible variety of technologies
  - Each with very different constraints
- Lots of *multiplexing*
- No single administrative entity
- Evolving demands, protocols, applications
  - Each with very different requirements!

Layering



Application

TCP | UDP

IP

Link Layer

Abstraction to the rescue!

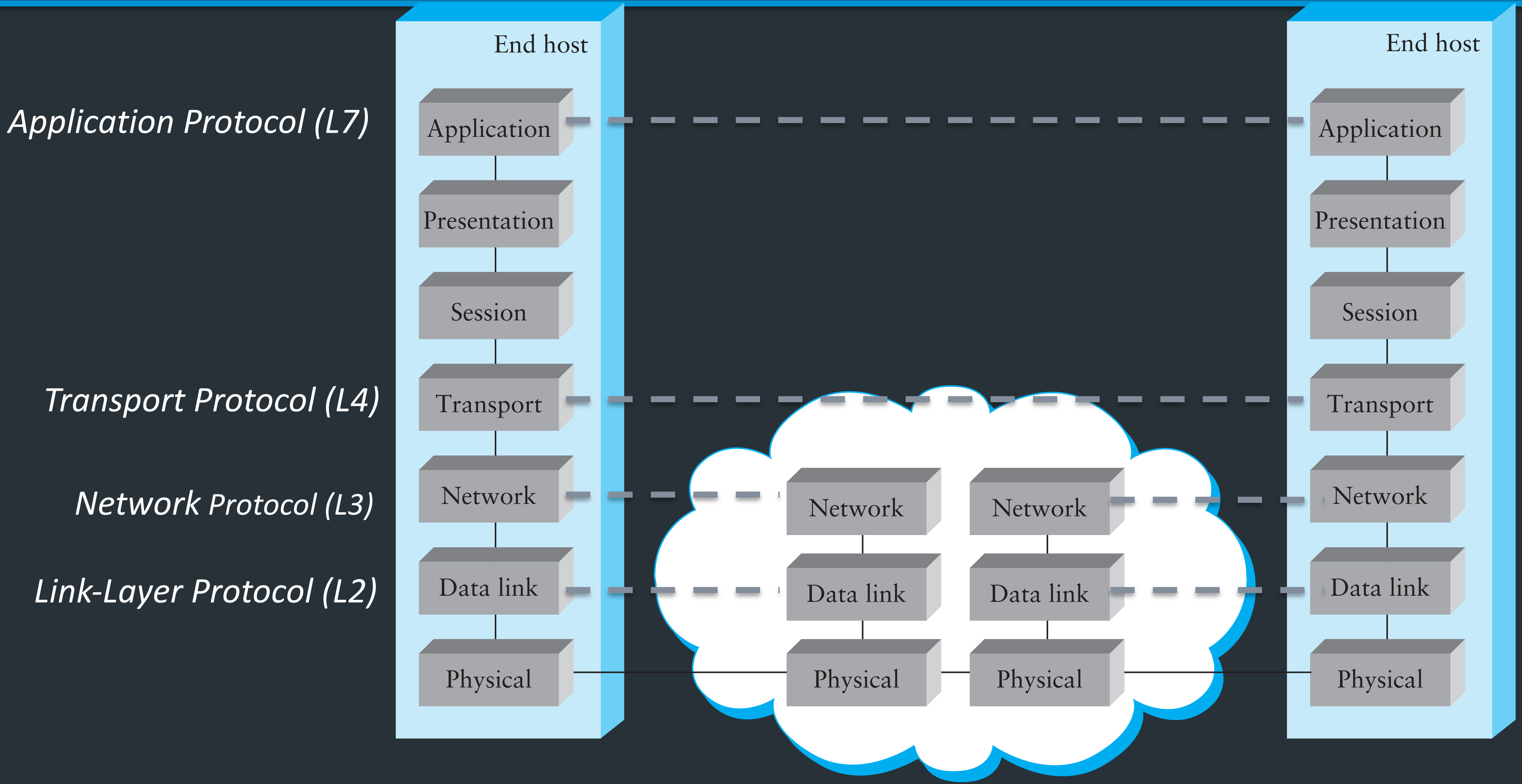
- Break problem into separate parts, solve part independently
- Abstract data from the layer above inside data from the layer below

Encapsulate data from "higher layer" inside "lower layer"
=> Lower layer can handle data without caring what's above it!

An analogy

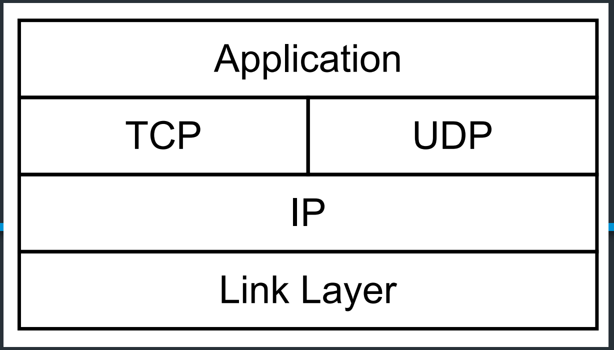
How to deliver a package?

The big complex picture



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

Applications (Layer 7)


Application

TCP | UDP

IP

Link Layer

The applicatons/programs/etc you use every day

Examples:

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

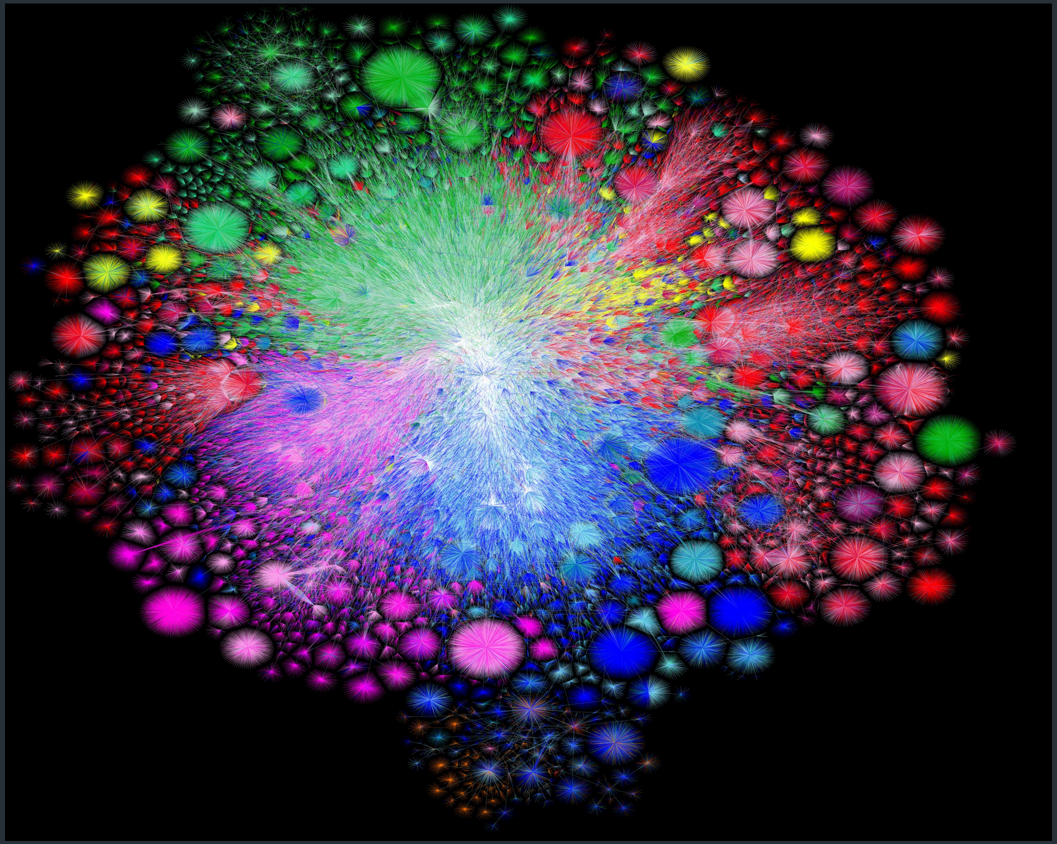
When you're building programs,
you usually work here

How to make apps use the network?

```
print(“Hello world”)
```

↓

```
send(“Hello world”)
```



How to make apps use the network?

```
print(“Hello world”)
```

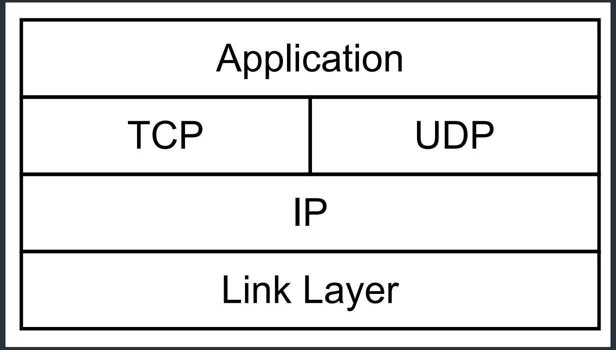
↓

```
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)



Application

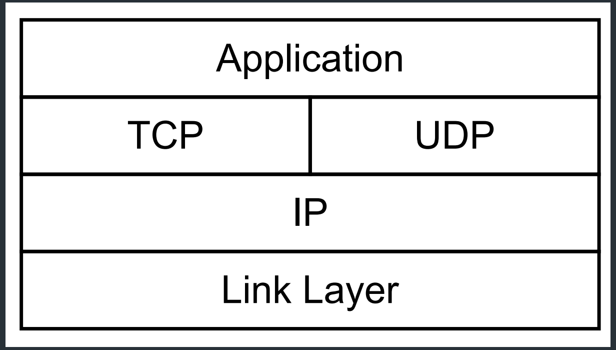
TCP | UDP

IP

Link Layer

- Generally provided by OS as socket interface
- For app, creates a "pipe" to send/recv data to/from another endpoint (think like a file descriptor)

Apps rely on: transport layer (layer 4)



Application

TCP | UDP

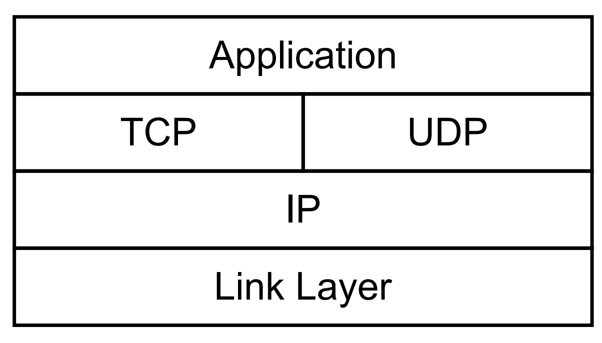
IP

Link Layer

- Generally provided by OS as socket interface
- For app, creates a "pipe" to send/recv data to/from another endpoint *(think like a file descriptor)*
- OS keeps track of sockets which sockets belong to which app => multiplexing

Key transport layer details for now

- Multiplexing provided by port numbers
  - 16-bit number 0—65535
  - Servers use well-known port numbers, clients typically choose one at random



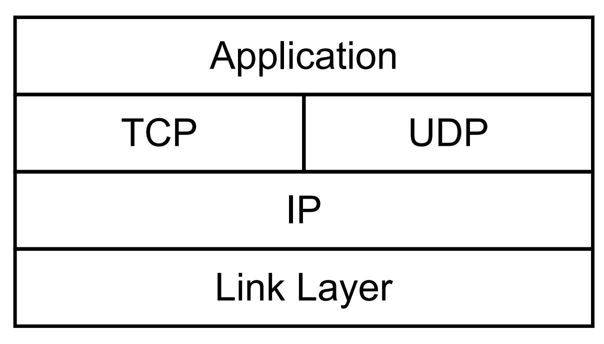
- Two main forms
  - TCP: reliable transport
  - UDP: unreliable transport

(more details later)

Port	Service
22	Secure Shell (SSH)
25	SMTP (Email)
80	HTTP (Web traffic)
443	HTTPS (Secure Web traffic)
16800	Snowcast

What service does the transport layer need?

Layer 3: Network layer



Application

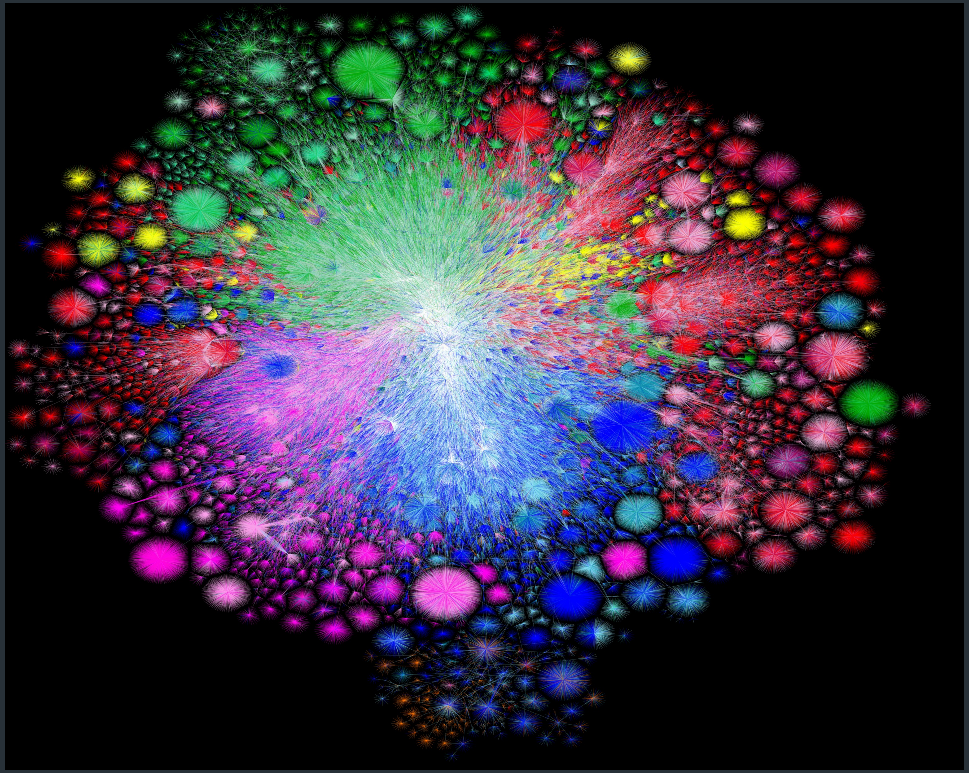
TCP | UDP

IP

Link Layer

Provided by: Internet Protocol (IP)

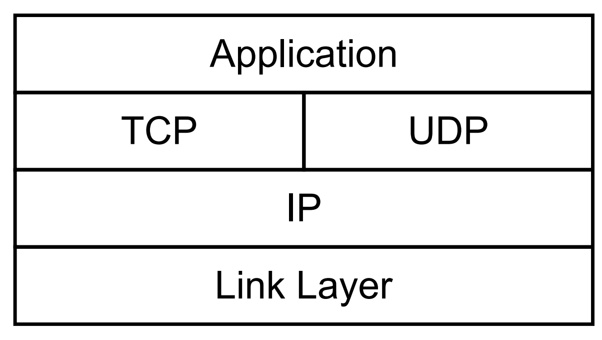
- Move packets between any two hosts anywhere on the Internet
- Responsible for *routing* and *forwarding* between nodes



- Every host has a unique address:

www.cs.brown.edu => 128.148.32.110

Layer 3: Network layer



Application

TCP | UDP

IP

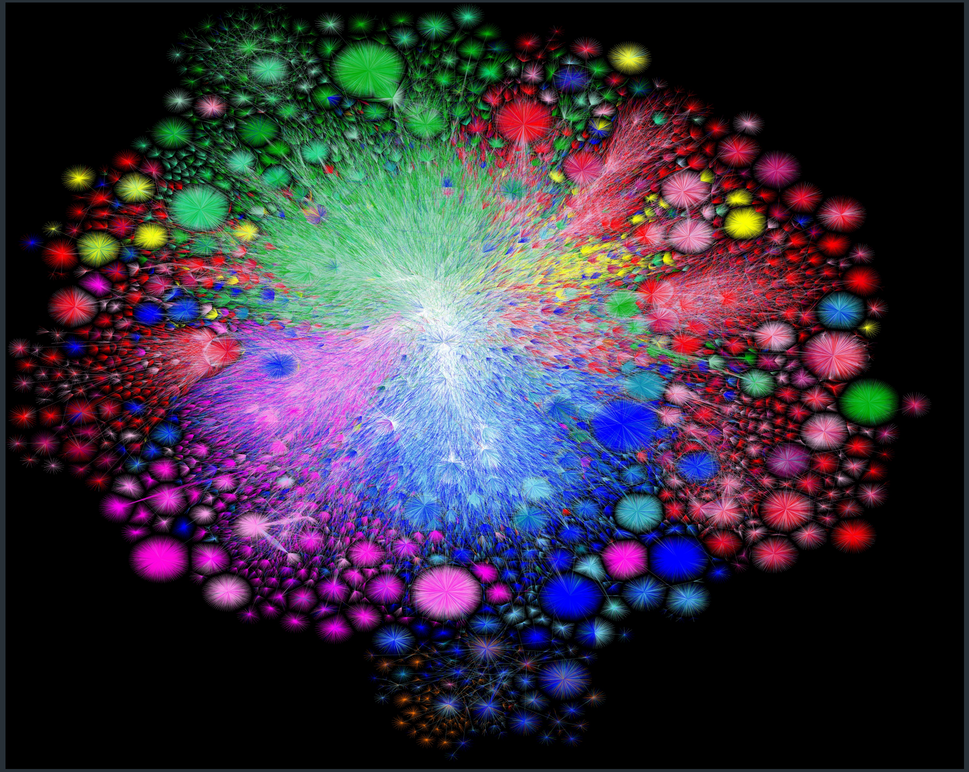
Link Layer

Provided by: Internet Protocol (IP)

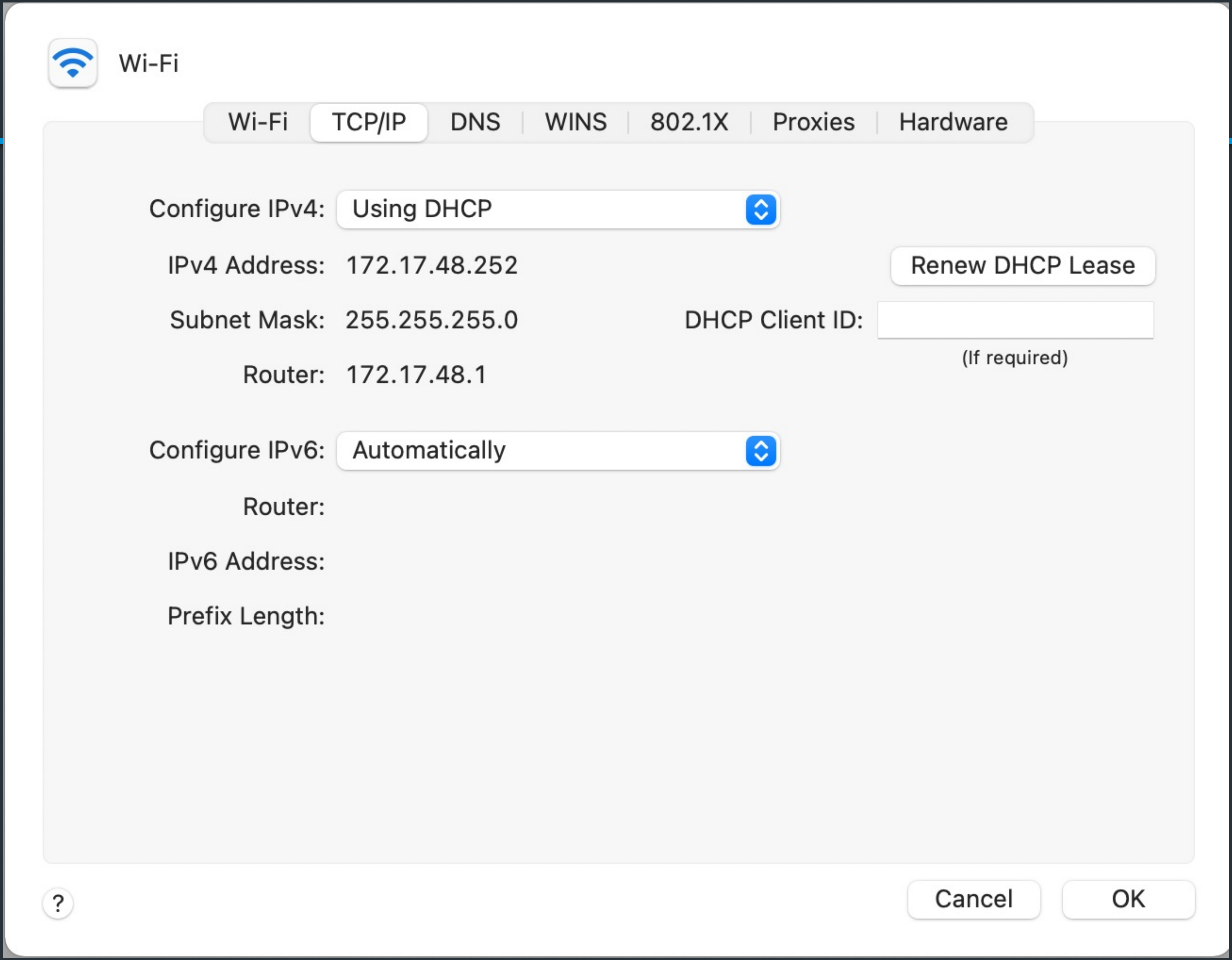
- Move packets between any two hosts anywhere on the Internet
- Responsible for *routing* and *forwarding* between nodes

- Every host has a unique address:

www.cs.brown.edu => 128.148.32.110



Given address, the network knows how to get the packet there



Wi-Fi

Configure IPv4: Using DHCP

IPv4 Address: 172.17.48.252

Subnet Mask: 255.255.255.0

Router: 172.17.48.1

DHCP Client ID:

(If required)

Configure IPv6: Automatically

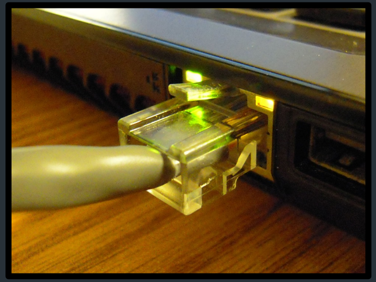
Router:

IPv6 Address:

Prefix Length:

Link layer (L2)

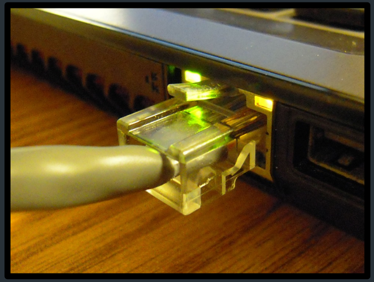
- Internet == Network of networks
- Networks are made up of many different types of links!
- Each type of link has its own challenges, protocols, etc depending on the medium



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

Link layer (L2)

- Internet == Network of networks
- Networks are made up of many different types of links!
- Each type of link has its own challenges, protocols, etc depending on the medium

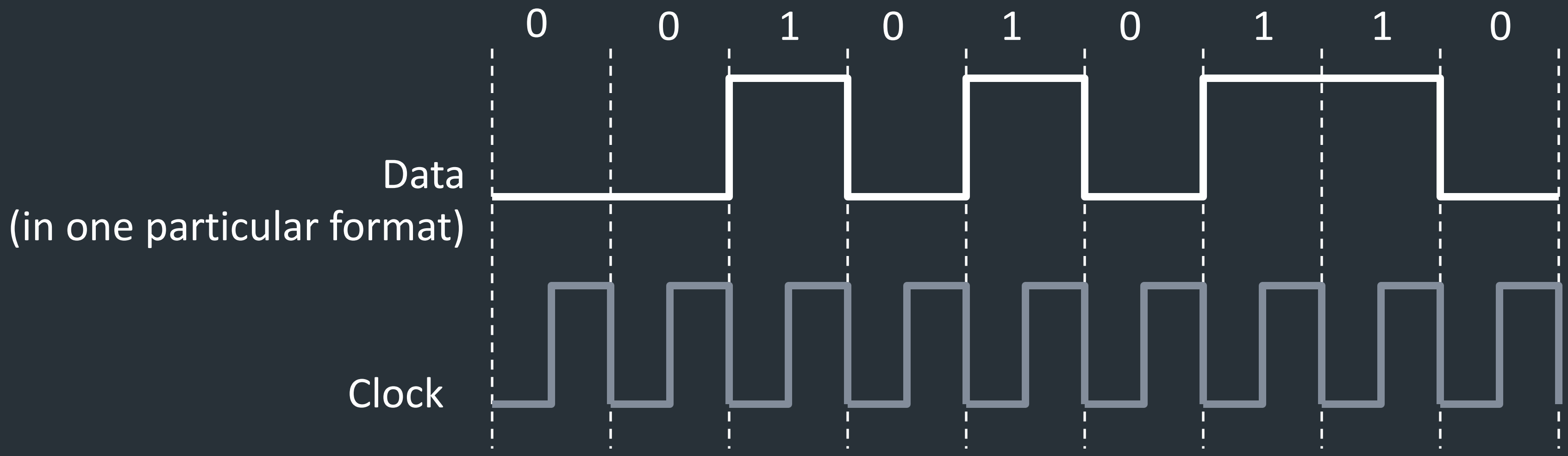


Examples

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

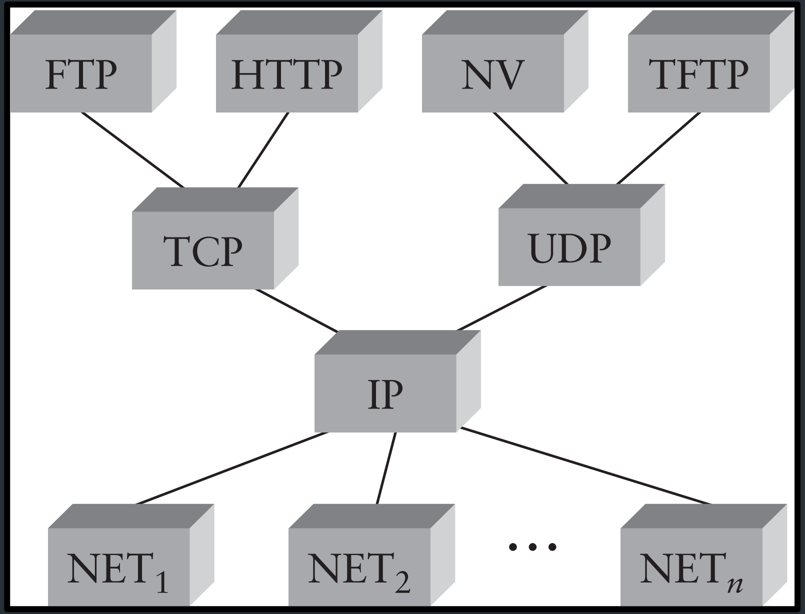
The OS sees links as interfaces
=> Each one probably has a driver that implements that particular protocol

Physical layer (Layer 1)



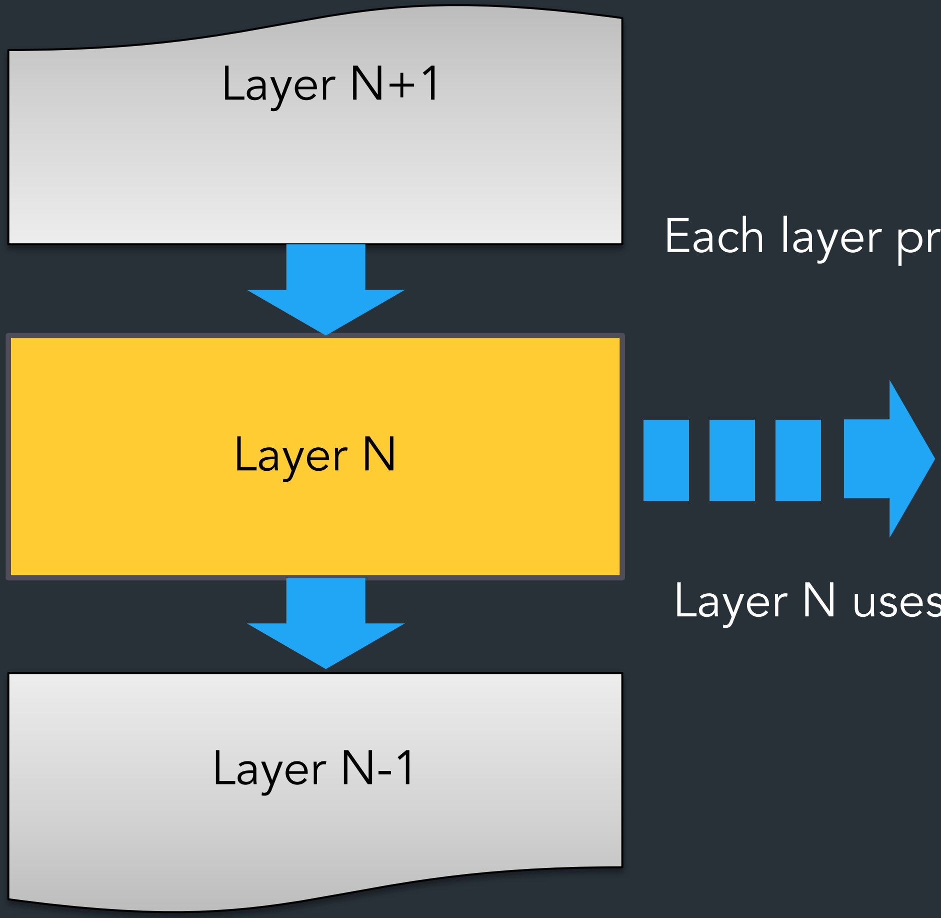
- How we move packets across one individual link
- Deals with individual bits
- More about electrical engineering/physics than computer science
- We'll talk about this briefly

IP: the "Narrow Waist"



- Applications built using IP; IP, Designed to connect many networks
- "Hourglass" structure => one (actually two) core abstractions!

What you should take away from this



Layer N+1

Layer N

Layer N-1

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
Reliable byte stream to other node (TCP),
Unreliable datagram (UDP)

3. Network

Service: move packets to any other node in the network
IP: Unreliable, best-effort service model

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)
- Example: reliability
  - IP offers pretty crappy service, even on top of reliable links… why?
  - TCP: offers reliable, in-order, no-duplicates service. Why would you want UDP?

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

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

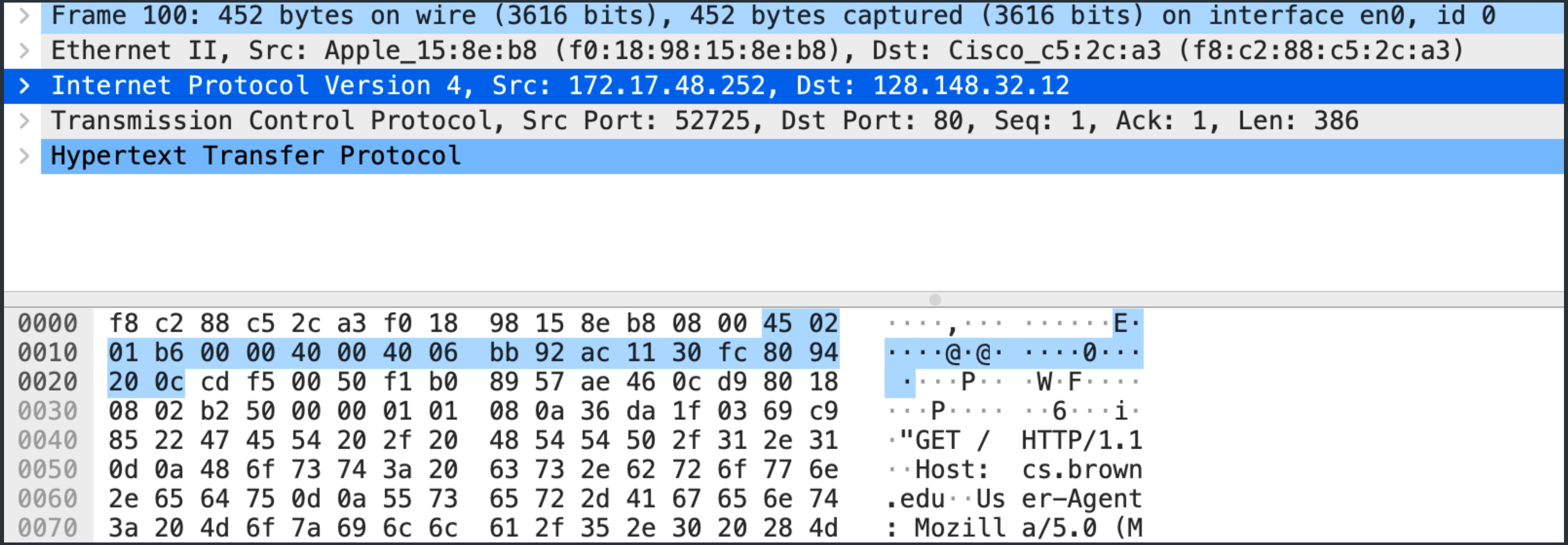
In general, when you have concerns other than a reliable "stream" of packets:

- 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)

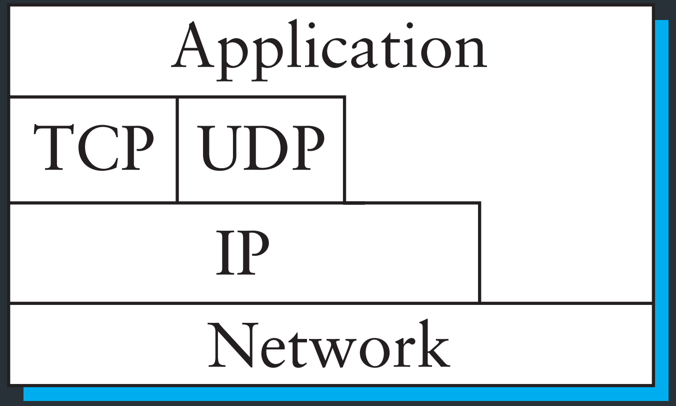
Anatomy of a packet



```
> 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
```

A note on layering



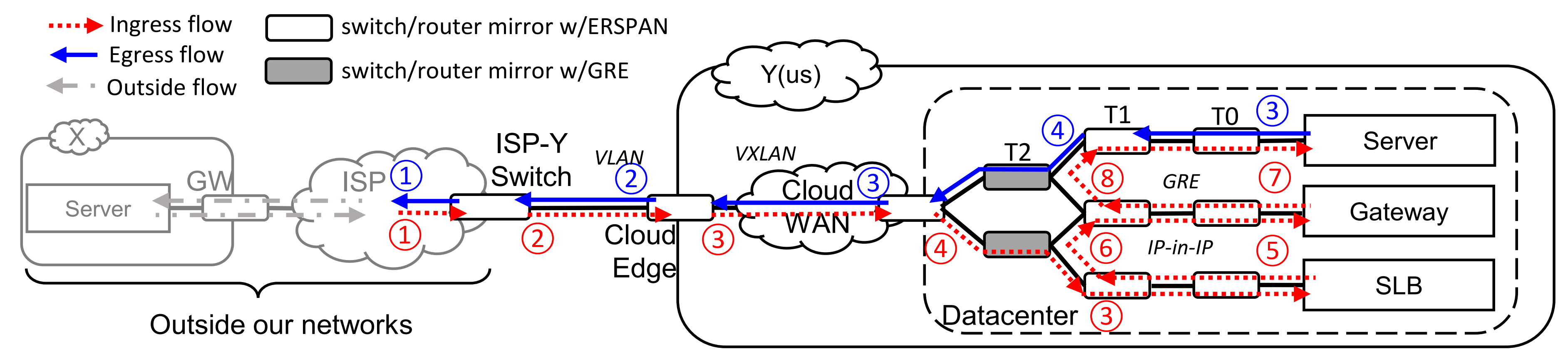
Strict layering not *required*

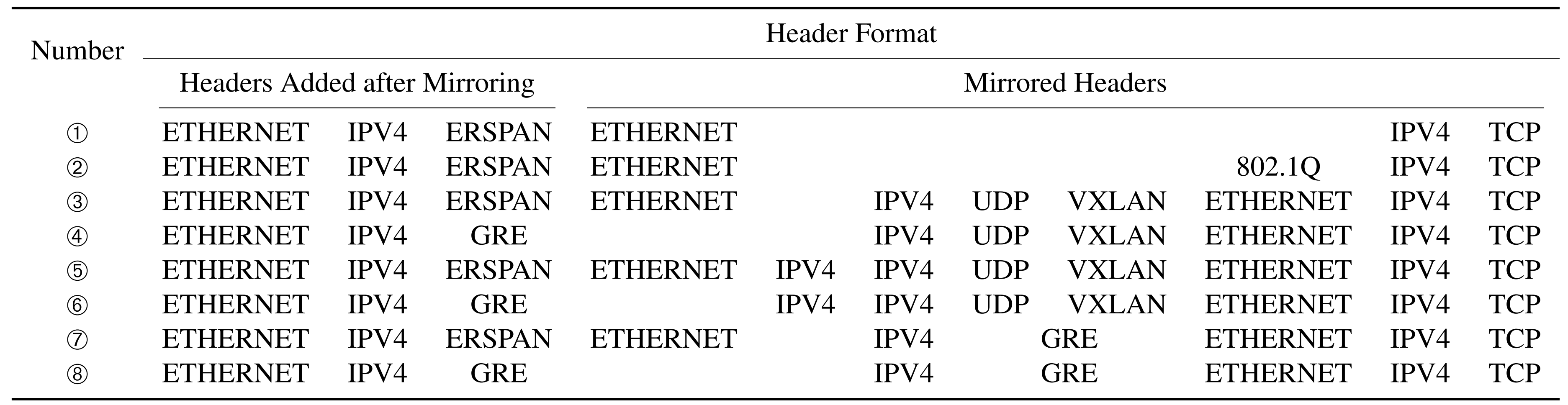
- TCP/UDP “cheat” to detect certain errors in IP-level information like address
- Overall, allows evolution, experimentation

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()