

# Networking

## CS 5007: Systems

Adrienne Slaughter, Joe Buck

Northeastern University

April 10, 2019

## 1 Knock Knock Example

- DNS

## 2 OSI Model

## 3 Networks and Sockets

## 4 Client

- IP Addresses
- DNS
- Create a socket

## 5 Server

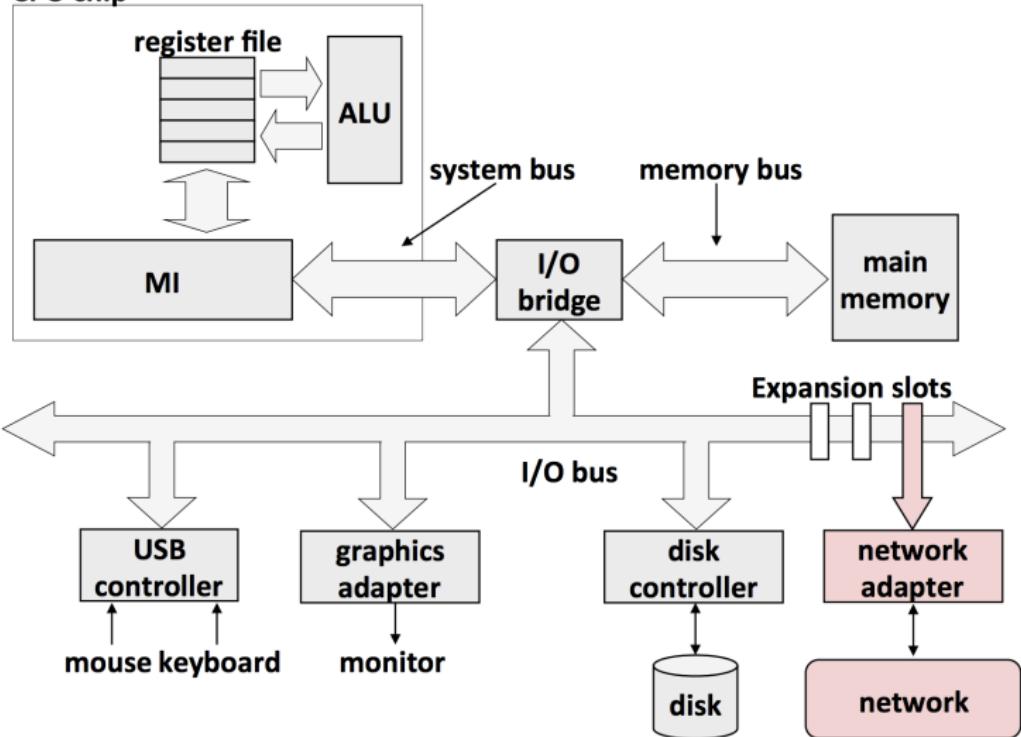
- Knock Knock Server
- Knock Knock Client

Ask me if I'm recording!

# Outline

- Introduction to networks
- A layered system: OSI Model
  - Physical layer
  - Data Link layer/ Link layer
  - Network layer
  - Transport layer
  - Session, Presentation layers (missing)
  - Application layer
- Network programming: POSIX sockets API
- Unix makes all I/O look like file I/O
  - “Just” read/write from a file descriptor
- Client/Server programming with TCP

## CPU chip



## 1 Knock Knock Example

- DNS

## 2 OSI Model

## 3 Networks and Sockets

## 4 Client

- IP Addresses
- DNS
- Create a socket

## 5 Server

- Knock Knock Server
- Knock Knock Client

# Goal: Communicate between two computers



Server

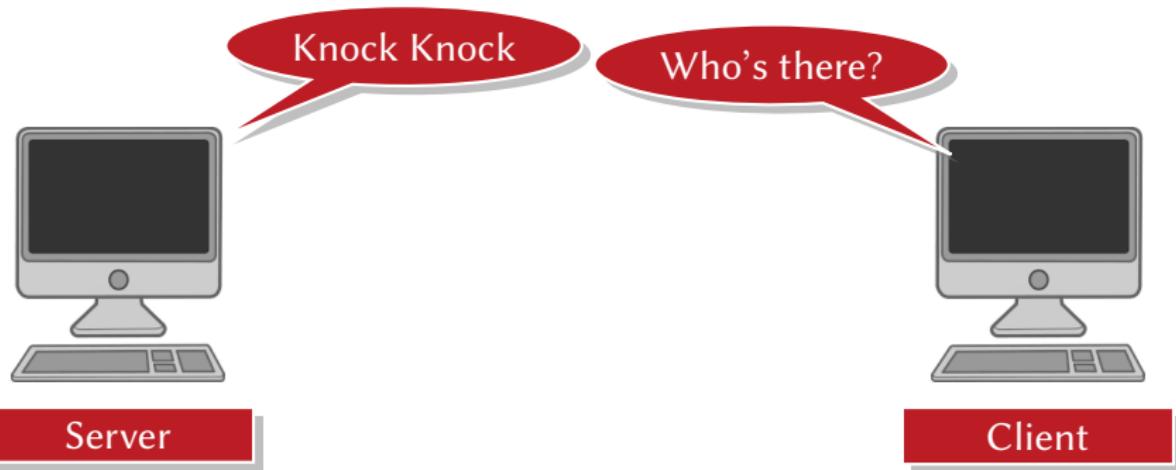


Client

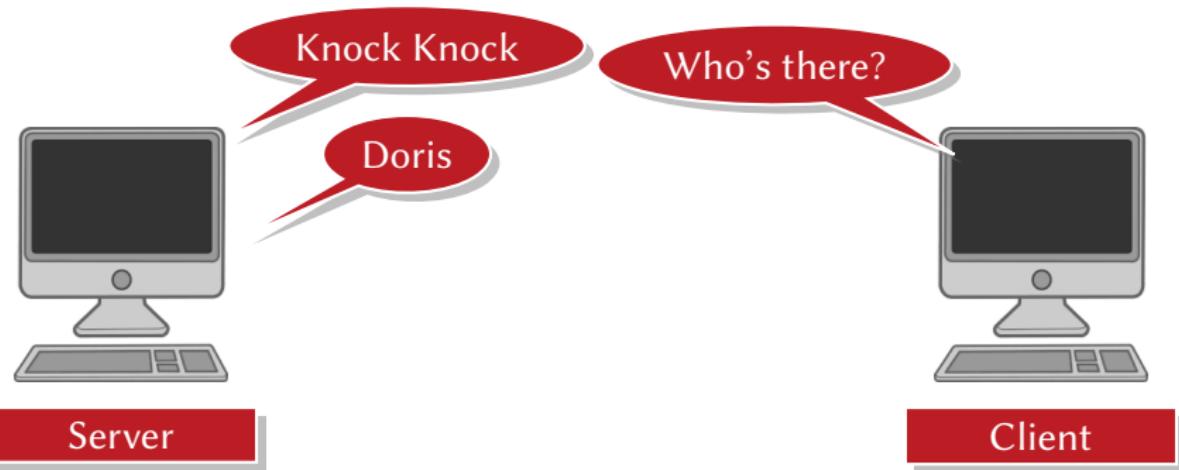
# Goal: Communicate between two computers



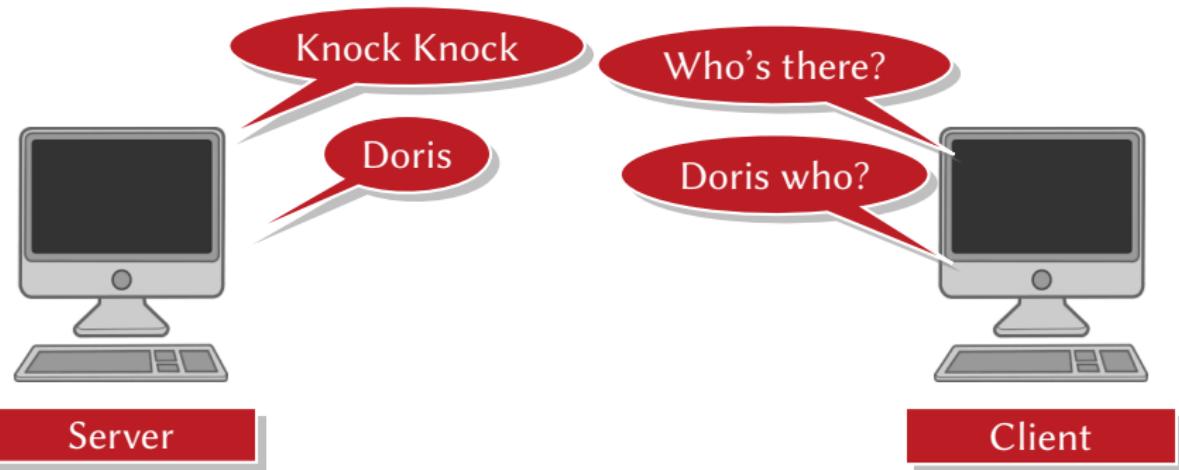
# Goal: Communicate between two computers



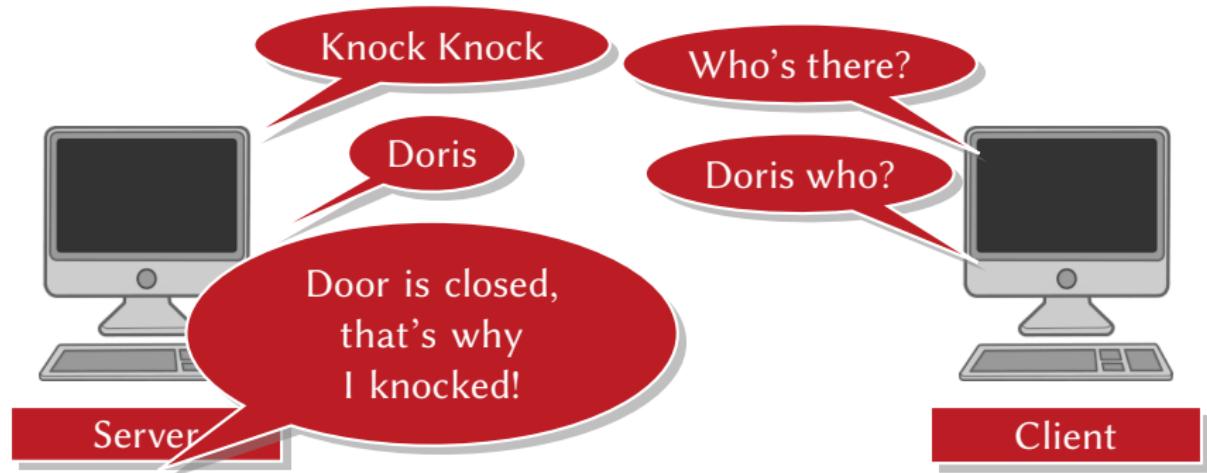
# Goal: Communicate between two computers



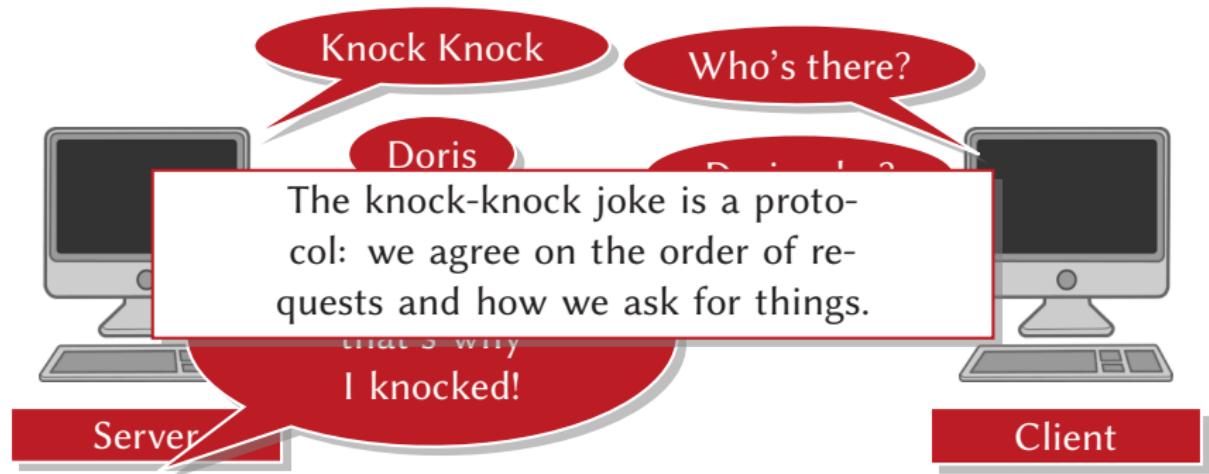
# Goal: Communicate between two computers



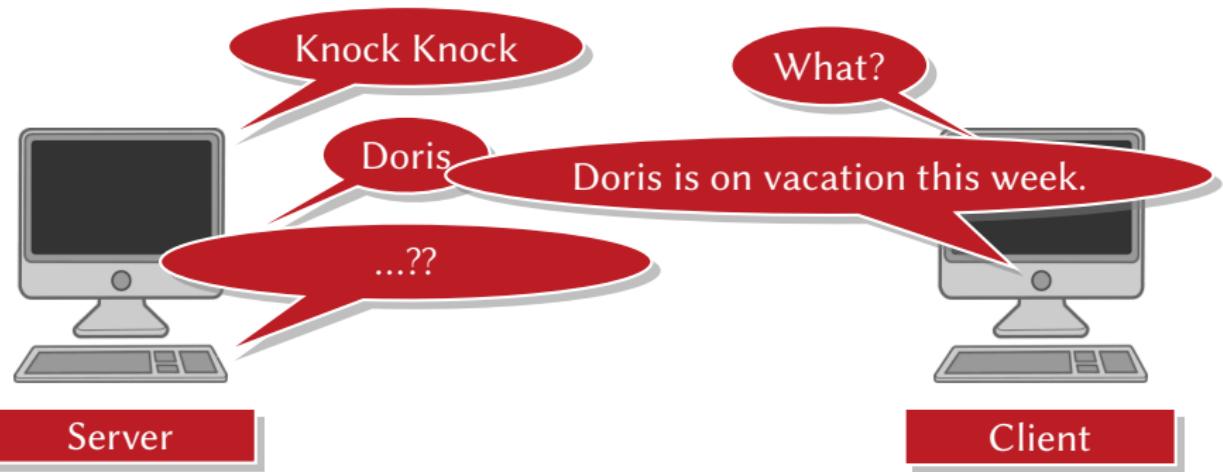
# Goal: Communicate between two computers



# Goal: Communicate between two computers



# Goal: Communicate between two computers



# Goal: Communicate between two computers

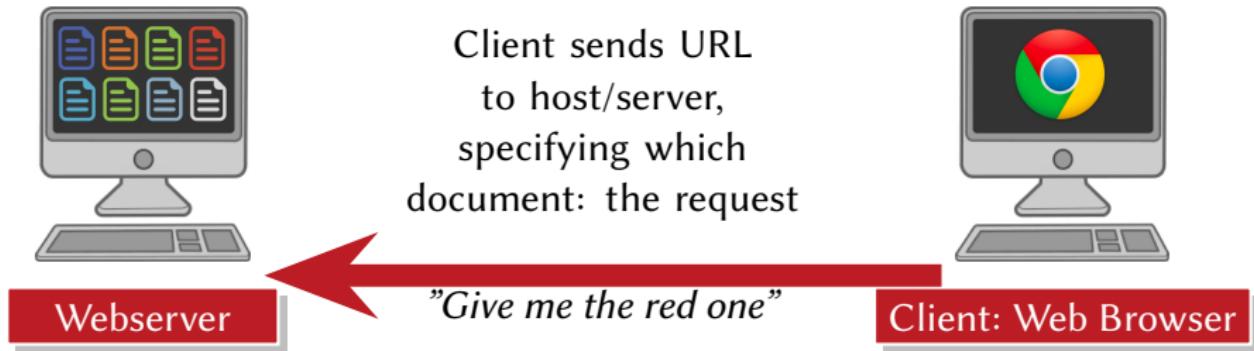


Webserver



Client: Web Browser

# Goal: Communicate between two computers



# Goal: Communicate between two computers



Webserver

Host sends file back  
to client, which  
is displayed in  
browser: the response



Client: Web Browser

*"Give me the red one"*



# Goal: Communicate between two computers



Webserver

This works because  
the server and client  
agree to use the  
same protocol: HTTP



Client: Web Browser

# HTTP

HyperText Transfer Protocol Consists of 2 basic messages: Request Response Each of the request/response consists of headers

# But how does the data get transferred?



Webserver

Application Data



Client: Web Browser

All that HTTP stuff is ***application data*** that 2 applications (the webserver and web browser) use to communicate).

# But how does the data get transferred?



Webserver

Application Data



Client: Web Browser

How do the machines actually connect to machines and transfer data?

# But how does the data get transferred?



Webserver

Application Data



Client: Web Browser

First, we open a **socket** on each computer.

# But how does the data get transferred?



Webserver

Application Data



Client: Web Browser

# But how does the data get transferred?



Webserver

TCP header Application Data



Client: Web Browser

The application data gets a  
TCP header added to it...

# But how does the data get transferred?



Webserver



Client: Web Browser

...and an IP header ...

# But how does the data get transferred?



Webserver

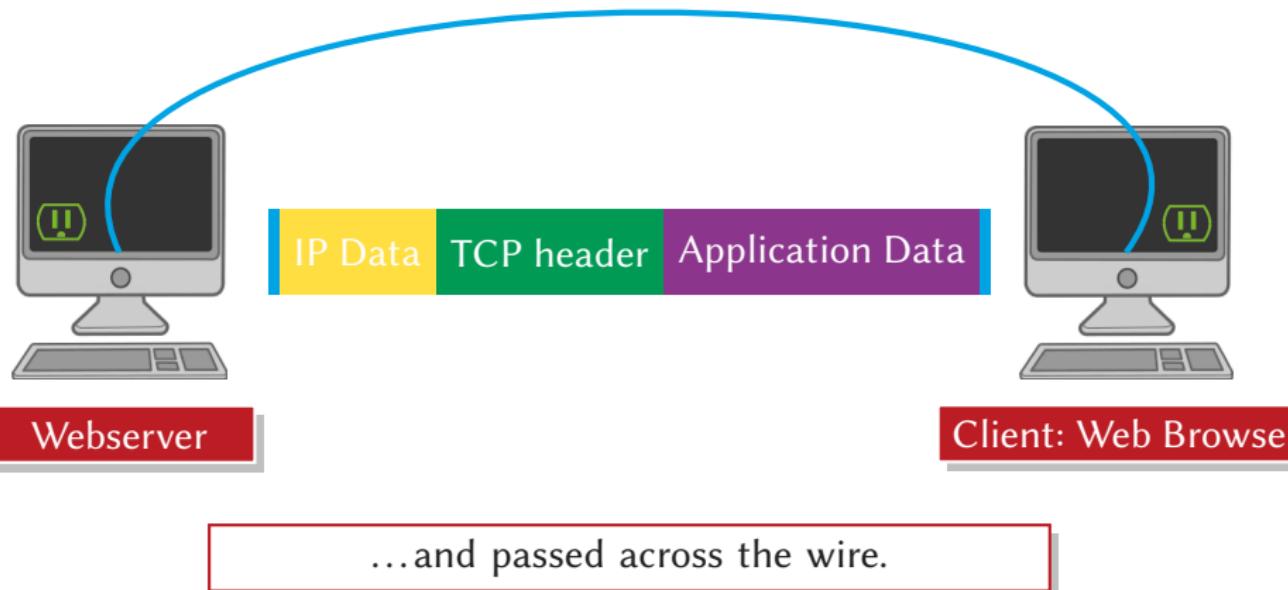
IP Data    TCP header    Application Data



Client: Web Browser

...wrapped with frame header/footer ...

# But how does the data get transferred?



# What do we need to make this work?

- Naming of network resources
  - How to specify which computer you want to connect to
- Sockets
  - How to allow your computer to talk directly to another computer
- Communication protocols
  - Agreeing on the communication

# Networking Concepts

- **Naming:** How to find the computer/host you want to connect to
- **Transfer:** The actual connection
- **Communicating:** Sending data back and forth in a way that both the client and host/server understand

# The General Process

- Find the address of the computer to connect to. (Naming)
- Create a socket. (Transfer)
- Read from and write to the stream according to the shared protocol. (Communication)
- Close the socket.

# Relevant Terminology

- Computer Network
  - hosts, routers, communication channels
- Host, Server: runs applications
- Routers: Forward information
- Packets: sequence of bytes
  - contains application data/message and control information, such as destination host.
- Protocol: an agreement
  - how to interpret packets
  - structure and size of packets
- Client
- Socket: abstraction through which an application may send and receive data
- Port
- DNS
- TCP/IP
- Session

- DNS: Domain Name System. Translates “<http://www.northeastern.edu>” into the “Internet Address”.
  - It’s the difference between going to “Ian’s House”, and the actual street address. When you ask DNS for the address to Ian’s House, it gives you the street address.
- TCP/IP: Transfer Control Protocol and Internet Protocol.
  - Used to break the application data into small pieces to be sent across the wire between the client and server. See the end of this lecture for more details.
- Session: A “conversation” between two computers.
  - Consider calling someone on the phone. When you call, you initiate the session. You and the person on the other end take turns talking, or exchanging dialog. When the two of you are done talking, you hang up, or close the session.

- URI: Uniform Resource Identifier
- URL: Uniform Resource Locator
- Often used interchangeably, but there is a difference:
  - URL is very specific: includes item (e.g. a specific file name) and protocol (how to get the item).
    - Example: `http://www.northeastern.edu/index.html`
  - URI can be less specific:
    - Example: `northeastern.edu`
    - Doesn't specify access (e.g., `ftp?` `http?`) or specific page (`index.html`).

# Anatomy of a URL

http://www.theimdbapi.org/api/movie?movie\_id=tt0089218

# Anatomy of a URL

http://www.theimdbapi.org/api/movie?movie\_id=tt0089218

Protocol

# Anatomy of a URL

http://www.theimdbapi.org/api/movie?movie\_id=tt0089218

Protocol Resource name

# Anatomy of a URL

http://www.theimdbapi.org/api/movie?movie\_id=tt0089218  
Protocol    Resource name                  Path

# Anatomy of a URL

`http://www.theimdbapi.org/api/movie?movie_id=tt0089218`

Protocol	Resource name	Path	Params
http	www.theimdbapi.org	api/movie	?movie_id=tt0089218

# Anatomy of a URL

http://www.themdbapi.org/api/movie?movie\_id=tt0089218  
Protocol    Resource name              Path              Params

Without protocol & resource name, we can't have a URL.

Path and parameters can be null.

# Anatomy of a URL

http://www.themdbapi.org/api/movie?movie\_id=tt0089218

Protocol    Resource name              Path              Params

- Hostname
- Filename
- Port Number
- Reference (optional)

Without protocol & resource name, we can't have a URL.

Path and parameters can be null.

# Anatomy of a URL

http://www.themdbapi.org/api/movie?movie\_id=tt0089218

Protocol   Resource name      Path      Params

- Hostname
- Filename
- Port Number
- Reference (optional)

All of this information allows a socket to be opened up.

But connecting only via URLs is pretty high level— a lot of abstraction is happening.

What if we want to define or use our own protocol?

⇒ We need to open a socket directly.

# DNS

People tend to use DNS names, not IP addresses

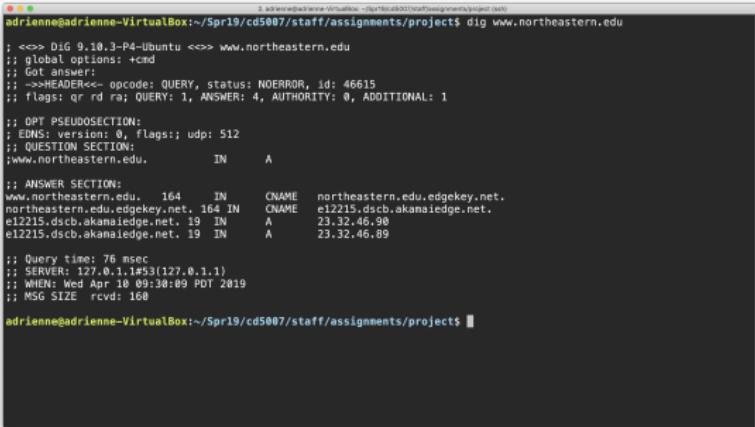
- the sockets API lets you convert between the two
- it's a complicated process, though:
  - a given DNS name can have many IP addresses
  - many different DNS names can map to the same IP address
    - an IP address will reverse map into at most one DNS name, and maybe none
  - a DNS lookup may require interacting with many DNS servers

You can use the `dig` Linux program to explore DNS

- `man dig`

# A fun thing to do:

dig let's you get DNS info on a particular domain:



```
adrienne@adrienne-VirtualBox:~/Spr19/cd5007/staff/assignments/project$ dig www.northeastern.edu

; <>> DIG 9.10.3-P4-Ubuntu <>> www.northeastern.edu
;; global options: +cmd
;; Got answer:
;; ->>>HEADER<- opcode: QUERY, status: NOERROR, id: 46615
;; flags: qr rd ra; QUERY: 1, ANSWER: 4, AUTHORITY: 0, ADDITIONAL: 1
;; OPT PSEUDOSECTION:
;; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
;www.northeastern.edu.           IN      A

;; ANSWER SECTION:
www.northeastern.edu.    164     IN      CNAME   northeastern.edu.edgekey.net.
northeastern.edu.edgekey.net. 164 IN  CNAME  e12215.dscb.akamaiedge.net.
e12215.dscb.akamaiedge.net. 19  IN      A      23.32.46.90
e12215.dscb.akamaiedge.net. 19  IN      A      23.32.46.89

;; Query time: 76 msec
;; SERVER: 127.0.1.1#53(127.0.1.1)
;; WHEN: Wed Apr 10 09:30:09 PDT 2019
;; MSG SIZE rcvd: 168

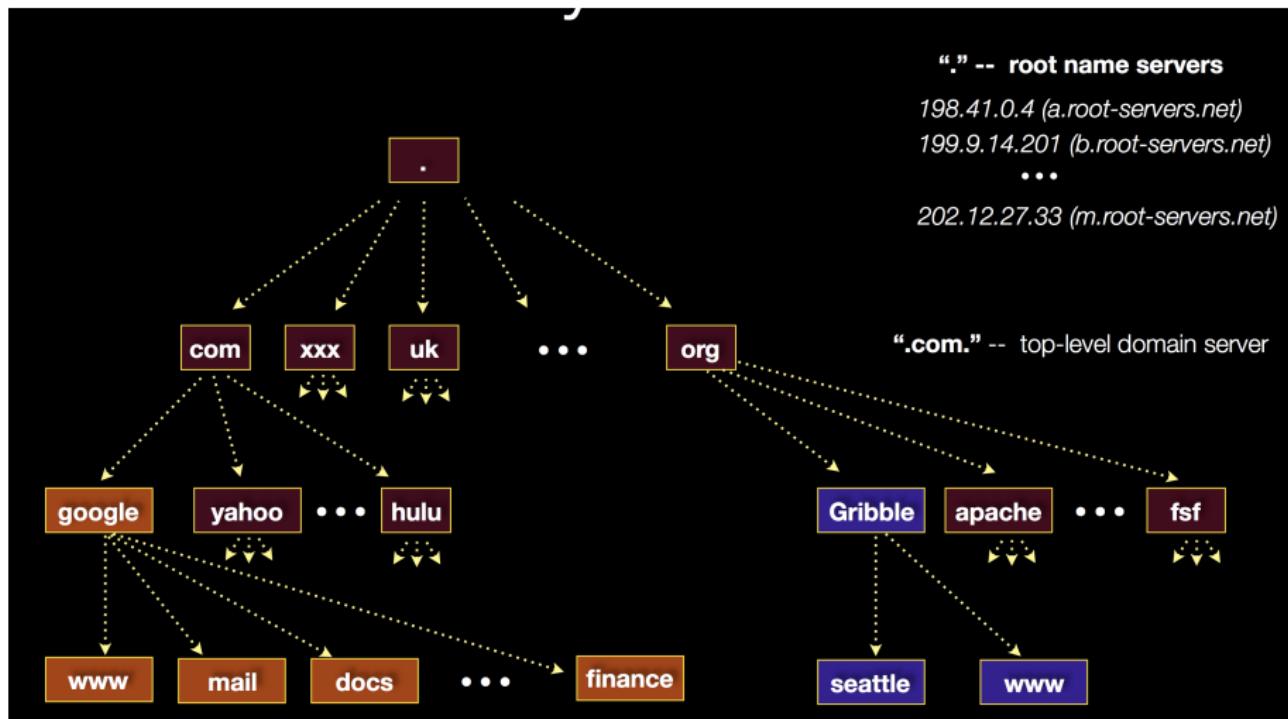
adrienne@adrienne-VirtualBox:~/Spr19/cd5007/staff/assignments/project$
```

This tells me that `www.northeastern.edu` maps to `northeastern.edu.edgekey.net`, which maps to `e12215.dscb.akamaiedge.net`. `e12215.dscb.akamaiedge.net` maps to two IP addresses, `23.32.46.90` and `23.32.46.89`.

I happen to know that Akamai is a service that helps provide reliability and speed for a domain (such as `www.northeastern.edu`). I take all this to mean that Akamai is providing the service to Northeastern, presumably to make sure the website is always available. Akamai is utilizing two different servers to

handle the load of requests for `www.northeastern.edu`.

# DNS Hierarchy



# Resolving DNS names

The POSIX way is to use `getaddrinfo( )`

- a pretty complicated system call; the basic idea...
  - set up a “hints” structure with constraints you want respected
    - e.g., IPv6, IPv4, or either
  - tell `getaddrinfo( )` which host and port you want resolved
    - host: a string representation; DNS name or IP address
  - `getaddrinfo( )` gives you a list of results packet in an `addrinfo struct`
  - free the `addrinfo` structure using `freeaddrinfo( )`

- ipconfig
- ping
- traceroute e.g. traceroute www.google.com
- host
- netstat -nt
- dig
-

## 1 Knock Knock Example

- DNS

## 2 OSI Model

## 3 Networks and Sockets

## 4 Client

- IP Addresses
- DNS
- Create a socket

## 5 Server

- Knock Knock Server
- Knock Knock Client

# OSI Model

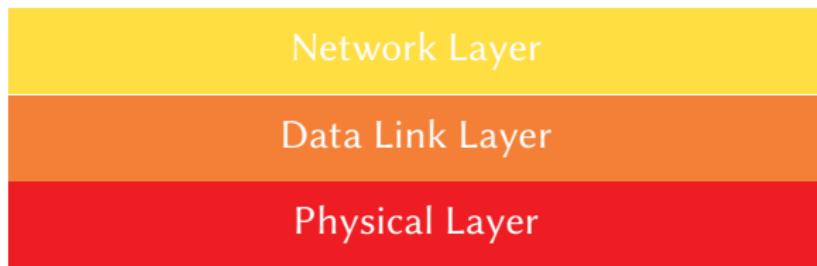
Physical Layer

# OSI Model

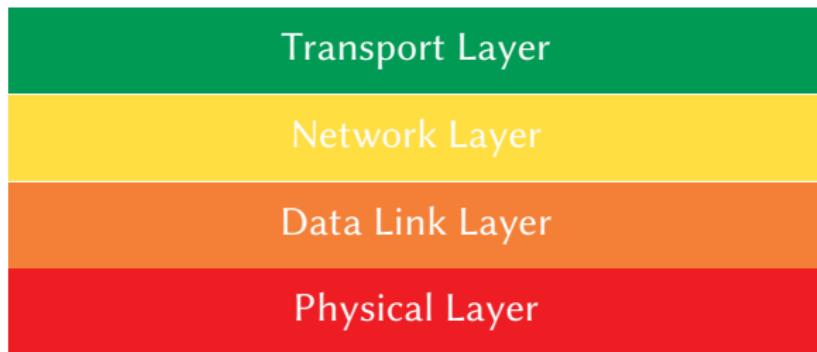
Data Link Layer

Physical Layer

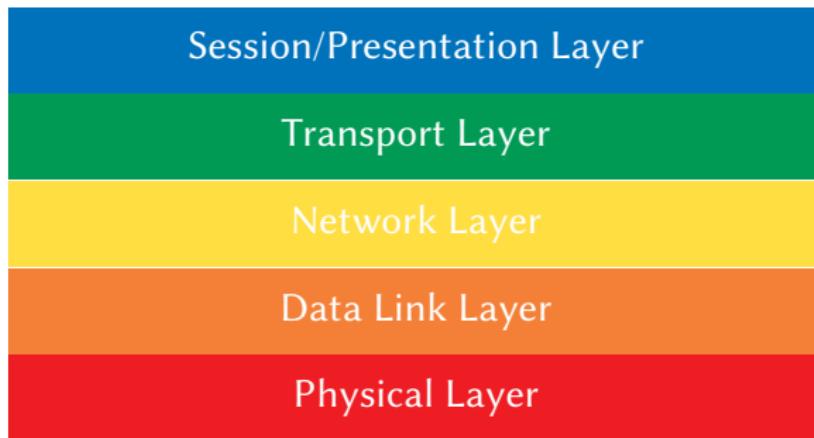
# OSI Model



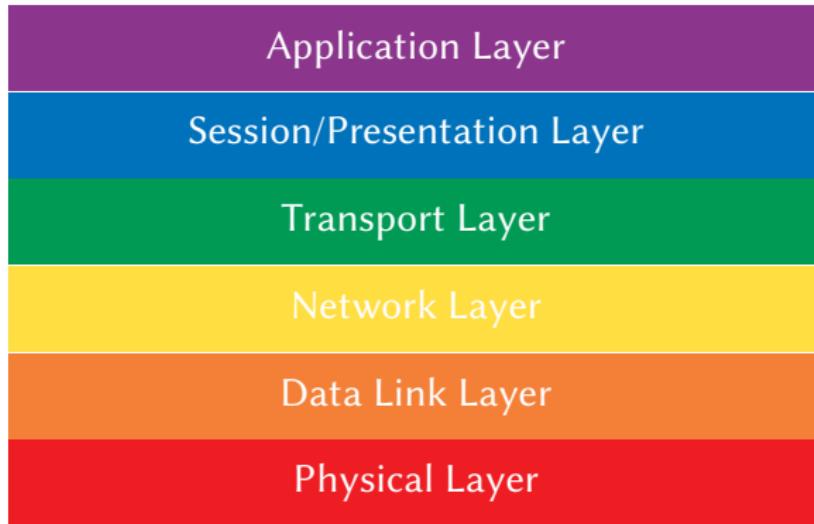
# OSI Model



# OSI Model

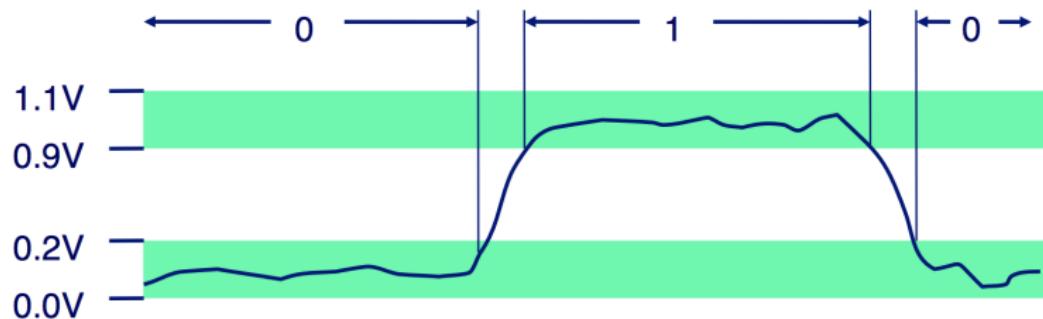


# OSI Model



# Starting at the bottom: Everything is Bits<sup>1</sup>

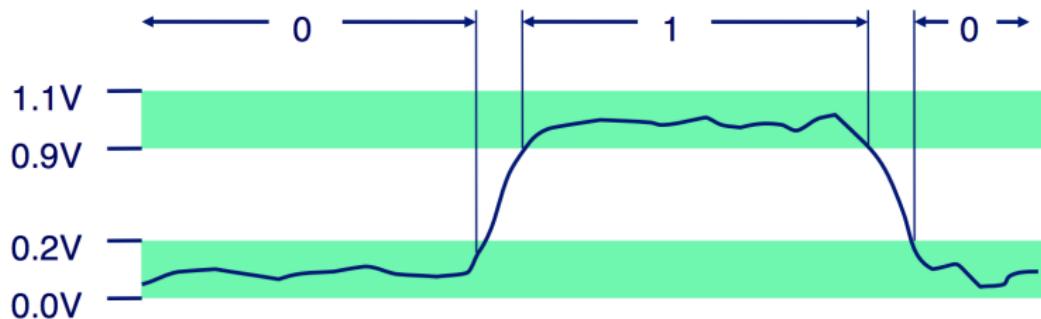
- Each bit is 1 or 0



<sup>1</sup>From Bryant & O'Halloran

# Starting at the bottom: Everything is Bits<sup>1</sup>

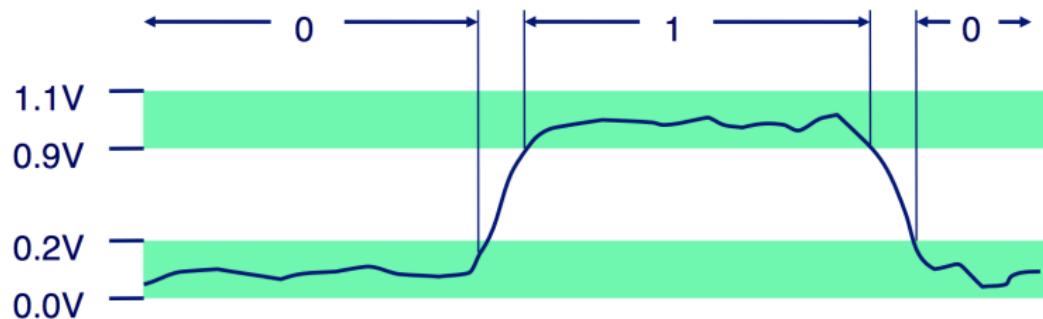
- Each bit is 1 or 0
- By encoding/interpreting sets of bits in various ways



<sup>1</sup>From Bryant & O'Halloran

# Starting at the bottom: Everything is Bits<sup>1</sup>

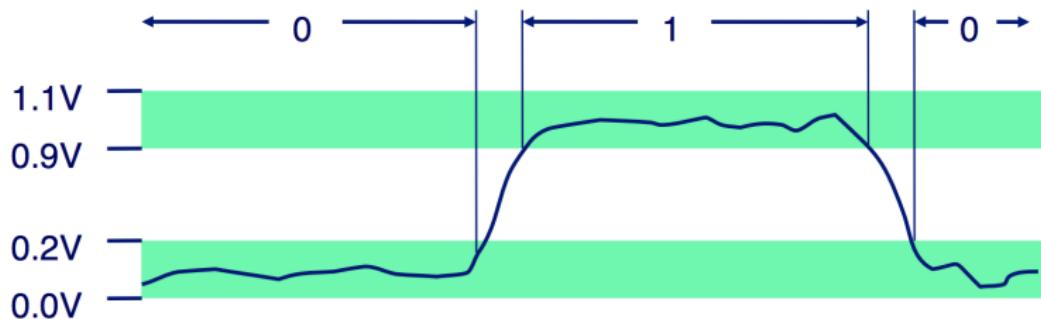
- Each bit is 1 or 0
- By encoding/interpreting sets of bits in various ways
- Why bits? Electronic implementation



<sup>1</sup>From Bryant & O'Halloran

# Starting at the bottom: Everything is Bits<sup>1</sup>

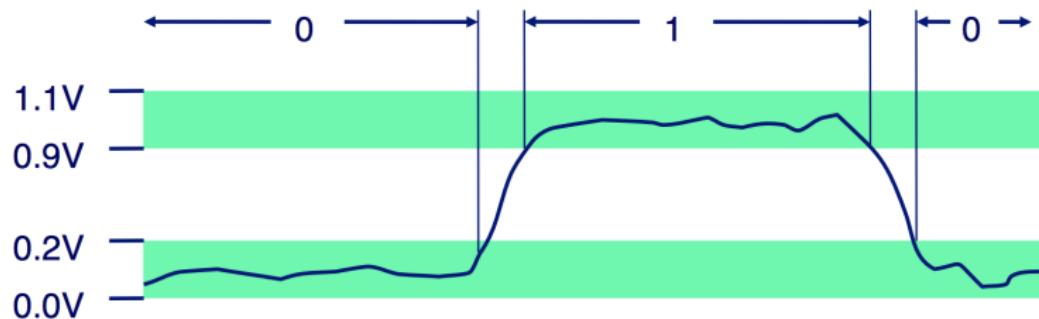
- Each bit is 1 or 0
- By encoding/interpreting sets of bits in various ways
- Why bits? Electronic implementation
  - Easy to store with bistable elements



<sup>1</sup>From Bryant & O'Halloran

# Starting at the bottom: Everything is Bits<sup>1</sup>

- Each bit is 1 or 0
- By encoding/interpreting sets of bits in various ways
- Why bits? Electronic implementation
  - Easy to store with bistable elements
  - Reliably transmitted on noisy and inaccurate wires



---

<sup>1</sup>From Bryant & O'Halloran

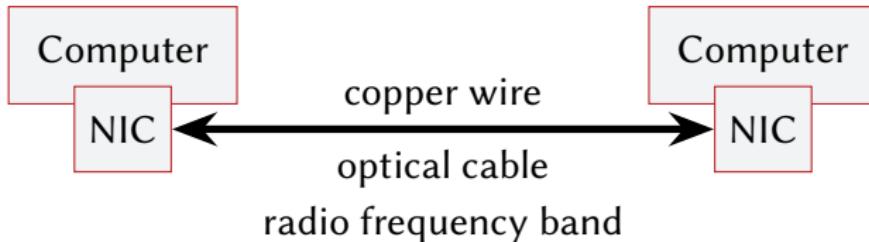
# OSI Model

## Physical Layer

# Physical layer

Individual bits are modulated onto a wire or transmitted over radio

- physical layer specifies how bits are encoded at a signal level
- e.g., a simple spec would encode “1” as +1V, “0” as -1V



NIC: Network Interface Controller

# Physical Layer: In your house

What does this look like in your house?

- Wires (on poles or underground) in your neighborhood

# Physical Layer: In your house

What does this look like in your house?

- Wires (on poles or underground) in your neighborhood
- Connect to your house

## Physical Layer: In your house

What does this look like in your house?

- Wires (on poles or underground) in your neighborhood
  - Connect to your house
  - Wire goes through the outside wall and into a jack in your living room

# Physical Layer: In your house

What does this look like in your house?

- Wires (on poles or underground) in your neighborhood
- Connect to your house
- Wire goes through the outside wall and into a jack in your living room
- Cable runs from the jack in the wall to a cable modem

# Physical Layer: In your house

What does this look like in your house?

- Wires (on poles or underground) in your neighborhood
- Connect to your house
- Wire goes through the outside wall and into a jack in your living room
- Cable runs from the jack in the wall to a cable modem
- Cable goes from the cable modem to a router

# Physical Layer: In your house

What does this look like in your house?

- Wires (on poles or underground) in your neighborhood
- Connect to your house
- Wire goes through the outside wall and into a jack in your living room
- Cable runs from the jack in the wall to a cable modem
- Cable goes from the cable modem to a router
- Ethernet cable goes from the router to my machine– the NIC card

# Physical Layer: In your house

What does this look like in your house?

- Wires (on poles or underground) in your neighborhood
- Connect to your house
- Wire goes through the outside wall and into a jack in your living room
- Cable runs from the jack in the wall to a cable modem
- Cable goes from the cable modem to a router
- Ethernet cable goes from the router to my machine– the NIC card
- The card is wired into the motherboard

# OSI Model

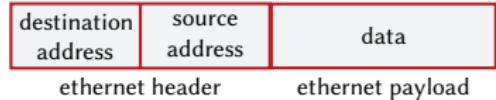
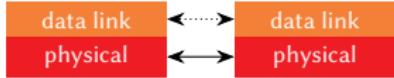
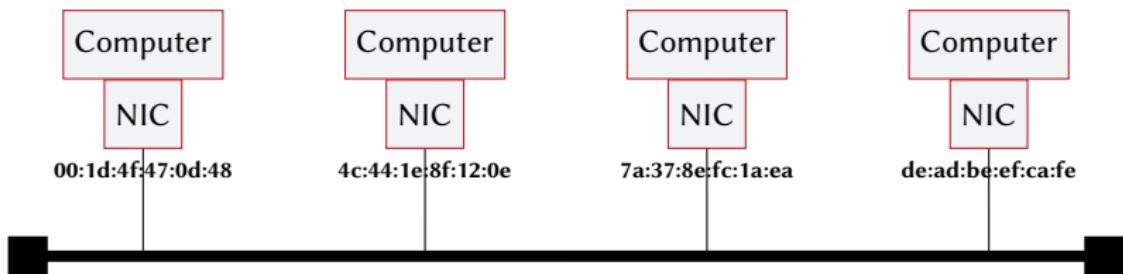
Data Link Layer

Physical Layer

# Data Link Layer

Multiple computers on a LAN share the network medium

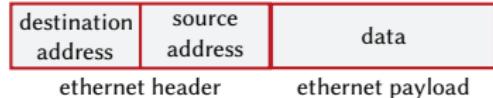
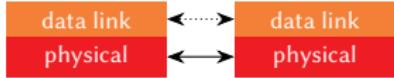
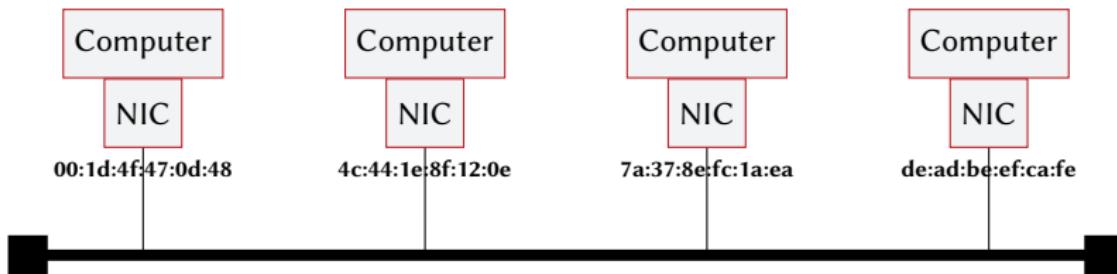
- media access control (MAC) specifies how computers cooperate



# Data Link Layer

Multiple computers on a LAN share the network medium

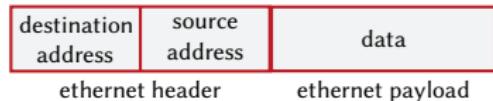
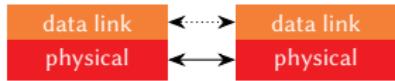
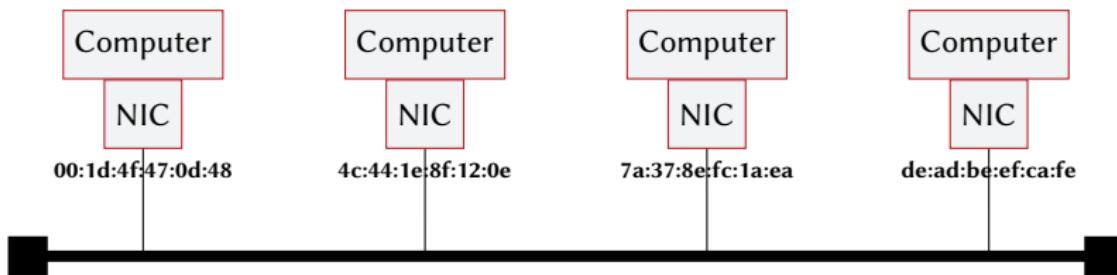
- media access control (MAC) specifies how computers cooperate
  - Every NIC card has its own MAC address



# Data Link Layer

Multiple computers on a LAN share the network medium

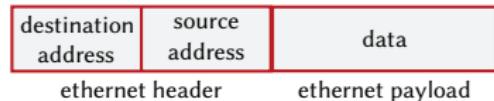
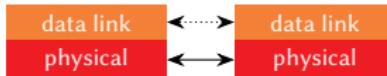
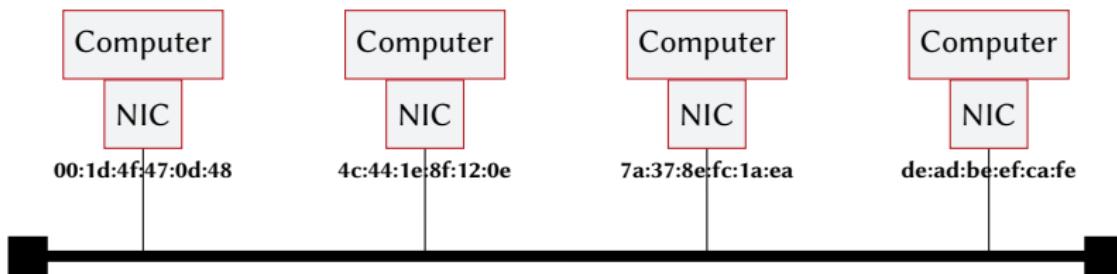
- media access control (MAC) specifies how computers cooperate
  - Every NIC card has its own MAC address
  - If you have multiple NICs in a machine (ethernet, Wi-Fi), you have multiple MACs



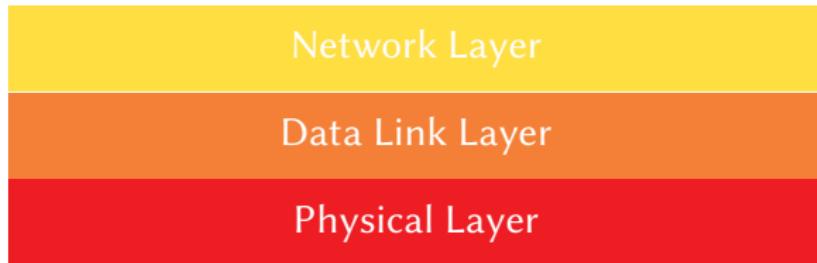
# Data Link Layer

Multiple computers on a LAN share the network medium

- media access control (MAC) specifies how computers cooperate
  - Every NIC card has its own MAC address
  - If you have multiple NICs in a machine (ethernet, Wi-Fi), you have multiple MACs
- link layer also specifies how bits are packetized and NICs are addressed



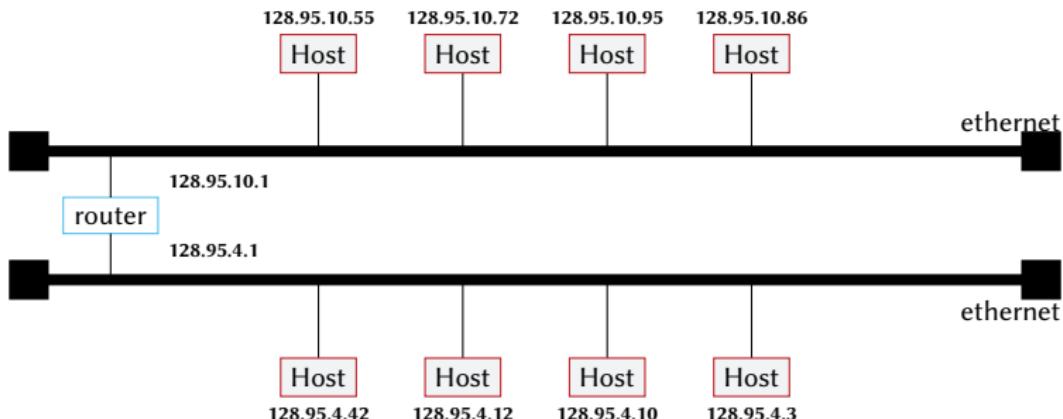
# OSI Model



# Network Layer

The Internet Protocol (IP) routes packets across multiple networks

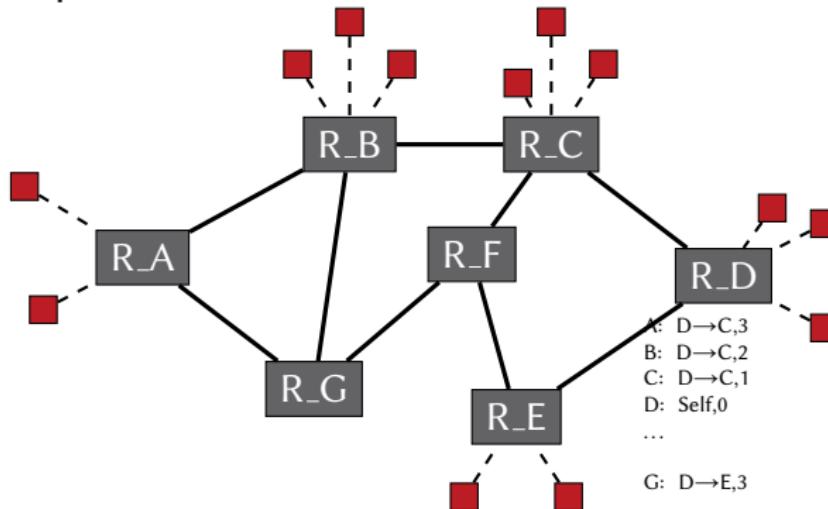
- every computer has a unique Internet address (IP address)
- individual networks are connected by routers that span networks



# The Network Layer (IP)

Protocols to:

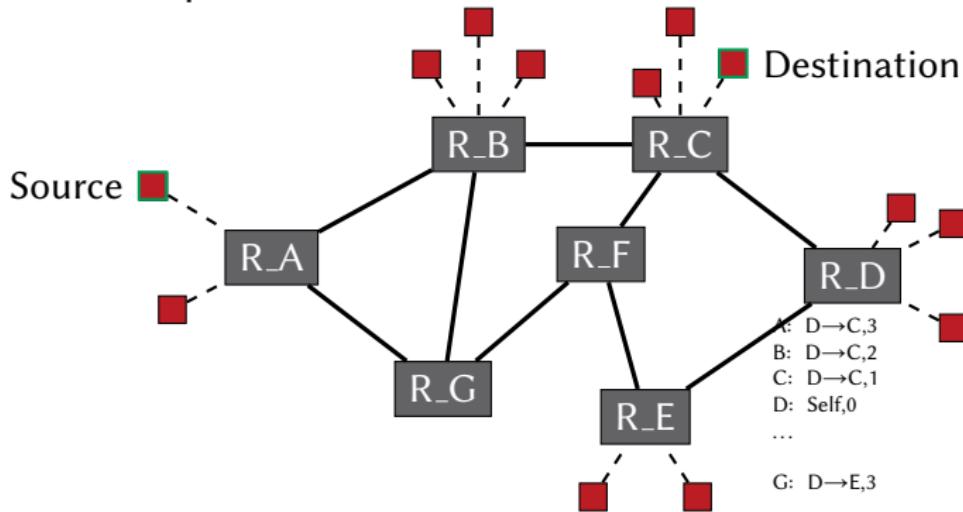
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



# The Network Layer (IP)

Protocols to:

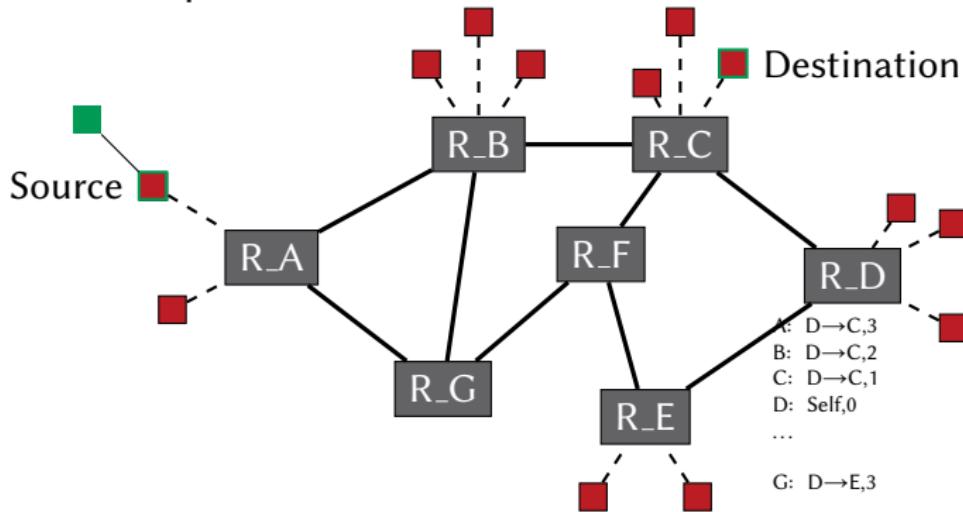
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



# The Network Layer (IP)

Protocols to:

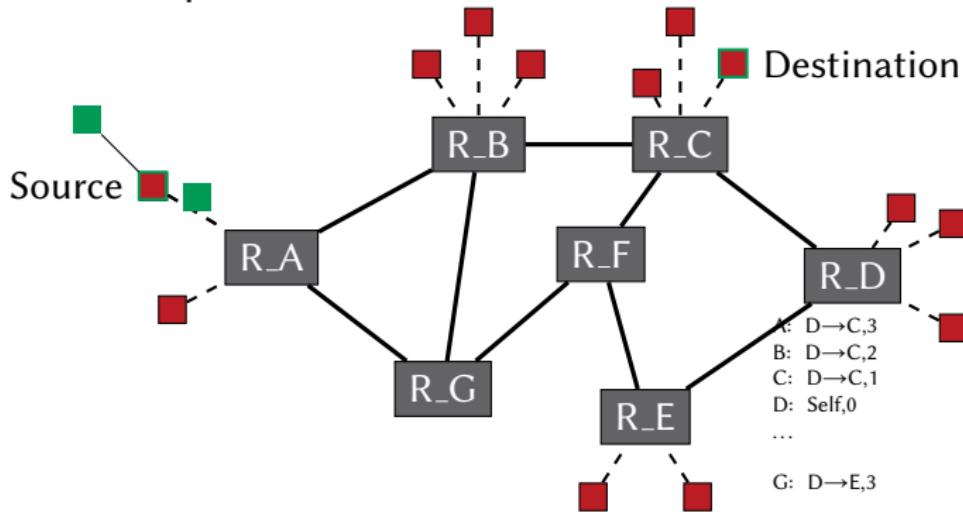
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



# The Network Layer (IP)

Protocols to:

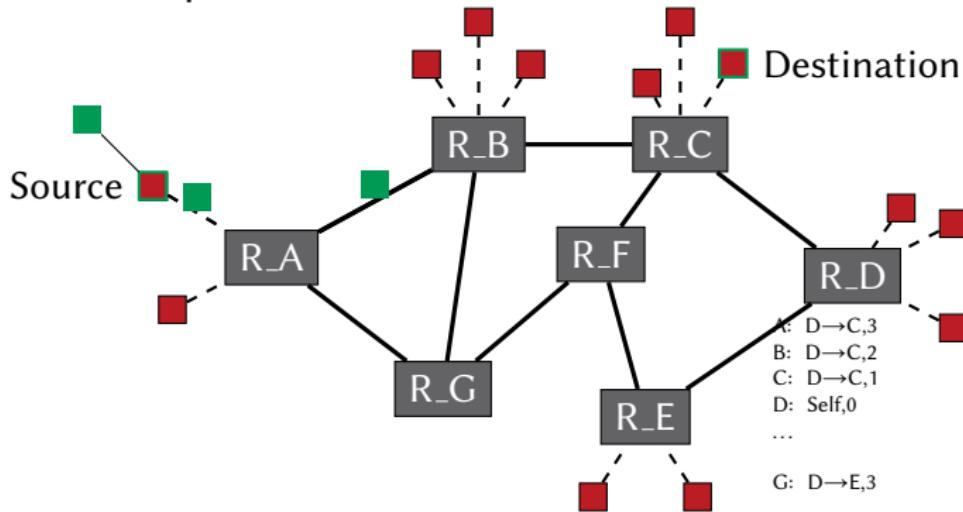
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



# The Network Layer (IP)

Protocols to:

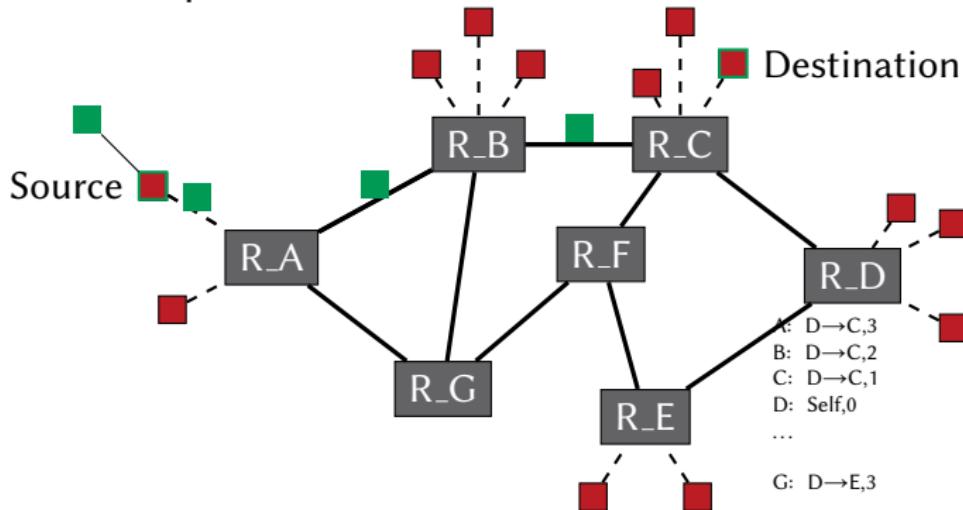
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



# The Network Layer (IP)

Protocols to:

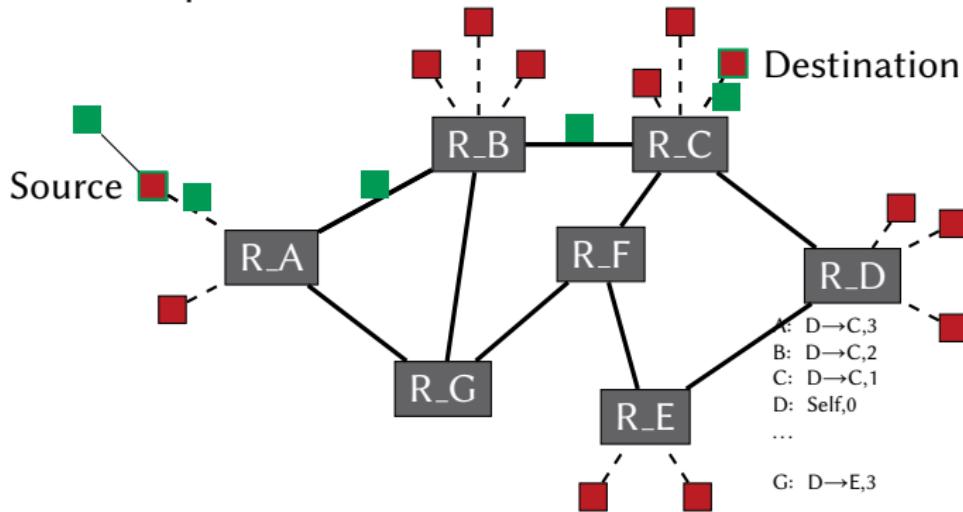
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



# The Network Layer (IP)

Protocols to:

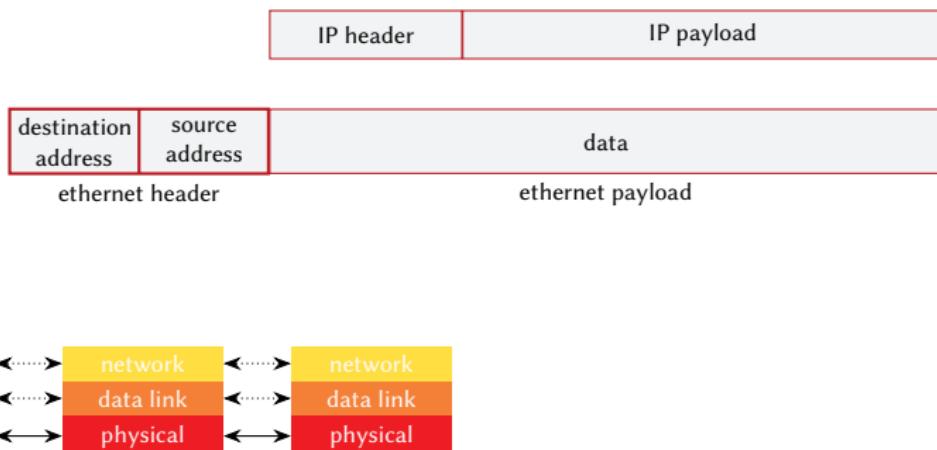
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



# The Network Layer

## Packet encapsulation

- an IP packet is encapsulated as the payload of an Ethernet frame
- as IP packets traverse networks, routers pull out the IP packet from an ethernet frame and plunk it into a new one on the next network



# A fun thing to do

`traceroute` let's you see the path a request takes to get to a destination:

```
adrienne@adrienne-VirtualBox:~/Spr19/cd5007/staff/assignments/projects$ traceroute drive.google.com
traceroute to drive.google.com (172.217.3.206), 30 hops max, 60 byte packets
 1  10.0.2.2 (10.0.2.2)  4.772 ms  4.607 ms  4.447 ms
 2  * *
 3  96.120.102.57 (96.120.102.57)  10.538 ms  14.508 ms  14.530 ms
 4  po-108-rur201.seattle.wa.seattle.comcast.net (96.108.11.93)  13.952 ms  15.120 ms  15.165 ms
 5  po-2-rur202.seattle.wa.seattle.comcast.net (69.139.161.46)  14.782 ms  13.467 ms  14.723 ms
 6  be-220-ar01.seattle.wa.seattle.comcast.net (69.139.160.249)  15.495 ms  10.767 ms  14.256 ms
 7  be-33650-cr01.seattle.wa.ibone.comcast.net (68.86.93.165)  18.337 ms  17.218 ms  18.255 ms
 8  be-10846-pe01.seattle.wa.ibone.comcast.net (68.86.86.90)  16.921 ms  16.754 ms  18.092 ms
 9  as46489-2-c.600wseventh.ca.ibone.comcast.net (50.242.148.22)  16.412 ms  50.248.118.134 (50.248.118.134)  18.062
ms  14.891 ms
10  * * 108.170.245.113 (108.170.245.113)  12.631 ms
11  108.170.233.155 (108.170.233.155)  12.586 ms seal5s12-in-f206.1e100.net (172.217.3.206)  15.587 ms  209.85.242.3
8 (209.85.242.38)  14.859 ms
adrienne@adrienne-VirtualBox:~/Spr19/cd5007/staff/assignments/projects$
```

# A fun thing to do

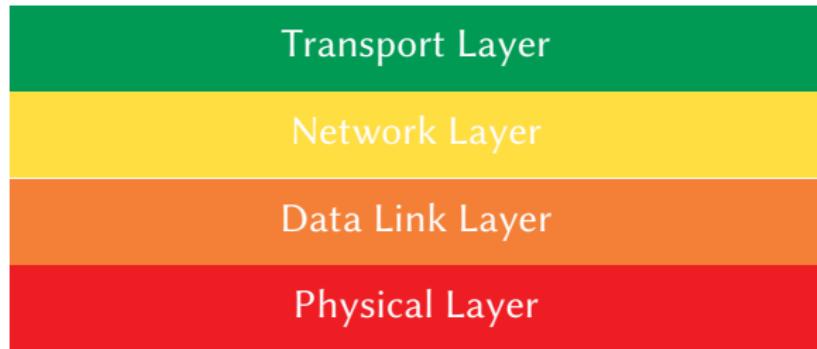
traceroute<sup>2</sup> let's you see the path a request takes to get to a destination:

```
2. adrienne@adrienne-VirtualBox:~/Spr19/cd5007/staff/assignments/projects$ traceroute northeastern.edu
traceroute to northeastern.edu (155.33.17.68), 30 hops max, 60 byte packets
 1  10.0.2.2 (10.0.2.2)  3.423 ms  3.321 ms  3.244 ms
 2  * * *
 3  96.120.102.57 (96.120.102.57)  11.524 ms  12.674 ms  12.040 ms
 4  po-108-rur201.seattle.wa.seattle.comcast.net (96.108.11.93)  13.632 ms  14.478 ms  14.054 ms
 5  po-2-rur202.seattle.wa.seattle.comcast.net (69.139.161.46)  12.879 ms  14.192 ms  14.221 ms
 6  be-220-ar01.seattle.wa.seattle.comcast.net (69.139.160.249)  15.058 ms  10.559 ms  15.247 ms
 7  be-33650-cr01.seattle.wa.ibone.comcast.net (68.86.93.165)  17.545 ms  17.127 ms  16.023 ms
 8  be-10820-cr01.champa.co.ibone.comcast.net (68.86.84.206)  51.001 ms  50.517 ms  48.499 ms
 9  be-12021-cr02.1601milehigh.co.ibone.comcast.net (68.86.84.226)  50.972 ms  51.032 ms  50.499 ms
10  be-10521-cr02.350ecermak.il.ibone.comcast.net (68.86.85.169)  71.778 ms  72.647 ms  71.297 ms
11  be-7922-ar01.woburn.ma.boston.comcast.net (68.86.91.6)  96.784 ms  96.386 ms  95.272 ms
12  be-1-sur01.roxbury.ma.boston.comcast.net (68.85.106.10)  98.533 ms  96.384 ms  95.414 ms
13  96.108.157.74 (96.108.157.74)  99.890 ms  99.881 ms  100.027 ms
14  50.203.71.122-static.hfc.comcastbusiness.net (50.203.71.122)  97.418 ms  95.089 ms  96.737 ms
15  10.2.29.229 (10.2.29.229)  95.762 ms  98.007 ms  103.601 ms
16  10.2.26.62 (10.2.26.62)  102.463 ms  100.120 ms  96.783 ms
17  * * *
18  * * *
19  * * *
20  * * *
21  * * *
22  * * *
23  * * *
24  * * *
25  * * *
26  * * *
27  * * *
28  * * *
29  * * *
30  * * *

adrienne@adrienne-VirtualBox:~/Spr19/cd5007/staff/assignments/projects$
```

<sup>2</sup><https://community.spiceworks.com/networking/articles/2531-traceroute-request-timed-out-why-traceroute-is-broken>

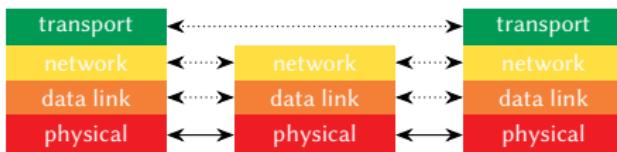
# OSI Model



# Transport Layer

## TCP

- the “transmission control protocol”
- provides apps with reliable, ordered, congestion-controlled byte streams
- fabricates them by sending multiple IP packets, using sequence numbers to detect missing packets, and retransmitting them
- a single host (IP address) can have up to 65,535 “ports”
- kind of like an apartment number at a postal address



# Transport Layer

## TCP

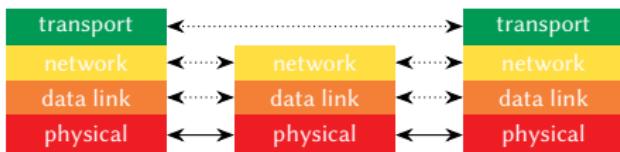
- useful analogy: how would you send a book by mail via postcards?



# Transport Layer

## TCP

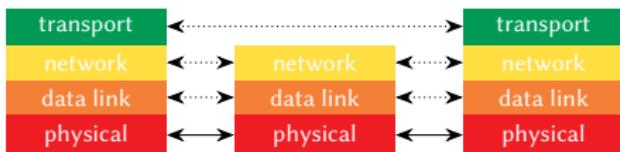
- useful analogy: how would you send a book by mail via postcards?
- split the book into multiple postcards, send each one by one, including sequence numbers that indicate the assembly order



# Transport Layer

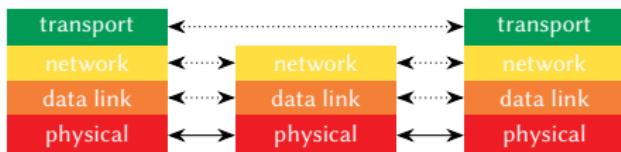
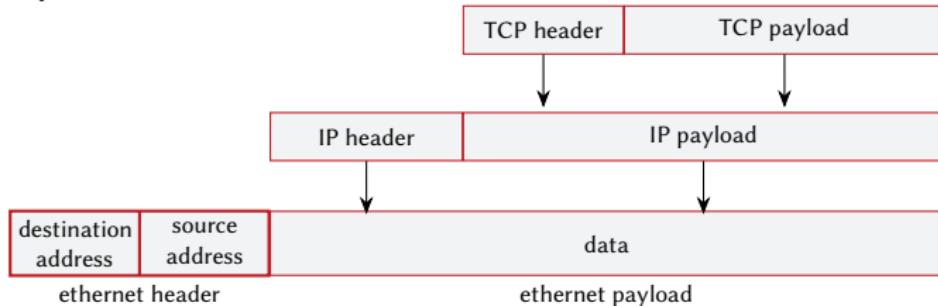
## TCP

- useful analogy: how would you send a book by mail via postcards?
- split the book into multiple postcards, send each one by one, including sequence numbers that indicate the assembly order
- receiver sends back postcards to acknowledge receipt and indicate which got lost in the mail



# The Transport Layer (TCP)

Packet encapsulation!



# Transport Layer (TCP)

Applications use OS services to establish TCP streams

- the “Berkeley sockets” API – a set of OS system calls
- clients `connect( )` to a server IP address + application port number
- servers `listen( )` for and `accept( )` client connections
- clients, servers `read( )` and `write( )` data to each other
- Messages sent as chunks
- TCP rearranges data packets in the order specified.
- There is absolute guarantee that the data transferred remains intact and arrives in the same order in which it was sent.
- TCP is heavy-weight. TCP requires three packets to set up a socket connection, before any user data can be sent. TCP handles reliability and congestion control.
- SYN, SYN-ACK, ACK

# Sniffing TCP traffic

knockknock.pcapng

Apply a display filter... >?

No. Time Source Destination Protocol Length Info

1 0.00000 192.168.86.190 192.168.86.191 TCP 78 64997 -> 2869 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=442014108 Tscr=0 SACK\_PERM=1

2 0.000004 192.168.86.190 192.168.86.191 TCP 78 [TCP Out-of-Order] 64997 -> 2869 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981182342 Tscr=442014108 SACK\_PERM=1

3 0.000148 192.168.86.191 192.168.86.190 TCP 78 2869 -> 64997 [SYN, ACK] Seq=1 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981182342 Tscr=442014108 SACK\_PERM=1

4 0.000148 192.168.86.191 192.168.86.190 TCP 78 [TCP Out-of-Order] 2869 -> 64997 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981182342 Tscr=442014108 SACK\_PERM=1

5 0.000457 192.168.86.190 192.168.86.191 TCP 78 192.168.86.191 -> 64997 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=442014386 Tscr=0 SACK\_PERM=1

6 0.000595 192.168.86.191 192.168.86.190 TCP 78 [TCP Out-of-Order] 2869 -> 64997 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981182342 Tscr=442014108 SACK\_PERM=1

7 0.000513 192.168.86.190 192.168.86.191 TCP 66 64997 -> 2869 [ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=442014398 Tscr=981182342

8 0.0005217 192.168.86.190 192.168.86.191 TCP 66 [TCP Dup ACK 7@1] 64997 -> 2869 [ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=442014398 Tscr=981182342

9 0.0005218 192.168.86.190 192.168.86.191 TCP 66 [TCP Dup ACK 7@2] 64997 -> 2869 [ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=442014398 Tscr=981182342

10 0.0005219 192.168.86.190 192.168.86.191 TCP 77 64997 -> 2869 [PSH, ACK] Seq=1 Ack=1 Win=131744 Len=11 Tsvl=442014398 Tscr=981182342 [TCP segment of a reassembly]

11 0.0005206 192.168.86.191 192.168.86.190 TCP 66 [TCP Window Update] 2869 -> 64997 [ACK] Seq=0 Ack=1 Win=131744 Len=0 Tsvl=981182347 Tscr=442014390

12 0.0005297 192.168.86.191 192.168.86.190 TCP 66 2869 -> 64997 [ACK] Seq=1 Ack=12 Win=131744 Len=0 Tsvl=981182347 Tscr=442014398

13 0.0005351 192.168.86.191 192.168.86.190 TCP 78 2869 -> 64997 [PSH, ACK] Seq=1 Ack=12 Win=131744 Len=12 Tsvl=981182347 Tscr=442014390 [TCP segment of a reassembly]

14 0.0008282 192.168.86.190 192.168.86.191 TCP 66 64997 -> 2869 [ACK] Seq=12 Ack=13 Win=131744 Len=0 Tsvl=442014394 Tscr=981182347

15 0.0008284 192.168.86.190 192.168.86.191 TCP 72 64997 -> 2869 [PSH, ACK] Seq=12 Ack=13 Win=131744 Len=6 Tsvl=442014394 Tscr=981182347 [TCP segment of a reassembly]

16 0.0008863 192.168.86.191 192.168.86.190 TCP 66 2869 -> 64997 [ACK] Seq=13 Ack=18 Win=131744 Len=0 Tsvl=981182352 Tscr=442014394

17 0.018943 192.168.86.191 192.168.86.190 TCP 77 2869 -> 64997 [PSH, ACK] Seq=13 Ack=18 Win=131744 Len=11 Tsvl=981182352 Tscr=442014394 [TCP segment of a reassembly]

18 0.015109 192.168.86.190 192.168.86.191 TCP 66 64997 -> 2869 [ACK] Seq=18 Ack=24 Win=131744 Len=0 Tsvl=442014399 Tscr=981182352

19 0.015114 192.168.86.190 192.168.86.191 TCP 183 64997 -> 2869 [PSH, ACK] Seq=18 Ack=24 Win=131744 Len=37 Tsvl=442014399 Tscr=981182352 [TCP segment of a reassembly]

20 0.015187 192.168.86.191 192.168.86.190 TCP 66 2869 -> 64997 [ACK] Seq=24 Ack=55 Win=131712 Len=0 Tsvl=981182356 Tscr=442014399

21 0.0152520 192.168.86.191 192.168.86.190 TCP 66 2869 -> 64997 [FIN, ACK] Seq=24 Ack=55 Win=131712 Len=0 Tsvl=981182356 Tscr=442014399

22 0.019428 192.168.86.190 192.168.86.191 TCP 66 64997 -> 2869 [ACK] Seq=55 Ack=25 Win=131744 Len=0 Tsvl=442014403 Tscr=981182356

23 0.019433 192.168.86.190 192.168.86.191 TCP 66 64997 -> 2869 [FIN, ACK] Seq=55 Ack=25 Win=131744 Len=0 Tsvl=442014403 Tscr=981182356

24 0.019512 192.168.86.191 192.168.86.190 TCP 66 [TCP Dup ACK 2@1] 2869 -> 64997 [ACK] Seq=25 Ack=55 Win=131712 Len=0 Tsvl=981182360 Tscr=442014403

25 0.019575 192.168.86.191 192.168.86.190 TCP 66 2869 -> 64997 [ACK] Seq=25 Ack=56 Win=131712 Len=0 Tsvl=981182360 Tscr=442014403

26 2.683992 192.168.86.190 192.168.86.191 TCP 78 64998 -> 2869 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=442016979 Tscr=0 SACK\_PERM=1

27 2.684163 192.168.86.191 192.168.86.190 TCP 78 2869 -> 64998 [SYN, ACK] Seq=1 Ack=1 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981184941 Tscr=442016979 SACK\_PERM=1

28 2.689341 192.168.86.190 192.168.86.191 TCP 66 64998 -> 2869 [ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=442016987 Tscr=981184941

29 2.689344 192.168.86.190 192.168.86.191 TCP 77 64998 -> 2869 [PSH, ACK] Seq=1 Ack=1 Win=131744 Len=11 Tsvl=442016987 Tscr=981184941 [TCP segment of a reassembly]

30 2.689494 192.168.86.191 192.168.86.190 TCP 66 [TCP Window Update] 2869 -> 64998 [ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=981184946 Tscr=442016987

31 2.689494 192.168.86.191 192.168.86.190 TCP 66 2869 -> 64998 [ACK] Seq=1 Ack=12 Win=131744 Len=0 Tsvl=981184946 Tscr=442016987

32 2.689465 192.168.86.191 192.168.86.190 TCP 78 2869 -> 64998 [PSH, ACK] Seq=1 Ack=12 Win=131744 Len=2 Tsvl=981184946 Tscr=442016987 [TCP segment of a reassembly]

Frame 1: 78 bytes on wire (624 bits), 78 bytes captured (624 bits) on interface 0

Ethernet II, Src: Apple\_68:5f:52 (78:4f:43:68:5f:52), Dst: Apple\_9e:89:a2 (60:03:08:9e:89:a2)

Internet Protocol Version 4, Src: 192.168.86.190, Dst: 192.168.86.191

Transmission Control Protocol, Src Port: 64997, Dst Port: 2869, Seq: 0, Len: 0

0000 60 03 8e 9e 89 a2 78 4f 43 68 5f 52 08 00 45 00 .....x0 Ch\_R..E.

0001 00 40 68 33 00 80 40 00 03 b7 c0 5b 56 bc ab ..0.30.0.....V.

0072 56 b1 fd e5 00 35 02 cd 00 a7 00 00 00 00 00 V....S.....

0030 ff ff 00 00 00 00 04 05 b4 81 03 03 01 01 ..C.....

0040 00 0a 1a 58 99 64 00 00 00 00 00 00 04 02 00 00 ..X.....

Packets: 104 - Displayed: 104 (100.0%) - Load time: 0:0:2

Profile: Default

# Sniffing TCP traffic

knockknock.pcapng

Apply a display filter... <31>

No.	Time	Source	Destination	Protocol	Length	Info
1	00:00:00.192, 192.168.86.190	192.168.86.191	TCP	78	64997 - 2869	[SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=442014188 Tscr=0 SACK_PERM=1
2	0.000004 192.168.86.190	192.168.86.191	TCP	78	64997 - 2869	[TCP Out-of-Order] 64997 - 2869 [SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=442014286 Tscr=0 SACK_PERM=1
3	0.000148 192.168.86.191	192.168.86.191	TCP	78	2869 - 64997	[SYN, ACK] Seq=1 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981182342 Tscr=442014188 SACK_PERM=1
4	0.000148 192.168.86.191	192.168.86.191	TCP	78	[TCP Out-of-Order] 2869 - 64997	[SYN, ACK] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981182342 Tscr=442014188 SACK_PERM=1
5	0.000457 192.168.86.190	192.168.86.191	TCP	78	[TCP Out-of-Order] 64997 - 2869	[SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=442014386 Tscr=0 SACK_PERM=1
6	0.000595 192.168.86.191	192.168.86.191	TCP	78	[TCP Out-of-Order] 2869 - 64997	[SYN, ACK] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981182342 Tscr=442014188 SACK_PERM=1
7	0.005213 192.168.86.190	192.168.86.191	TCP	66	64997 - 2869	[ACK] Seq=1 Win=131744 Len=0 Tsvl=42014394 Tscr=981182342
8	0.005217 192.168.86.190	192.168.86.191	TCP	66	[TCP Dup ACK 7@1] 64997 - 2869 [ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=42014394 Tscr=981182342	
9	0.005218 192.168.86.190	192.168.86.191	TCP	66	[TCP Dup ACK 7@1] 64997 - 2869 [ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=42014394 Tscr=981182342	
10	0.005219 192.168.86.190	192.168.86.191	TCP	78	64997 - 2869	[PSH, ACK] Seq=1 Ack=1 Win=131744 Len=11 Tsvl=442014398 Tscr=981182342 [TCP segment of a reassembly]
11	0.005296 192.168.86.191	192.168.86.191	TCP	66	[TCP Window Update] 2869 - 64997	[ACK] Seq=0 Ack=1 Win=131744 Len=0 Tsvl=981182347 Tscr=442014398
12	0.005297 192.168.86.191	192.168.86.191	TCP	78	2869 - 64997	[ACK] Seq=1 Ack=12 Win=131744 Len=0 Tsvl=981182347 Tscr=442014398
13	0.005351 192.168.86.191	192.168.86.191	TCP	78	2869 - 64997	[PSH, ACK] Seq=1 Win=131744 Len=12 Tsvl=981182347 Tscr=442014398 [TCP segment of a reassembly]
14	0.008182 192.168.86.190	192.168.86.191	TCP	66	64997 - 2869	[ACK] Seq=12 Ack=13 Win=131744 Len=0 Tsvl=442014394 Tscr=981182347
15	0.008182 192.168.86.190	192.168.86.191	TCP	72	64997 - 2869	[PSH, ACK] Seq=12 Ack=13 Win=131744 Len=6 Tsvl=442014394 Tscr=981182347 [TCP segment of a reassembly]
16	0.008183 192.168.86.191	192.168.86.191	TCP	66	2869 - 64997	[ACK] Seq=13 Ack=18 Win=131744 Len=0 Tsvl=981182352 Tscr=442014394
17	0.018943 192.168.86.191	192.168.86.191	TCP	77	2869 - 64997	[PSH, ACK] Seq=13 Ack=18 Win=131744 Len=0 Tsvl=981182352 Tscr=442014394 [TCP segment of a reassembly]
18	0.018949 192.168.86.190	192.168.86.191	TCP	66	64997 - 2869	[ACK] Seq=18 Ack=24 Win=131744 Len=0 Tsvl=442014399 Tscr=981182352
19	0.015114 192.168.86.190	192.168.86.191	TCP	183	64997 - 2869	[PSH, ACK] Seq=18 Ack=24 Win=131744 Len=37 Tsvl=442014399 Tscr=981182352 [TCP segment of a reassembly]
20	0.015187 192.168.86.191	192.168.86.191	TCP	66	2869 - 64997	[ACK] Seq=24 Ack=55 Win=131713 Len=0 Tsvl=981182356 Tscr=442014399
21	0.015250 192.168.86.191	192.168.86.191	TCP	66	2869 - 64997	[FIN, ACK] Seq=24 Ack=55 Win=131712 Len=0 Tsvl=981182356 Tscr=442014399
22	0.019428 192.168.86.190	192.168.86.191	TCP	66	64997 - 2869	[ACK] Seq=55 Ack=25 Win=131744 Len=0 Tsvl=442014403 Tscr=981182356
23	0.019433 192.168.86.190	192.168.86.191	TCP	66	64997 - 2869	[FIN, ACK] Seq=55 Ack=25 Win=131744 Len=0 Tsvl=442014403 Tscr=981182356
24	0.019512 192.168.86.191	192.168.86.191	TCP	66	[TCP Dup ACK 2@1] 2869 - 64997	[ACK] Seq=25 Ack=55 Win=131712 Len=0 Tsvl=981182360 Tscr=442014403
25	0.019575 192.168.86.191	192.168.86.191	TCP	66	2869 - 64997	[ACK] Seq=25 Ack=56 Win=131712 Len=0 Tsvl=981182360 Tscr=442014403
26	2.683992 192.168.86.190	192.168.86.191	TCP	78	64998 - 2869	[SYN] Seq=0 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=442016979 Tscr=0 SACK_PERM=1
27	2.684163 192.168.86.191	192.168.86.191	TCP	78	2869 - 64998	[SYN, ACK] Seq=1 Ack=1 Win=65535 Len=0 MSS=1460 WS=32 Tsvl=981184941 Tscr=442016979 SACK_PERM=1
28	2.689341 192.168.86.190	192.168.86.191	TCP	66	64998 - 2869	[ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=442016987 Tscr=981184941
29	2.689344 192.168.86.190	192.168.86.191	TCP	77	64998 - 2869	[PSH, ACK] Seq=1 Ack=1 Win=131744 Len=11 Tsvl=442016987 Tscr=981184941 [TCP segment of a reassembly]
30	2.689494 192.168.86.191	192.168.86.191	TCP	66	[TCP Window Update] 2869 - 64998	[ACK] Seq=1 Ack=1 Win=131744 Len=0 Tsvl=981184946 Tscr=442016987
31	2.689494 192.168.86.191	192.168.86.191	TCP	66	2869 - 64998	[ACK] Seq=1 Ack=12 Win=131744 Len=0 Tsvl=981184946 Tscr=442016987
32	2.689465 192.168.86.191	192.168.86.191	TCP	78	2869 - 64998	[PSH, ACK] Seq=1 Ack=12 Win=131744 Len=22 Tsvl=981184946 Tscr=442016987 [TCP segment of a reassembly]

Frame 18: 77 bytes on wire (616 bits), 77 bytes captured (616 bits) on interface 0

Ethernet II, Src: Apple\_68:5f:52 (78:4f:43:68:5f:52), Dst: Apple\_9e:89:a2 (60:03:89:89:a2)

Internet Protocol Version 4, Src: 192.168.86.190, Dst: 192.168.86.191

Transmission Control Protocol, Src Port: 64997, Dst Port: 2869, Seq: 1, Ack: 1, Len: 11

0000 00 03 08 9e 89 a2 78 f4 43 68 5f 52 08 00 45 00 . ....x0 Ch\_R..E.  
0001 00 31 a2 07 40 89 a0 69 e3 c8 5b 06 c8 a8 ?..0..1..V...  
0072 56 b1 0c c5 35 a2 c0 7d 31 8e 18 V.....1..  
0030 18 15 f7 00 00 01 86 00 1a 58 9d 66 3a 7b ..F....X.:{  
0040 a7 86 4b 6e 6f 63 6b 20 6b 6e 6f 63 6b ..Knock Knock

Packets: 104 - Displayed: 104 (100.0%) - Load time: 0:0:2

Profile: Default

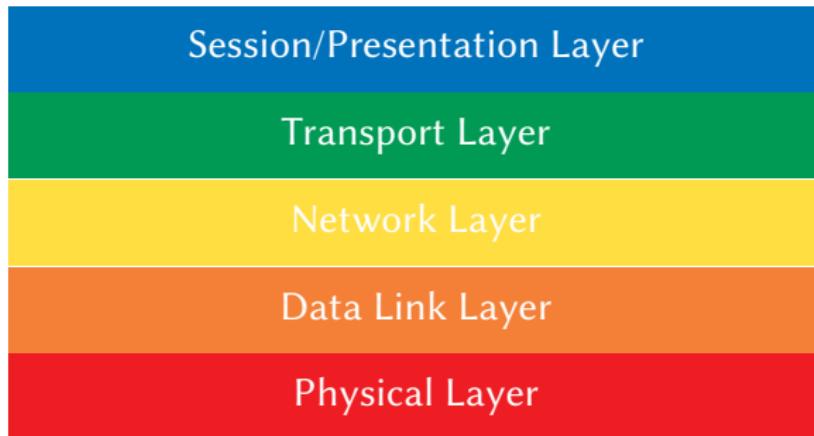
# Transport Layer (UDP)

## UDP

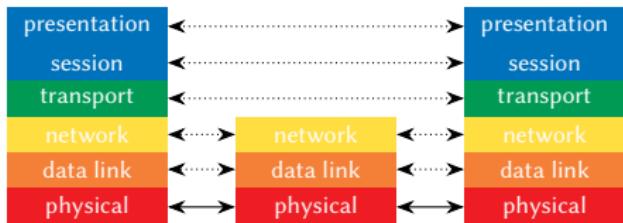
- the “user datagram protocol”
- UDP is faster because error recovery is not attempted. It is a “best effort” protocol.
- There is no guarantee that the messages or packets sent would reach at all.
- UDP datagrams are fragmented into multiple IP packets
  - UDP is a really thin, simple layer on top of IP
- UDP is lightweight. There is no ordering of messages, no tracking connections, etc. It is a small transport layer designed on top of IP.
- No ACK sent
- No handshake; connectionless

I have a UDP joke but you wouldn't get it.

# OSI Model



# Session, Presentation Layers



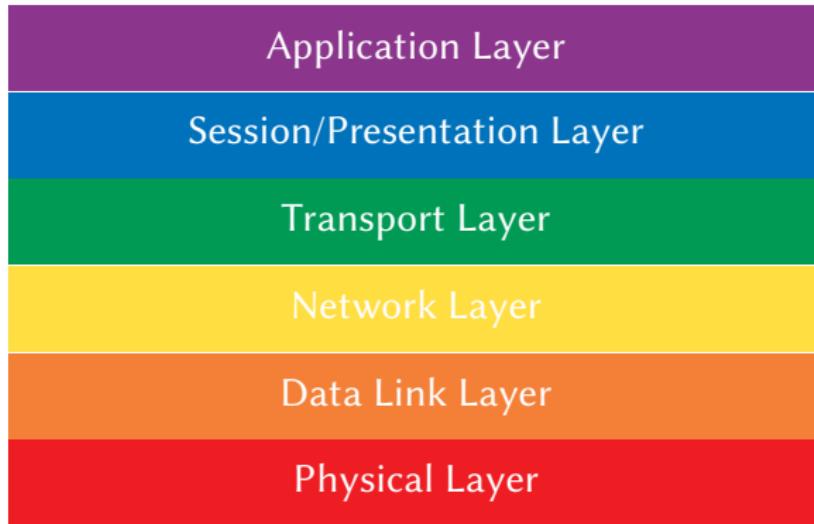
## Layer 5: session layer

- supposedly handles establishing, terminating application sessions
- RPC kind of fits in here

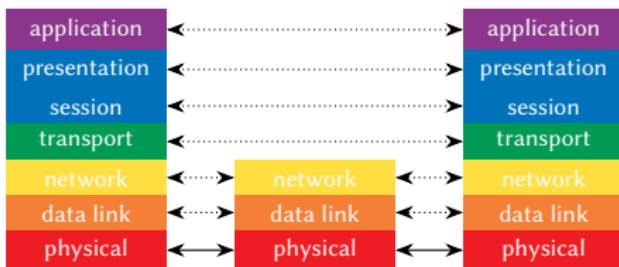
## Layer 6: presentation layer

- supposedly maps application specific data units into a more network-neutral representation
- encryption (SSL) kind of fits in here

# OSI Model



# Application Layer

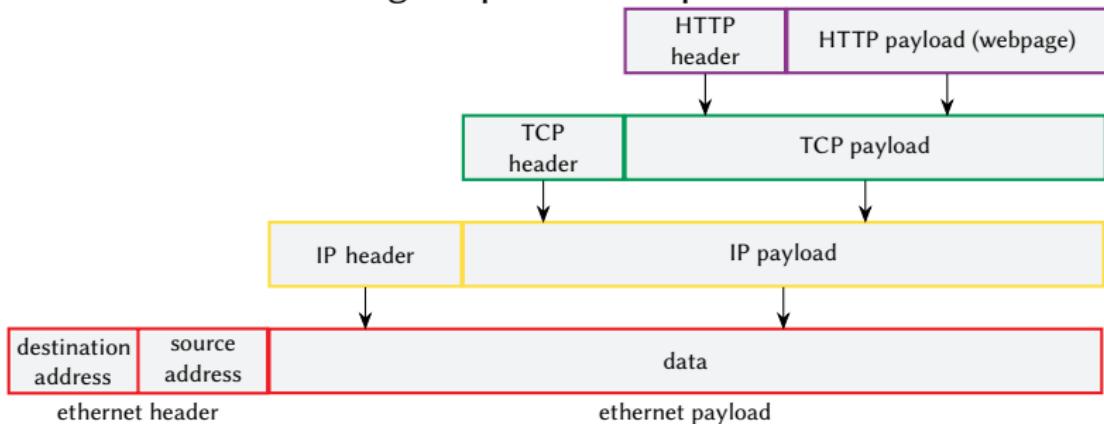


## Application protocols

- the format and meaning of messages between application entities
- e.g., HTTP is an application level protocol that dictates how web browsers and web servers communicate
  - HTTP is implemented on top of TCP streams

# Application Layer

Once again, packet encapsulation:



# The Application Layer



# The Application Layer

Popular application-level protocols:

- DNS: translates a DNS name (www.google.com) into one or more IP addresses (74.125.155.105, 74.125.155.106, ...)

# The Application Layer

Popular application-level protocols:

- DNS: translates a DNS name (www.google.com) into one or more IP addresses (74.125.155.105, 74.125.155.106, ...)
  - a hierarchy of DNS servers cooperate to do this

# The Application Layer

Popular application-level protocols:

- DNS: translates a DNS name (www.google.com) into one or more IP addresses (74.125.155.105, 74.125.155.106, ...)
  - a hierarchy of DNS servers cooperate to do this
- HTTP: web protocols

# The Application Layer

Popular application-level protocols:

- DNS: translates a DNS name (www.google.com) into one or more IP addresses (74.125.155.105, 74.125.155.106, ...)
  - a hierarchy of DNS servers cooperate to do this
- HTTP: web protocols
- SMTP, IMAP, POP: mail delivery and access protocols

# The Application Layer

Popular application-level protocols:

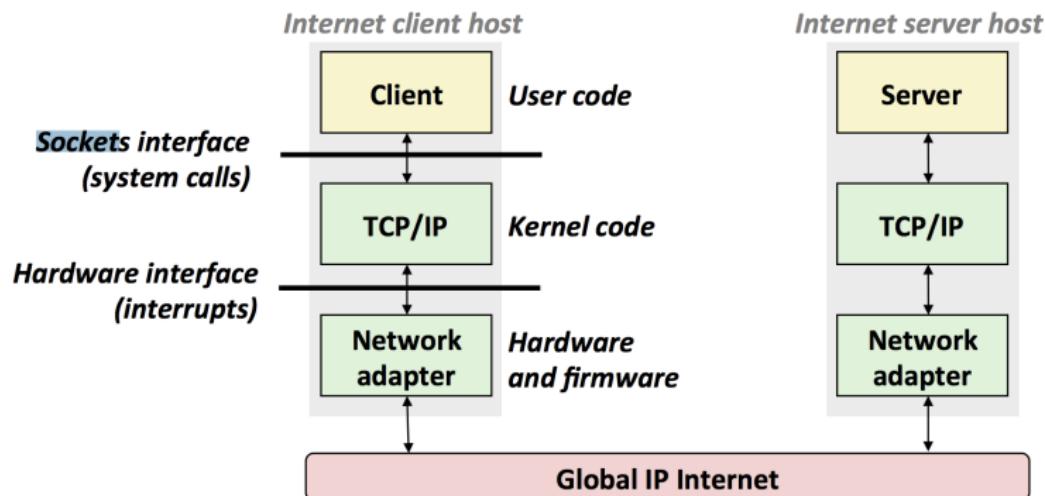
- DNS: translates a DNS name (www.google.com) into one or more IP addresses (74.125.155.105, 74.125.155.106, ...)
  - a hierarchy of DNS servers cooperate to do this
- HTTP: web protocols
- SMTP, IMAP, POP: mail delivery and access protocols
- ssh: remote login protocol

# The Application Layer

Popular application-level protocols:

- DNS: translates a DNS name (www.google.com) into one or more IP addresses (74.125.155.105, 74.125.155.106, ...)
  - a hierarchy of DNS servers cooperate to do this
- HTTP: web protocols
- SMTP, IMAP, POP: mail delivery and access protocols
- ssh: remote login protocol
- bittorrent: peer-to-peer, swarming file sharing protocol

# Hardware and Software Organization of an Internet Application



## Intro to Networking Summary

- How bits get from one computer to another and turned into “stuff”

## 1 Knock Knock Example

- DNS

## 2 OSI Model

## 3 Networks and Sockets

## 4 Client

- IP Addresses
- DNS
- Create a socket

## 5 Server

- Knock Knock Server
- Knock Knock Client

# Files and file descriptors

Remember open, read, write, and close?

- POSIX system calls for interacting with files

# Files and file descriptors

Remember open, read, write, and close?

- POSIX system calls for interacting with files
- `open()` returns a file descriptor

# Files and file descriptors

Remember open, read, write, and close?

- POSIX system calls for interacting with files
- `open()` returns a file descriptor
  - an integer that represents an open file

# Files and file descriptors

Remember open, read, write, and close?

- POSIX system calls for interacting with files
- `open()` returns a file descriptor
  - an integer that represents an open file
  - inside the OS, it's an index into a table that keeps track of any state associated with your interactions, such as the file position

# Files and file descriptors

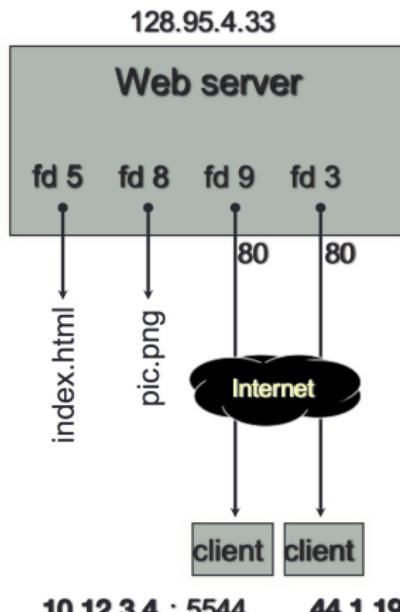
Remember open, read, write, and close?

- POSIX system calls for interacting with files
- `open()` returns a file descriptor
  - an integer that represents an open file
  - inside the OS, it's an index into a table that keeps track of any state associated with your interactions, such as the file position
  - you pass the file descriptor into read, write, and close

# Networks and Sockets

UNIX likes to make all I/O look like file I/O

# Graphically...



OS File Descriptor Table

file descriptor	type	connected to?
0	pipe	stdin
1	pipe	stdout
2	pipe	stderr
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP Socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

# Types of Sockets

## Stream sockets

- for connection-oriented, point-to-point, reliable bytestreams
  - uses TCP, SCTP, or other stream transports

# Types of Sockets

## Stream sockets

- for connection-oriented, point-to-point, reliable byte streams
  - uses TCP, SCTP, or other stream transports

## Datagram sockets

- for connection-less, one-to-many, unreliable packets
  - uses UDP or other packet transports

# Types of Sockets

## Stream sockets

- for connection-oriented, point-to-point, reliable bytestreams
  - uses TCP, SCTP, or other stream transports

## Datagram sockets

- for connection-less, one-to-many, unreliable packets
  - uses UDP or other packet transports

## Raw sockets

- for layer-3 communication (raw IP packet manipulation)

# Stream Sockets

Typically used for client / server communications

- but also for other architectures, like peer-to-peer

- ① Establish connection
- ② Communicate
- ③ Close connection

## *Client*

- an application that establishes a connection to a server

## *Server*

- an application that receives connections from clients

# Datagram Sockets

Used less frequently than stream sockets

- they provide no flow control, ordering, or reliability

Often used as a building block

- streaming media applications
- sometimes, DNS lookups

# The sockets API

Berkeley sockets originated in 4.2 BSD Unix circa 1983

- it is the standard API for network programming
  - available on most OSs

POSIX socket API

- a slight updating of the Berkeley sockets API
  - a few functions were deprecated or replaced
  - better support for multi-threading was added

## 1 Knock Knock Example

- DNS

## 2 OSI Model

## 3 Networks and Sockets

## 4 Client

- IP Addresses
- DNS
- Create a socket

## 5 Server

- Knock Knock Server
- Knock Knock Client

# Connecting as a Client

We'll start by looking at the API from the point of view of a client connecting to a server over TCP:

- There are five steps:
  - ① figure out the IP address and port to which to connect

# Connecting as a Client

We'll start by looking at the API from the point of view of a client connecting to a server over TCP:

- There are five steps:

- ① figure out the IP address and port to which to connect
- ② create a socket

# Connecting as a Client

We'll start by looking at the API from the point of view of a client connecting to a server over TCP:

- There are five steps:

- ① figure out the IP address and port to which to connect
- ② create a socket
- ③ connect the socket to the remote server

# Connecting as a Client

We'll start by looking at the API from the point of view of a client connecting to a server over TCP:

- There are five steps:

- ① figure out the IP address and port to which to connect
- ② create a socket
- ③ connect the socket to the remote server
- ④ `read( )` and `write( )` data using the socket

# Connecting as a Client

We'll start by looking at the API from the point of view of a client connecting to a server over TCP:

- There are five steps:

- ① figure out the IP address and port to which to connect
- ② create a socket
- ③ connect the socket to the remote server
- ④ `read( )` and `write( )` data using the socket
- ⑤ close the socket

Connecting from a client to a server.

Step 1. Figure out the IP address and port to which to connect.

# Network Addresses

For IPv4, an IP address is a 4-byte tuple

- e.g., 128.95.4.1 (80:5f:04:01 in hex)

For IPv6, an IP address is a 16-byte tuple

- e.g., 2d01:0db8:f188:0000:0000:0000:0000:1f33
  - 2d01:0db8:f188::1f33 in shorthand

# IPv4 address structures

```
1 // Port numbers and addresses are in *network order*.
2 // A mostly-protocol-independent address structure.
3 struct sockaddr {
4     short int sa_family; // Address family; AF_INET, AF_INET6
5     char sa_data[14]; // 14 bytes of protocol address
6 };
7
8 // An IPv4 specific address structure.
9 struct sockaddr_in {
10    short int sin_family; // Address family, AF_INET == IPv4
11    unsigned short int sin_port; // Port number
12    struct in_addr sin_addr; // Internet address
13    unsigned char sin_zero[8]; // Same size as struct sockaddr
14 };
15
16 struct in_addr {
17     uint32_t s_addr; // IPv4 address
18 };
```

# IPv4 address structures

```
1 // A structure big enough to hold either IPv4 or IPv6 structures.
2 struct sockaddr_storage {
3     sa_family_t ss_family; // address family
4     // a bunch of padding; safe to ignore it.
5     char __ss_pad1[_SS_PAD1SIZE];
6     int64_t __ss_align;
7     char __ss_pad2[_SS_PAD2SIZE];
8 };
9
10 // An IPv6 specific address structure.
11 struct sockaddr_in6 {
12     u_int16_t sin6_family; // address family, AF_INET6
13     u_int16_t sin6_port; // Port number
14     u_int32_t sin6_flowinfo; // IPv6 flow information
15     struct in6_addr sin6_addr; // IPv6 address
16     u_int32_t sin6_scope_id; // Scope ID
17 };
18
19 struct in6_addr {
20     unsigned char s6_addr[16]; // IPv6 address
21 };
```

# Generating these structures

Often you have a string representation of an address

- how do you generate one of the address structures?

```
1 #include <stdlib.h>
2 #include <arpa/inet.h>
3 int main(int argc, char **argv) {
4     struct sockaddr_in sa; // IPv4
5     struct sockaddr_in6 sa6; // IPv6
6     // IPv4 string to sockaddr_in.
7     inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));
8     // IPv6 string to sockaddr_in6.
9     inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));
10    return EXIT_SUCCESS;
11 }
```

# Resolving DNS names

The POSIX way is to use `getaddrinfo( )`

- a pretty complicated system call; the basic idea...
  - set up a “hints” structure with constraints you want respected
    - e.g., IPv6, IPv4, or either
  - tell `getaddrinfo( )` which host and port you want resolved
    - host: a string representation; DNS name or IP address
  - `getaddrinfo( )` gives you a list of results packet in an `addrinfo struct`
  - free the `addrinfo` structure using `freeaddrinfo( )`

# DNS lookup example<sup>3</sup>

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <sys/types.h>
4 #include <sys/socket.h>
5 #include <netdb.h>
6
7 struct addrinfo hints, *infoptr;
8
9 int main() {
10     hints.ai_family = AF_INET; // AF_INET means IPv4 only addresses
11
12     int result = getaddrinfo("www.northeastern.edu", NULL, &hints, &infoptr);
13     if (result) {
14         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
15         exit(1);
16     }
17
18     struct addrinfo *p;
19     char host[256];
20
21     for(p = infoptr; p != NULL; p = p->ai_next) {
22
23         getnameinfo(p->ai_addr, p->ai_addrlen, host, sizeof(host), NULL, 0, NI_NUMERICHOST);
24         puts(host);
25     }
26
27     freeaddrinfo(infoptr);
28     return 0;
29 }
```

---

<sup>3</sup>[https://github.com/angrave/SystemProgramming/wiki/Networking,  
-Part-2:-Using-getaddrinfo](https://github.com/angrave/SystemProgramming/wiki/Networking,-Part-2:-Using-getaddrinfo)

Connecting from a client to a server.

Step 2. Create a socket.

# Creating a socket

Use the `socket()` system call

- creating a socket doesn't yet bind it to a local address or port

```
1 #include <stdio.h>
2 #include <sys/socket.h>
3 #include <stdlib.h>
4 #include <string.h>
5
6
7 int main() {
8     int socket_fd = socket(PF_INET, SOCK_STREAM, 0);
9     if (socket_fd == -1) {
10         printf("Error");
11         return EXIT_FAILURE;
12     } else{
13         printf("Socket opened\n");
14     }
15 }
16 close(socket_fd);
17 return EXIT_SUCCESS;
18 }
19 }
```

Connecting from a client to a server.

Step 3. Connect the socket to the remote server.

## connect()

The `connect( )` system call establishes a connection to a remote host

- you pass the following arguments to `connect( )` :
  - the socket file descriptor you created in step 2
  - one of the address structures you created in step 1

## connect()

The `connect( )` system call establishes a connection to a remote host

- you pass the following arguments to `connect( )` :
  - the socket file descriptor you created in step 2
  - one of the address structures you created in step 1
- `connect()` may take some time to return
  - it is a blocking call by default
  - the network stack within the OS will communicate with the remote host to establish a TCP connection to it
  - this involves 2 round trips across the network

# Connecting<sup>4</sup>

```
1 int main(int argc, char **argv)
2 {
3     int s;
4     int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
5
6     struct addrinfo hints, *result;
7     memset(&hints, 0, sizeof(struct addrinfo));
8     hints.ai_family = AF_INET; /* IPv4 only */
9     hints.ai_socktype = SOCK_STREAM; /* TCP */
10
11    s = getaddrinfo("www.illinois.edu", "80", &hints, &result);
12    if (s != 0) {
13        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
14        exit(1);
15    }
16
17    if(connect(sock_fd, result->ai_addr, result->ai_addrlen) == -1){
18        perror("connect");
19        exit(2);
20    }
21
22    // For this trivial demo just assume write() sends all bytes in one go and is not
23    // interrupted
24    write(sock_fd, "hi!", 3);
25    char resp[1000];
26    int len = read(sock_fd, resp, 999);
27    resp[len] = '\0';
28    printf("%s\n", resp);
29
30    return 0;
}
```

---

<sup>4</sup>[https://github.com/angrave/SystemProgramming/wiki/Networking,  
-Part-3:-Building-a-simple-TCP-Client](https://github.com/angrave/SystemProgramming/wiki/Networking,-Part-3:-Building-a-simple-TCP-Client)

Connecting from a client to a server.

Step 4. `read( )` and `write( )` data using the socket.

# read()

By default, a blocking call

- if there is data that has already been received by the network stack, then read will return immediately with it
  - thus, read might return with less data than you asked for
- if there is no data waiting for you, by default `read( )` will block until some arrives
  - pop quiz: how might this cause deadlock?

# write()

By default, a blocking call

- but, in a more sneaky way
- when `write( )` returns, the receiver (i.e., the other end of the connection) probably has not yet received the data
  - in fact, the data might not have been sent on the network yet!
  - `write( )` enqueues your data in a send buffer in the OS, and then returns; the OS will transmit the data in the background
- if there is no more space left in the send buffer, by default `write( )` will block
  - how might this cause deadlock?

# read/write example

```
1 int main(int argc, char **argv)
2 {
3     int s;
4     int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
5
6     // Omitted for brevity... get IP address!!
7
8     s = getaddrinfo("www.illinois.edu", "80", &hints, &result);
9     if (s != 0) {
10         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
11         exit(1);
12     }
13
14     if(connect(sock_fd, result->ai_addr, result->ai_addrlen) == -1){
15         perror("connect");
16         exit(2);
17     }
18     char *buffer = "GET / HTTP/1.0\r\n\r\n";
19     printf("SENDING: %s", buffer);
20     printf("===\r\n");
21     // For this trivial demo just assume write() sends all bytes in one go and is not
22     // interrupted
23     write(sock_fd, buffer, strlen(buffer));
24     // For this trivial demo just assume write() sends all bytes in one go and is not
25     // interrupted
26     write(sock_fd, "hi!", 3);
27     char resp[1000];
28     int len = read(sock_fd, resp, 999);
29     resp[len] = '\0';
30     printf("%s\n", resp);
31 }
```



Connecting from a client to a server.

Step 5. `close( )` the socket.

## Section 5

### Server

# Servers

Pretty similar to clients, but with additional steps - there are seven steps:

- ➊ figure out the address and port on which to listen

# Servers

Pretty similar to clients, but with additional steps - there are seven steps:

- ① figure out the address and port on which to listen
- ② create a socket

# Servers

Pretty similar to clients, but with additional steps - there are seven steps:

- ① figure out the address and port on which to listen
- ② create a socket
- ③ bind the socket to the address and port on which to listen

# Servers

Pretty similar to clients, but with additional steps - there are seven steps:

- ① figure out the address and port on which to listen
- ② create a socket
- ③ bind the socket to the address and port on which to listen
- ④ indicate that the socket is a listening socket

# Servers

Pretty similar to clients, but with additional steps - there are seven steps:

- ① figure out the address and port on which to listen
- ② create a socket
- ③ bind the socket to the address and port on which to listen
- ④ indicate that the socket is a listening socket
- ⑤ accept a connection from a client

# Servers

Pretty similar to clients, but with additional steps - there are seven steps:

- ① figure out the address and port on which to listen
- ② create a socket
- ③ bind the socket to the address and port on which to listen
- ④ indicate that the socket is a listening socket
- ⑤ accept a connection from a client
- ⑥ read and write to that connection

# Servers

Pretty similar to clients, but with additional steps - there are seven steps:

- ① figure out the address and port on which to listen
- ② create a socket
- ③ bind the socket to the address and port on which to listen
- ④ indicate that the socket is a listening socket
- ⑤ accept a connection from a client
- ⑥ read and write to that connection
- ⑦ close the connection

Accepting a connection from a client:

- ① Figure out the address and port on which to listen.
- ② Create a socket.
- ③ Bind the socket to the address and port on which to listen.
- ④ Indicate that the socket is a listening socket.

# Servers

Servers can have multiple IP addresses

- “multihomed”
- usually have at least one externally visible IP address, as well as a local-only address (127.0.0.1)

When you bind a socket for listening, you can:

- specify that it should listen on all addresses
  - by specifying the address “INADDR\_ANY” or `in6addr_any` — `0.0.0.0` or `::` (i.e., all 0's)
- specify that it should listen on a particular address

# bind()

The `bind( )` system call associates with a socket:

- an address family
  - AF\_INET: IPv4
  - AF\_INET6: IPv6 (also handles IPv4 clients on POSIX systems)
- a local IP address
  - the special IP address INADDR\_ANY (0.0.0.0) means “all local IPv4 addresses of this host”
  - use in6addr\_any (instead of INADDR\_ANY) for IPv6
- a local port number

# listen()

The `listen( )` system call tells the OS that the socket is a listening socket to which clients can connect

- you also tell the OS how many pending connections it should queue before it starts to refuse new connections
  - you pick up a pending connection with `accept( )`
- when `listen` returns, remote clients can start connecting to your listening socket
  - you need to `accept( )` those connections to start using them

# Server socket, bind, listen

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netdb.h>
7 #include <unistd.h>
8 #include <arpa/inet.h>
9
10 int main(int argc, char **argv)
11 {
12     int s;
13     int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
14
15     struct addrinfo hints, *result;
16     memset(&hints, 0, sizeof(struct addrinfo));
17     hints.ai_family = AF_INET;
18     hints.ai_socktype = SOCK_STREAM;
19     hints.ai_flags = AI_PASSIVE;
20
21     s = getaddrinfo(NULL, "1234", &hints, &result);
22     if (s != 0) {
23         fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
24         exit(1);
25     }
26
27     if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
28         perror("bind()");
29         exit(1);
30     }
31
32     if (listen(sock_fd, 10) != 0) {
33         perror("listen()");
34         exit(1);
35     }
36
37     struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
```

# Server socket, bind, listen (pg 2)

```
1 int main(int argc, char **argv)
2 {
3     ....
4
5     if (listen(sock_fd, 10) != 0) {
6         perror("listen()");
7         exit(1);
8     }
9
10    struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
11    printf("Listening on file descriptor %d, port %d\n", sock_fd, ntohs(result_addr->sin_port))
12    ;
13
14    printf("Waiting for connection...\n");
15    int client_fd = accept(sock_fd, NULL, NULL);
16    printf("Connection made: client_fd=%d\n", client_fd);
17
18    char buffer[1000];
19    int len = read(client_fd, buffer, sizeof(buffer) - 1);
20    buffer[len] = '\0';
21
22    printf("Read %d chars\n", len);
23    printf("===%n");
24    printf("%s\n", buffer);
25
26    return 0;
}
```

## Accepting a connection from a client

- ⑤ accept( ) a connection from a client.
- ⑥ read( ) and write( ) to the client.
- ⑦ close( ) the connection.

# accept()

The `accept( )` system call waits for an incoming connection, or pulls one off the pending queue

- it returns an active, ready-to-use socket file descriptor connected to a client
- it returns address information about the peer
  - use `inet_ntop( )` to get the client's printable IP address
  - use `getnameinfo( )` to do a reverse DNS lookup on the client

# Server accept, read/write, close

See `kk_server.c` on the next two slides for reference...

```

1 //Includes omitted for space
2 int main(int argc, char **argv)
3 {
4     int s;
5     // Step 1: Get address stuff
6     struct addrinfo hints, *result;
7     memset(&hints, 0, sizeof(struct addrinfo));
8     hints.ai_family = AF_INET;
9     hints.ai_socktype = SOCK_STREAM;
10    hints.ai_flags = AI_PASSIVE;
11    s = getaddrinfo(NULL, "1234", &hints, &result);
12    if (s != 0) {
13        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
14        exit(1);
15    }
16    // Step 2: Open socket
17    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
18    // Step 3: Bind socket
19    if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
20        perror("bind()");
21        exit(1);
22    }
23    // Step 4: Listen on the socket
24    if (listen(sock_fd, 10) != 0) {
25        perror("listen()");
26        exit(1);
27    }
28    struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
29    printf("Listening on file descriptor %d, port %d\n", sock_fd, ntohs(result_addr->sin_port));
30    // Step 5: Accept connection
31    printf("Waiting for connection...\n");
32    int client_fd = accept(sock_fd, NULL, NULL);
33    printf("Connection made: client_fd=%d\n", client_fd);
34    // Step 6: Read, then write if you want
35    char buffer[1000];
36    int len = read(client_fd, buffer, sizeof(buffer) - 1);
37    buffer[len] = '\0';
38    printf("Read %d chars\n", len);
39    printf("===\n");
40    printf("%s\n", buffer);
41    // Step 7: Close
42    close(client_fd);

```

## 1 Knock Knock Example

- DNS

## 2 OSI Model

## 3 Networks and Sockets

## 4 Client

- IP Addresses
- DNS
- Create a socket

## 5 Server

- Knock Knock Server
- Knock Knock Client

# Knock Knock Server

Knock-Knock shows the back and forth between a client and server telling a knock knock joke.

- Server consists of functions:

- do\_open ()
- receive\_message ()
- send\_message ()

```
1 int do_open() {
2     int s;
3     // Step 1: Get Address stuff
4     struct addrinfo hints, *result;
5
6     // Setting up the hints struct...
7     memset(&hints, 0, sizeof(struct addrinfo));
8     hints.ai_family = AF_INET;
9     hints.ai_socktype = SOCK_STREAM;
10    hints.ai_flags = AI_PASSIVE;
11    s = getaddrinfo(NULL, "1414", &hints, &result);
12    if (s != 0) {
13        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
14        exit(1);
15    }
16
17    // Step 2: Create the socket
18    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
19
20    // Step 3: Bind the socket
21    if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
22        perror("bind()");
23        exit(1);
24    }
25
26    // Step 4: Listen
27    if (listen(sock_fd, 10) != 0) {
28        perror("listen()");
29        exit(1);
30    }
31
32    struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
33    printf("Listening on file descriptor %d, port %d\n", sock_fd, ntohs(result_addr->sin_port));
34
35    // Step 5: Accept a connection
36    printf("Waiting for connection...\n");
37    int client_fd = accept(sock_fd, NULL, NULL);
38    printf("Connection made: client_fd=%d\n", client_fd);
39
40    return client_fd;
41}
```

```
1 #include <string.h>
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netdb.h>
7 #include <unistd.h>
8 #include <arpa/inet.h>
9
10 int do_open() {
11     // See def on previous slide...
12 }
13
14 void recieve_message(int client_fd){
15     char buffer[1000];
16     int len = read(client_fd, buffer, sizeof(buffer) - 1);
17     buffer[len] = '\0';
18     printf("SERVER RECEIVED: %s \n", buffer);
19 }
20
21 void send_message(char *msg, int sock_fd){
22     printf("SERVER SENDING: %s", msg);
23     printf("===\n");
24     write(sock_fd, msg, strlen(msg));
25 }
26
27 int main(int argc, char **argv) {
28
29     int client_socket_id = do_open();
30
31     // Step 6: Read and write
32     recieve_message(client_socket_id);
33
34     send_message("Who's there?", client_socket_id);
35
36     // Step 7: Close
37     close(client_socket_id);
38
39     return 0;
40 }
```

# Knock Knock Client

Knock-Knock shows the back and forth between a client and server telling a knock knock joke.

- Client consists of functions:

- do\_connect ()
- read\_response ()
- send\_message ()

```

1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netdb.h>
7 #include <unistd.h>
8
9 int do_connect(char *host, char *port){
10    int s;
11    int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
12
13    struct addrinfo hints, *result;
14
15    // Allows "global"
16    memset(&hints, 0, sizeof(struct addrinfo));
17    hints.ai_family = AF_INET; /* IPv4 only */
18    hints.ai_socktype = SOCK_STREAM; /* TCP */
19
20    s = getaddrinfo(host, port, &hints, &result);
21
22    // If I can't get the address, write an error.
23    if (s != 0) {
24        fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
25        exit(1);
26    }
27
28    // Try to connect; if I can't, write an error.
29    if(connect(sock_fd, result->ai_addr, result->ai_addrlen) == -1){
30        perror("connect");
31        exit(2);
32    }
33    else{
34        printf("Connection is good!\n");
35    }
36
37    return sock_fd;
38}

```

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 #include <sys/types.h>
5 #include <sys/socket.h>
6 #include <netdb.h>
7 #include <unistd.h>
8
9 int do_connect(char *host, char *port){
10    // See implementation on the previous slide
11}
12
13 void send_message(char *msg, int sock_fd){
14    printf("SENDING: %s", msg);
15    printf("===\n");
16    write(sock_fd, msg, strlen(msg));
17}
18
19 void read_response(int sock_fd) {
20    // Response
21    char resp[1000];
22    int len = read(sock_fd, resp, 999);
23    resp[len] = '\0';
24
25    printf("RECEIVED: %s\n", resp);
26}
27
28 int main(int argc, char **argv) {
29
30    int sock_fd = do_connect("localhost", "1414");
31
32    send_message("Knock knock", sock_fd);
33
34    read_response(sock_fd);
35
36    close(sock_fd);
37
38    return 0;
39}
```

# Something to note...

Our server code is not concurrent

- single thread of execution
- the thread blocks waiting for the next connection
- the thread blocks waiting for the next message from the connection

A crowd of clients is, by nature, concurrent

- while our server is handling the next client, all other clients are stuck waiting for it

# Summary

- Layered OSI model for network communication
- Networks and Sockets
- Looking up IP from DNS
- Using IP & Port to open a socket
- Process for setting up a client
- Process for setting up a server
- Client/Server communication
- Sequential; not concurrent.

## 1 Knock Knock Example

- DNS

## 2 OSI Model

## 3 Networks and Sockets

## 4 Client

- IP Addresses
- DNS
- Create a socket

## 5 Server

- Knock Knock Server
- Knock Knock Client