Networking

CS 5007: Systems

Adrienne Slaughter, Joe Buck

Northeastern University

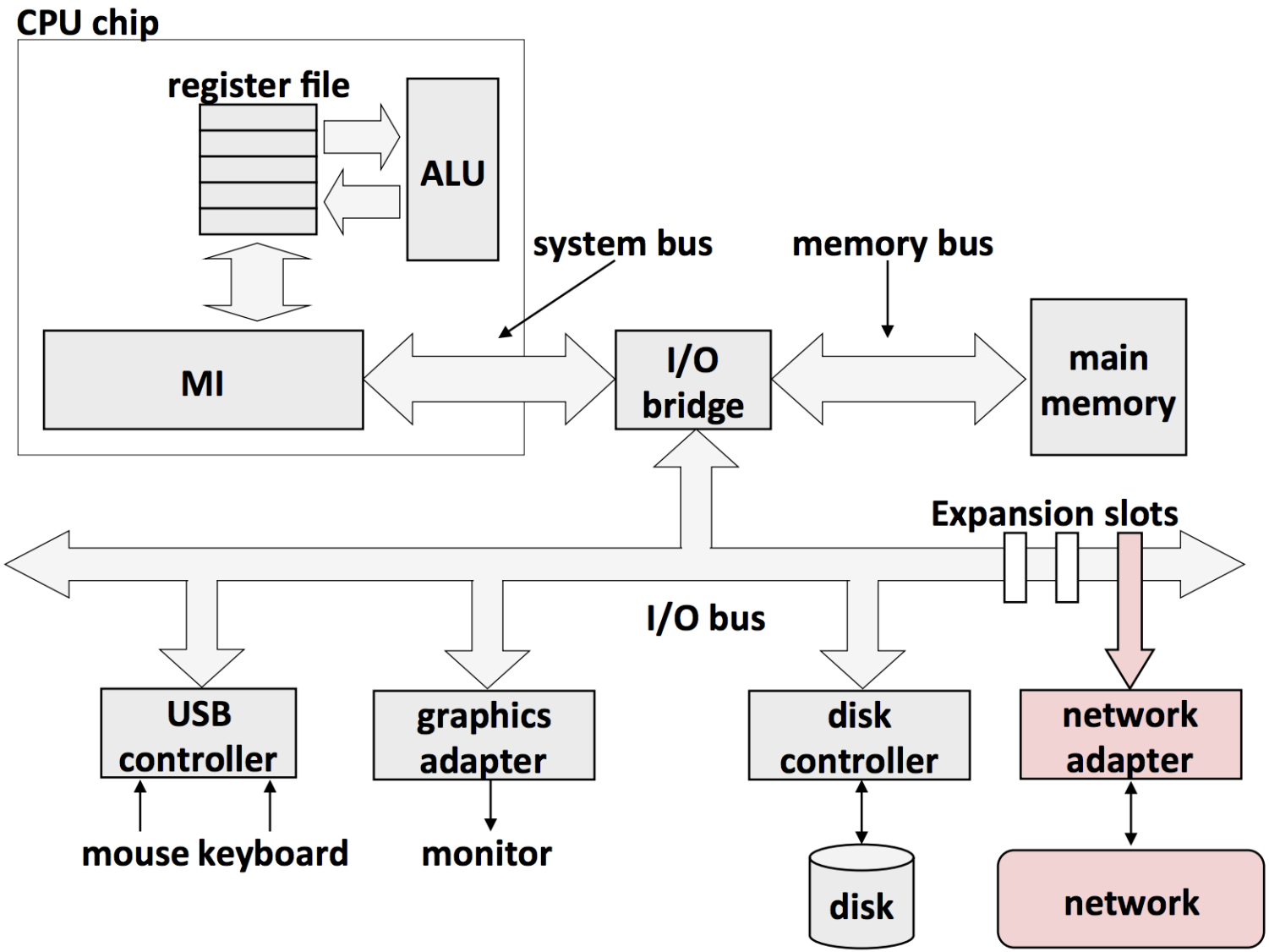
April 10, 2019

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
register file
ALU
system bus
memory bus
MI
I/O bridge
main memory
Expansion slots
I/O bus
USB controller
graphics adapter
disk controller
network adapter
mouse keyboard
monitor
disk
network

Goal: Communicate between two computers

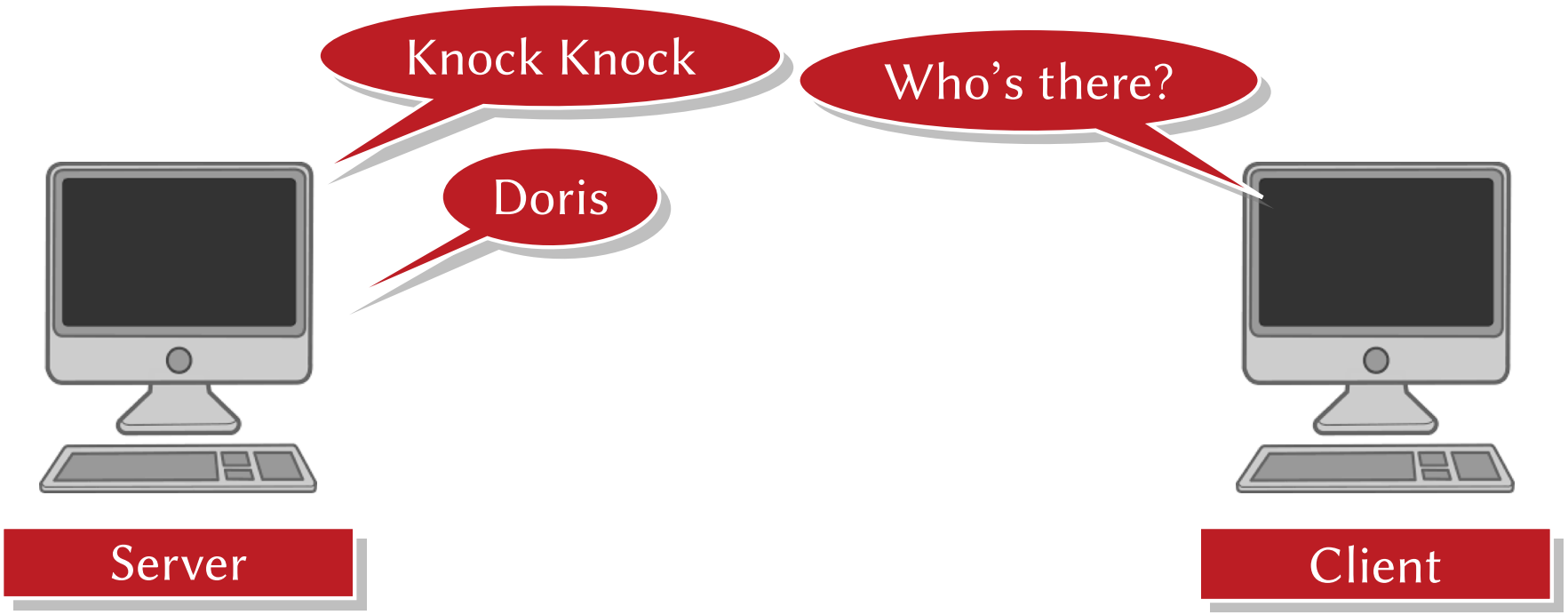
Server

Client

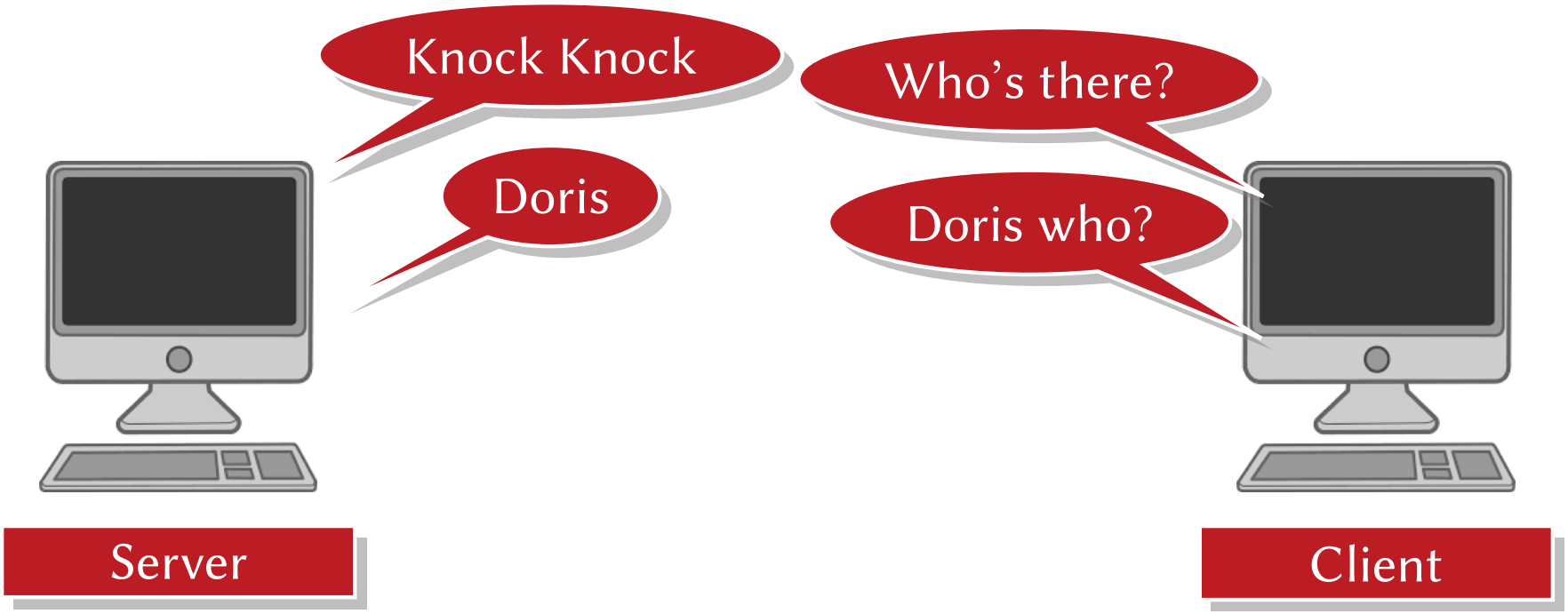
Goal: Communicate between two computers

Goal: Communicate between two computers

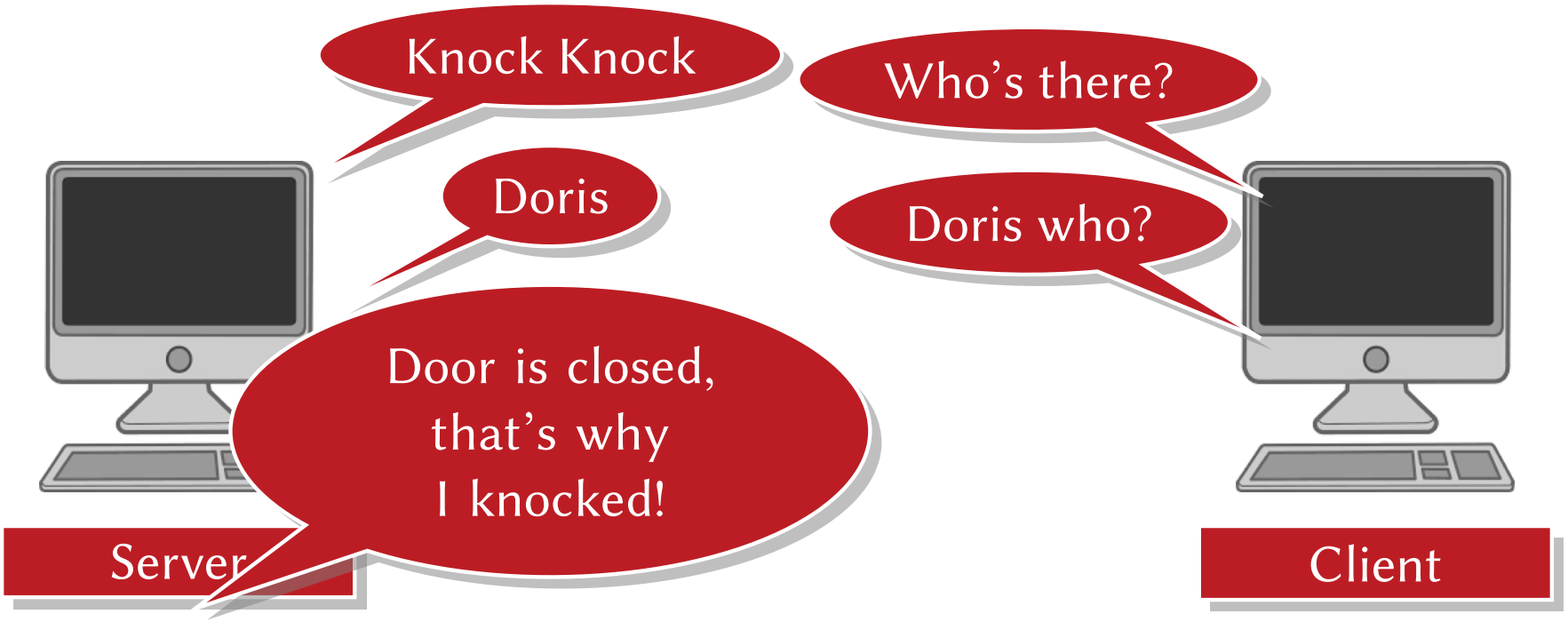
Goal: Communicate between two computers



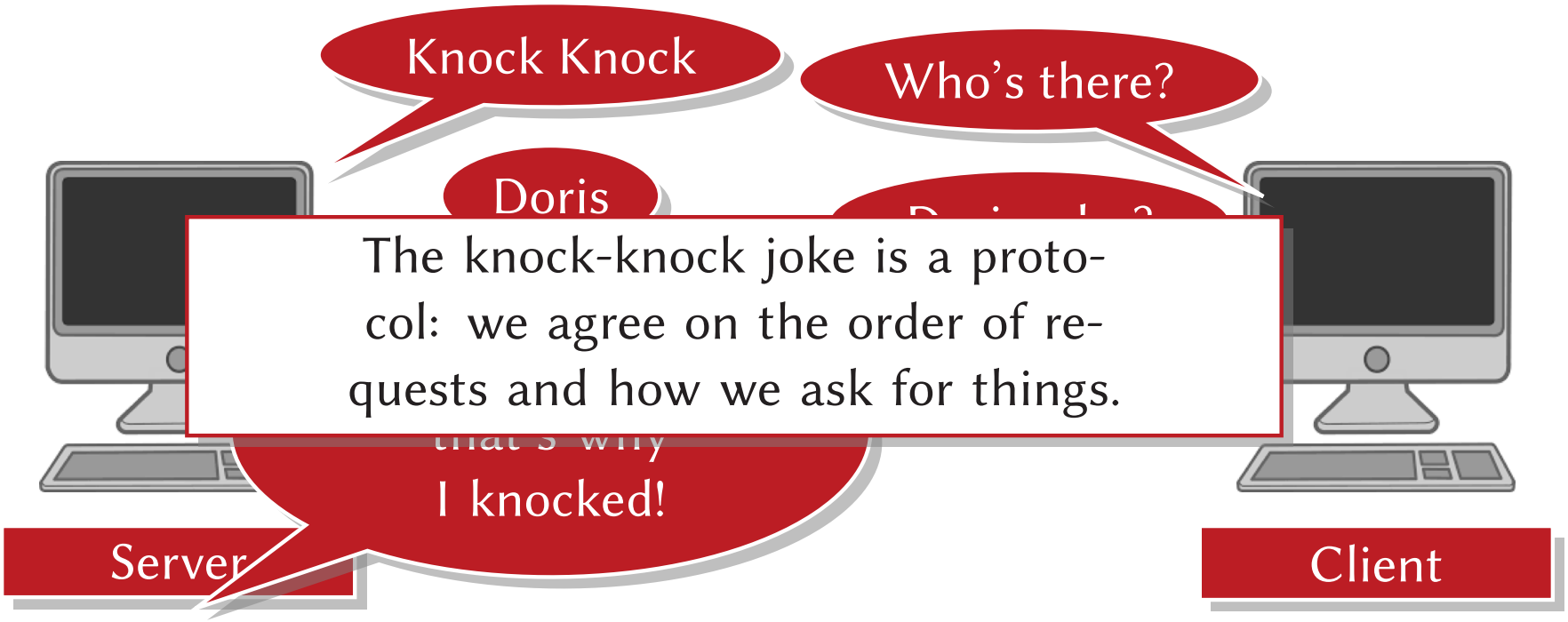
Goal: Communicate between two computers



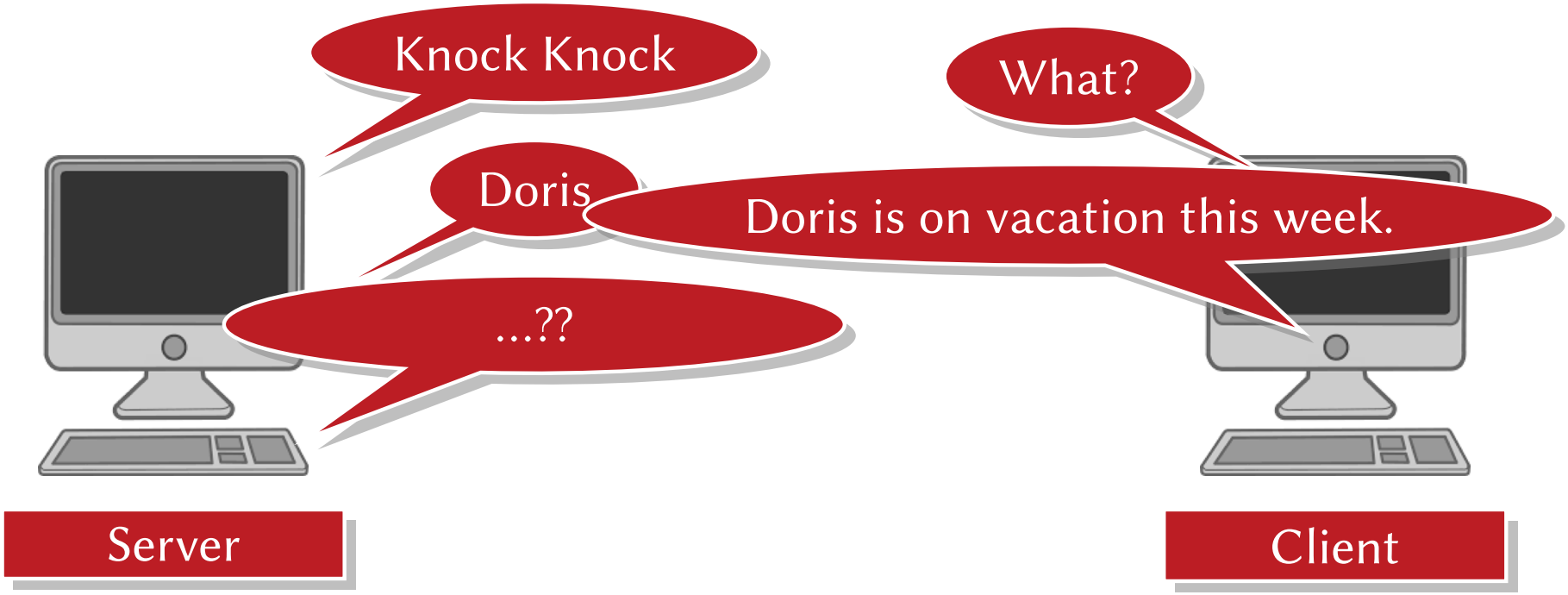
Goal: Communicate between two computers



Goal: Communicate between two computers



Goal: Communicate between two computers



Goal: Communicate between two computers

Webserver

Client: Web Browser

Goal: Communicate between two computers

Client sends URL
to host/server,
specifying which
document: the request

"Give me the red one"

Webserver

Client: Web Browser

Goal: Communicate between two computers

Webserver

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

"Give me the red one"

Client: Web Browser

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?

Application Data

Webserver

Client: Web Browser

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

But how does the data get transferred?

Application Data

Webserver

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

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 ...

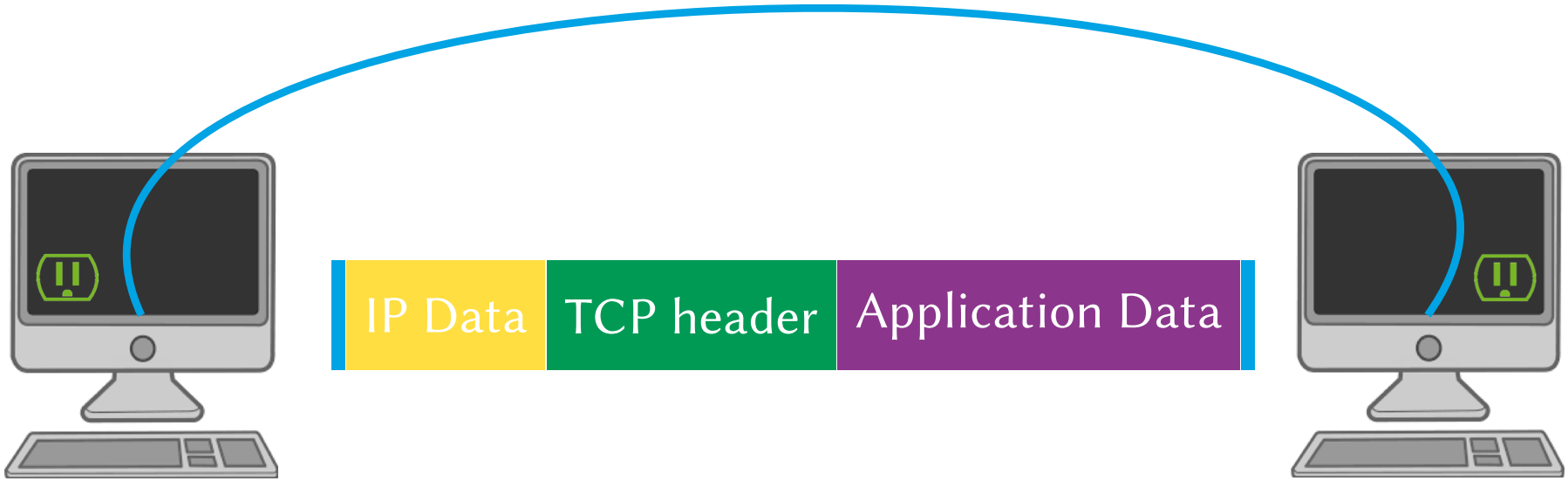
But how does the data get transferred?

Webserver

Client: Web Browser

…wrapped with frame header/footer …

But how does the data get transferred?



…and passed across the wire.

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

Anatomy of a URL

`http://www.theimdbapi.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.theimdbapi.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.theimdbapi.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`

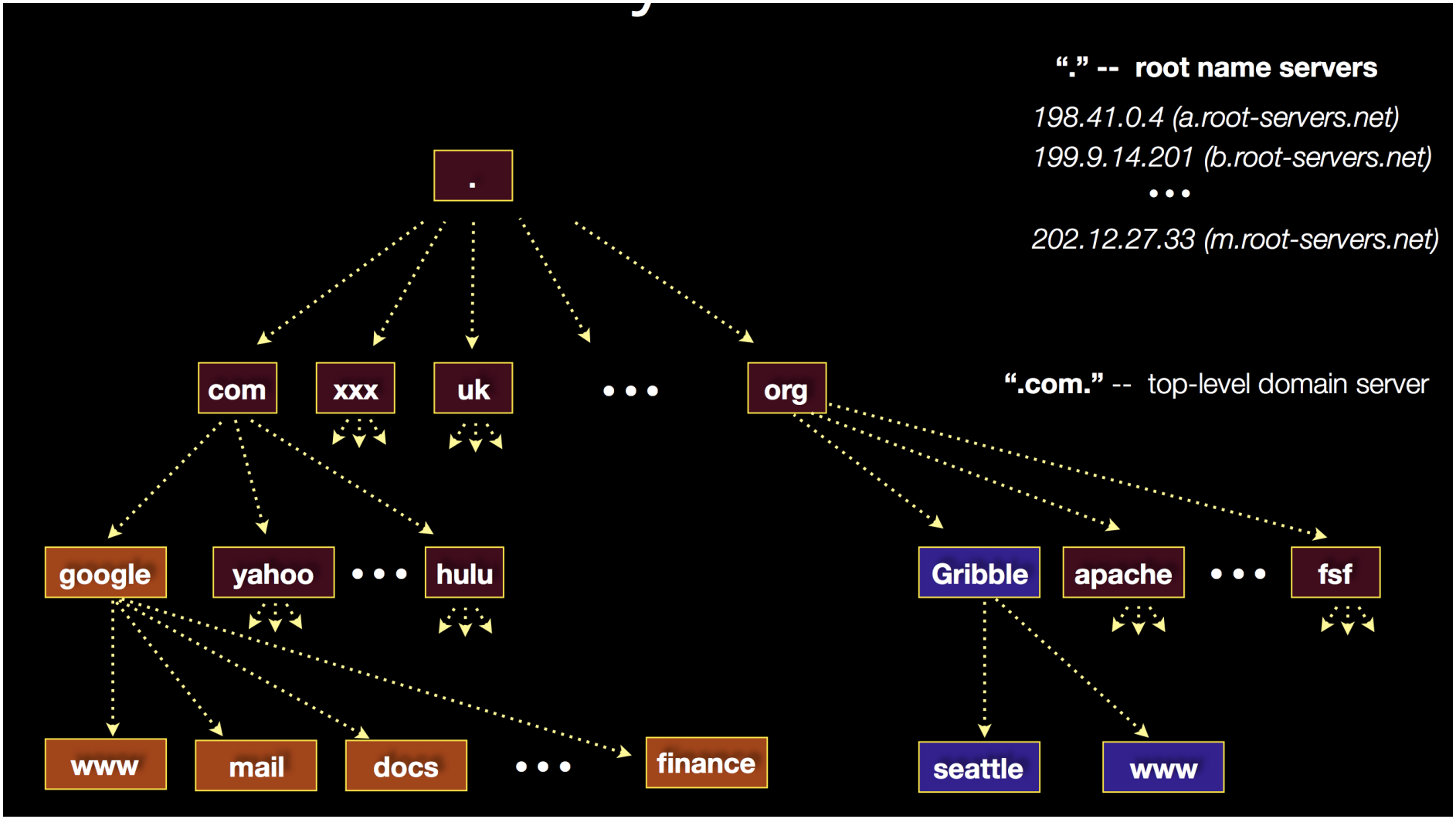
A fun thing to do:

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

This tells me that `www.northeastern.edu` maps to `northeastern.edu.edgekey.net`, which maps to `e122215.dscb.akamaiedge.net`. `e122215.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( )`

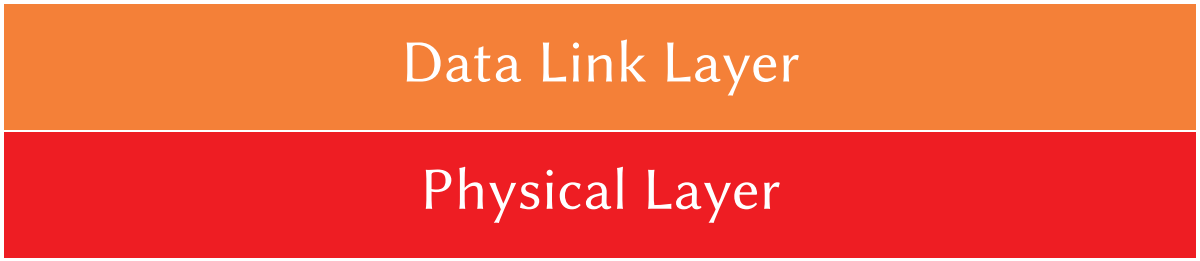
- a pretty complicated system call; the basic idea...
  - set up a "hints" structure with constraints you want respected
    - e.g., IPv6, IPv4, or either
  - tell `getaddrinfo( )` which host and port you want resolved
    - host: a string representation; DNS name or IP address
  - `getaddrinfo( )` gives 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`
-

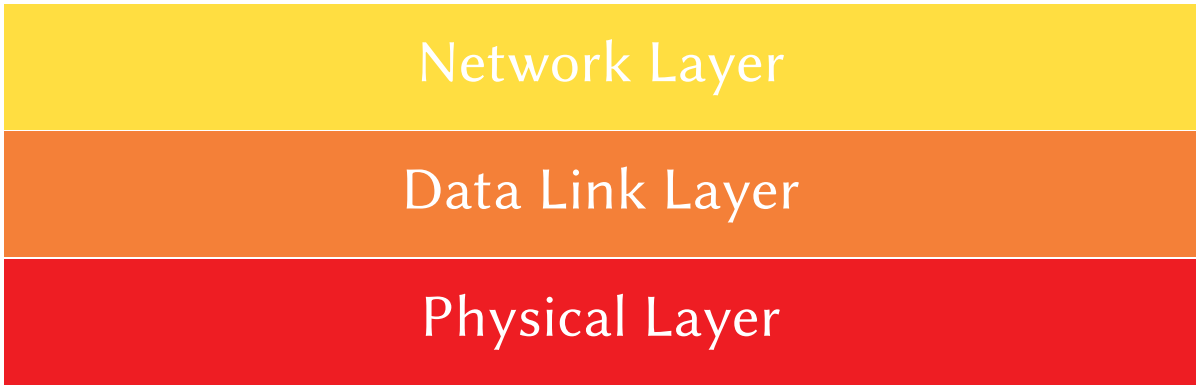
OSI Model

Physical Layer

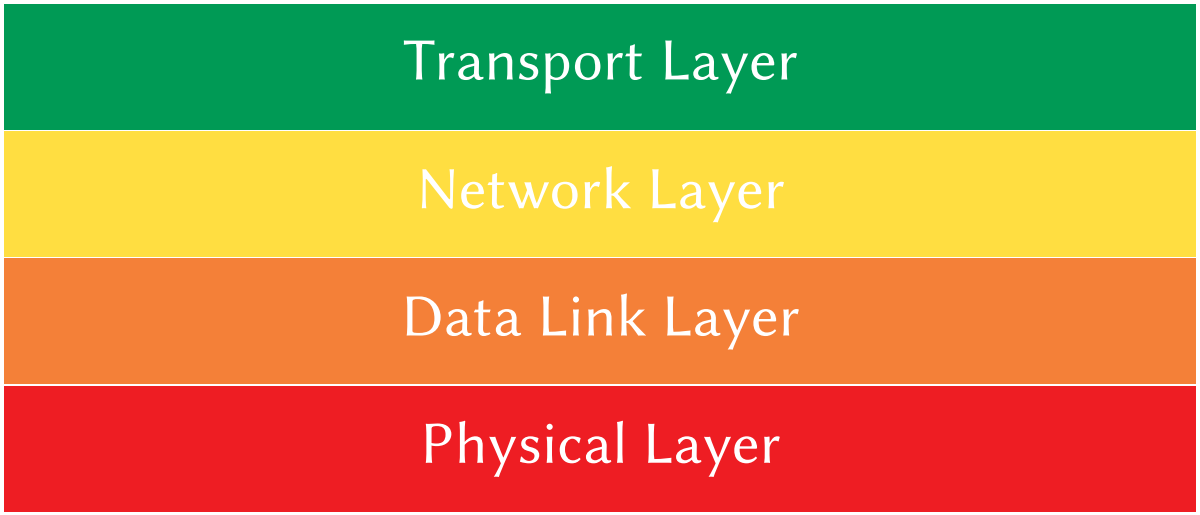
OSI Model



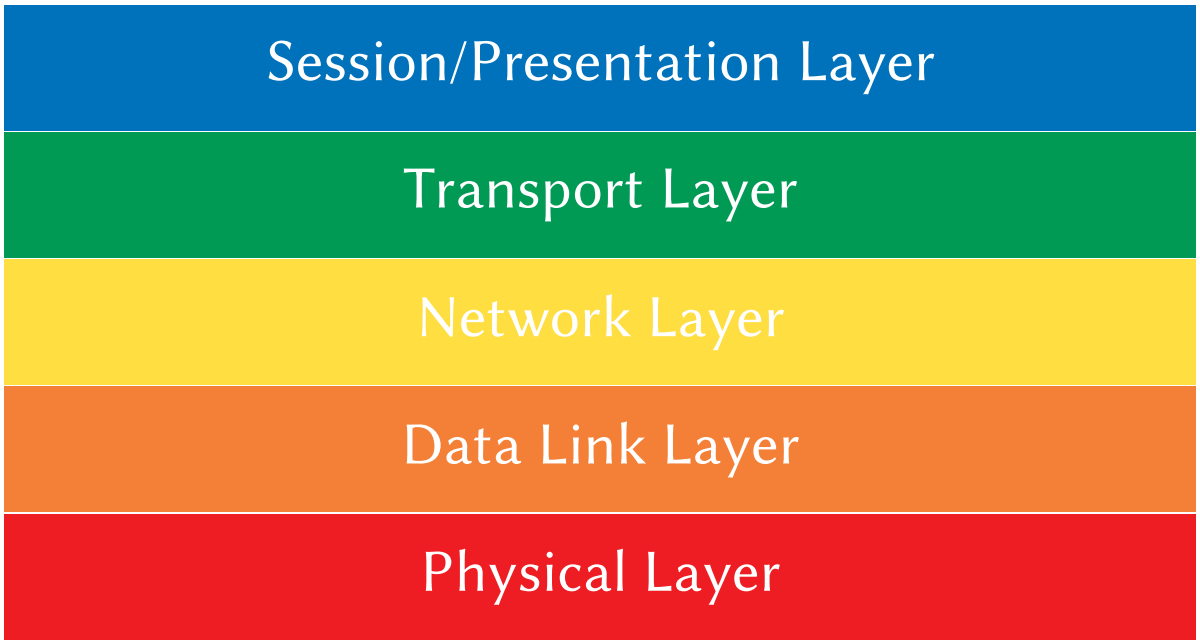
OSI Model



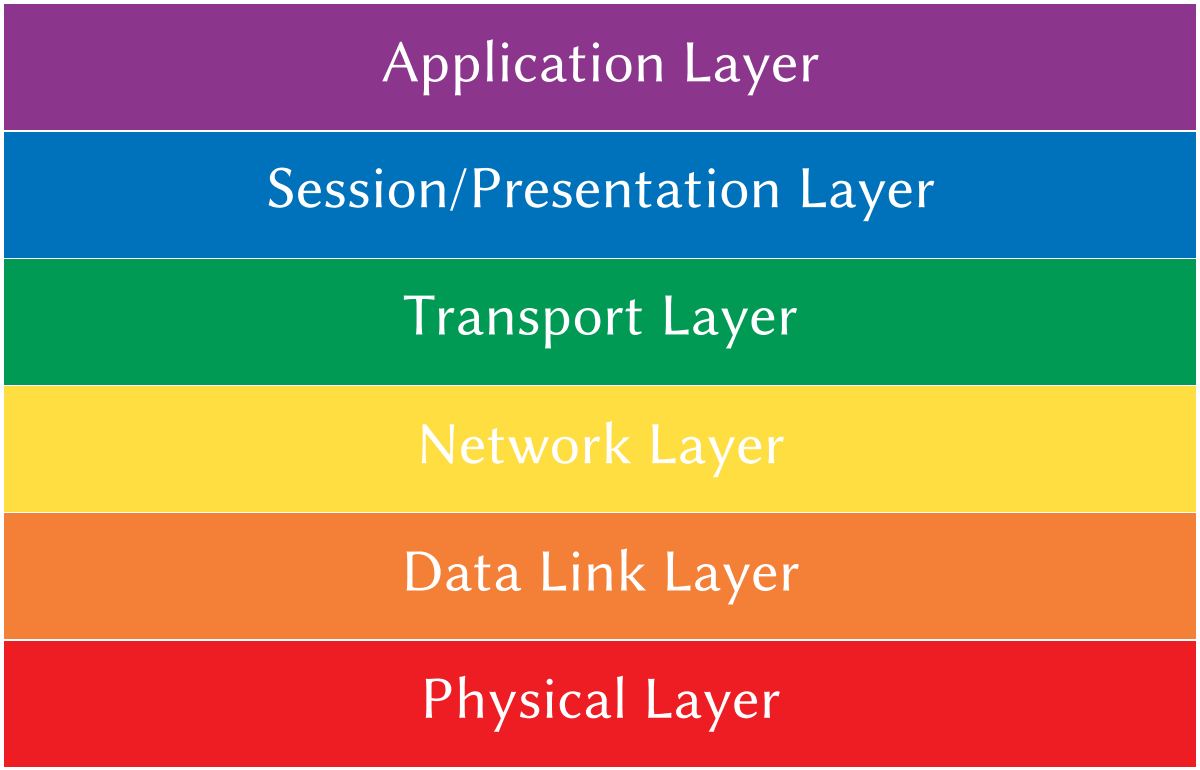
OSI Model


Transport Layer
Network Layer
Data Link Layer
Physical Layer

OSI Model

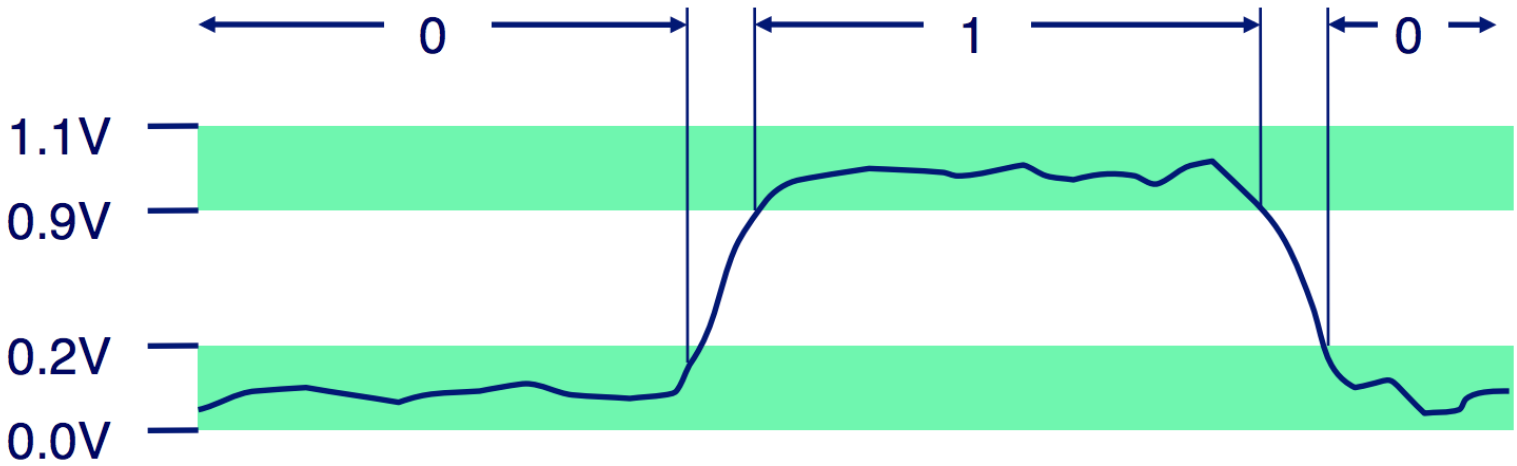

Session/Presentation Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer

OSI Model


Application Layer
Session/Presentation Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer

Starting at the bottom: Everything is Bits[1]

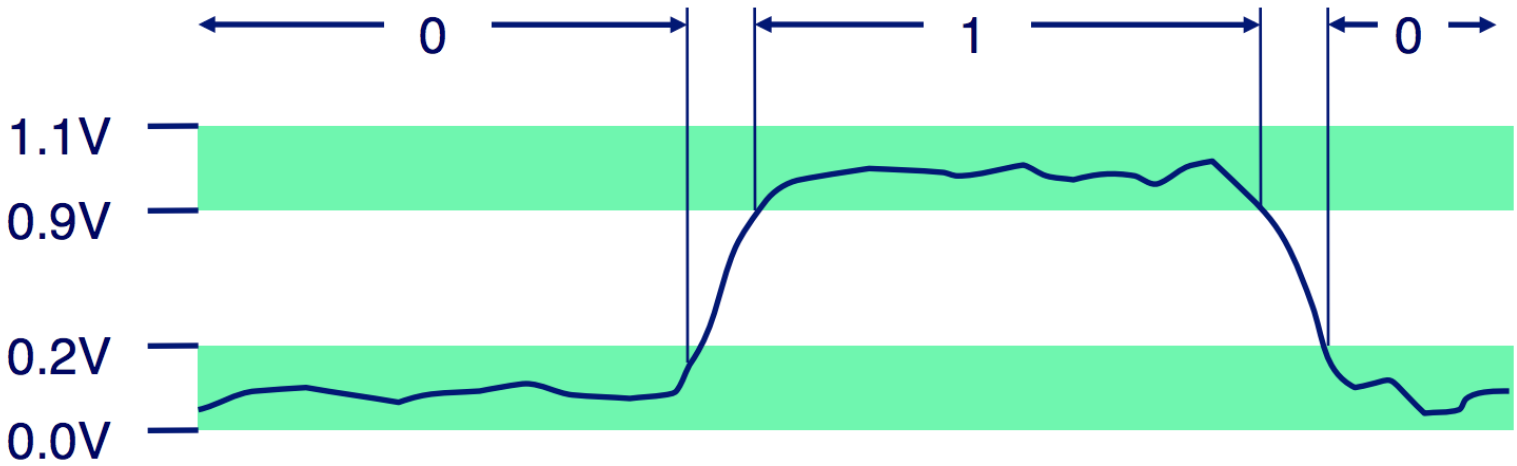
- Each bit is 1 or 0



[1] From Bryant & O'Halloran

Starting at the bottom: Everything is Bits[1]

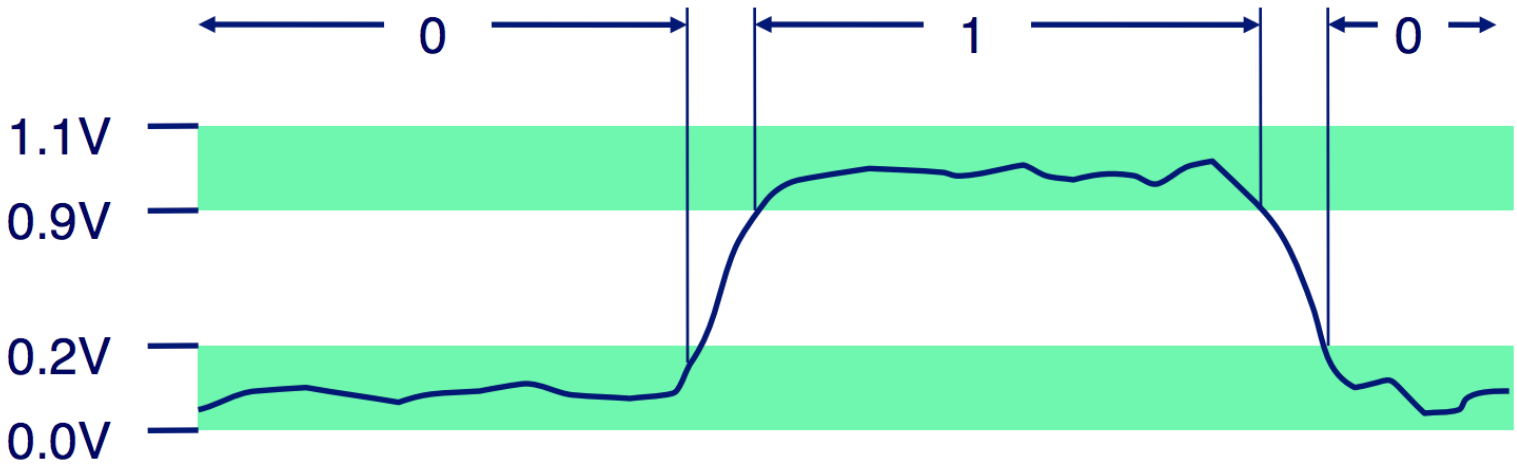
- Each bit is 1 or 0
- By encoding/interpreting sets of bits in various ways



[1] From Bryant & O'Halloran

Starting at the bottom: Everything is Bits[1]

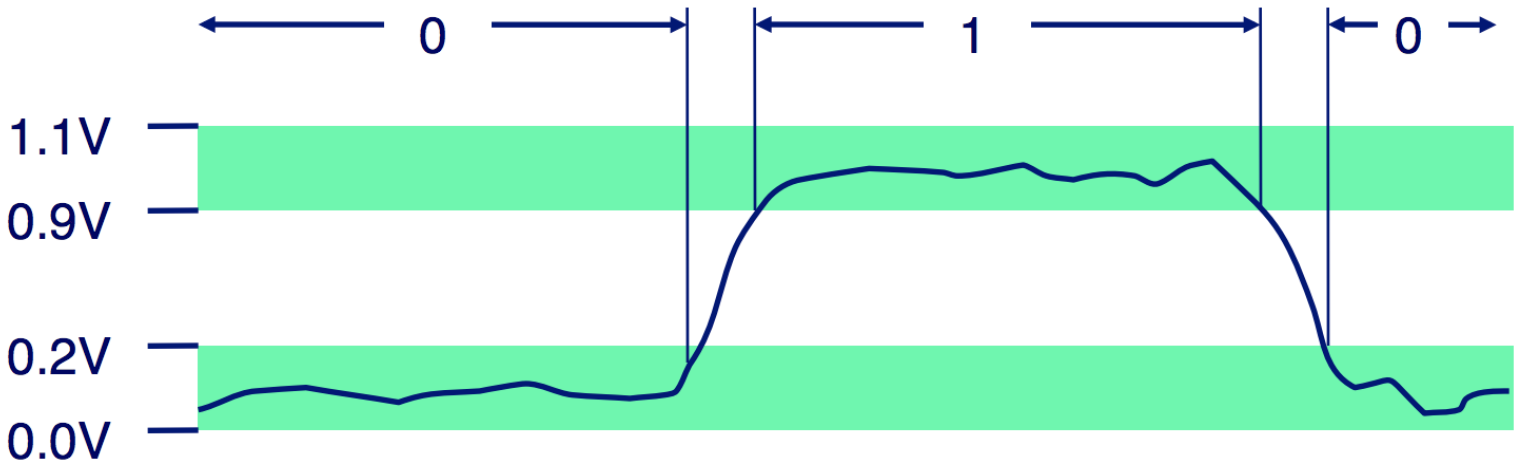
- Each bit is 1 or 0
- By encoding/interpreting sets of bits in various ways
- Why bits? Electronic implementation



[1] From Bryant & O'Halloran

Starting at the bottom: Everything is Bits[1]

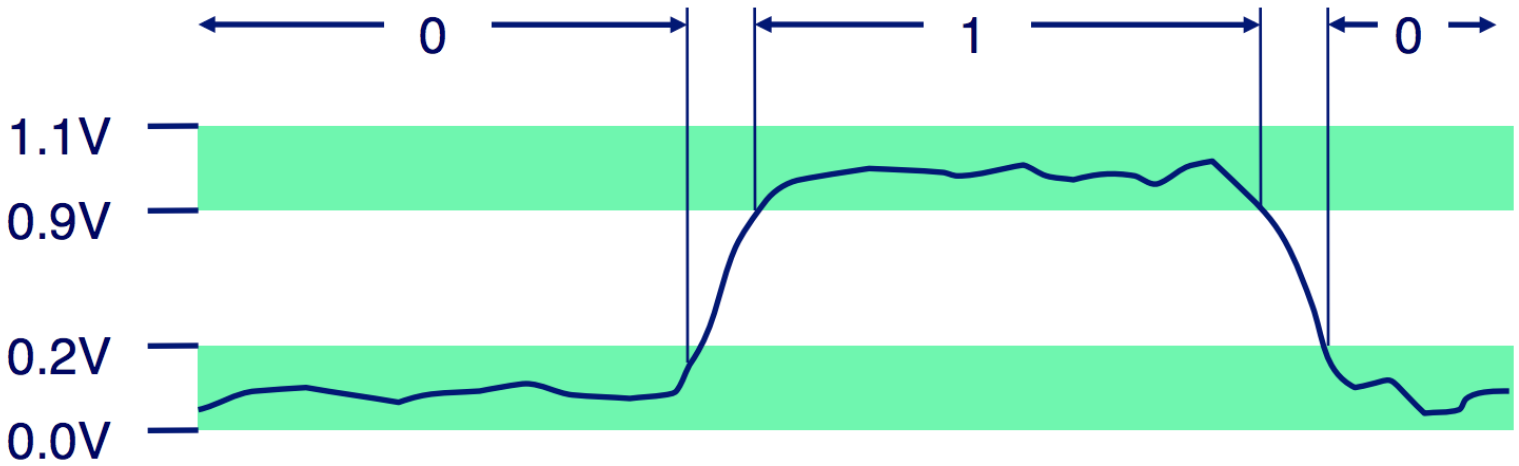
- Each bit is 1 or 0
- By encoding/interpreting sets of bits in various ways
- Why bits? Electronic implementation
  - Easy to store with bistable elements



[1] From Bryant & O'Halloran

Starting at the bottom: Everything is Bits[1]

- 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



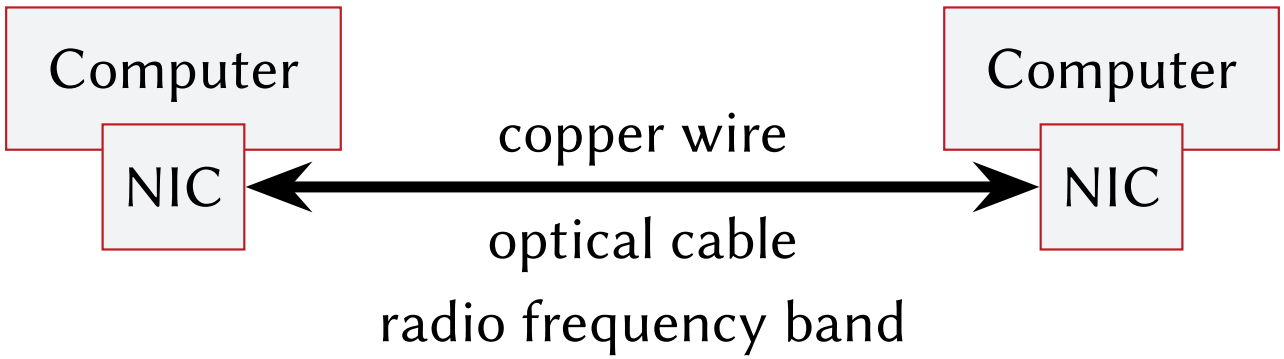
[1] From Bryant & O'Halloran

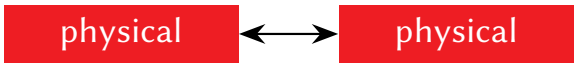
Physical Layer

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?

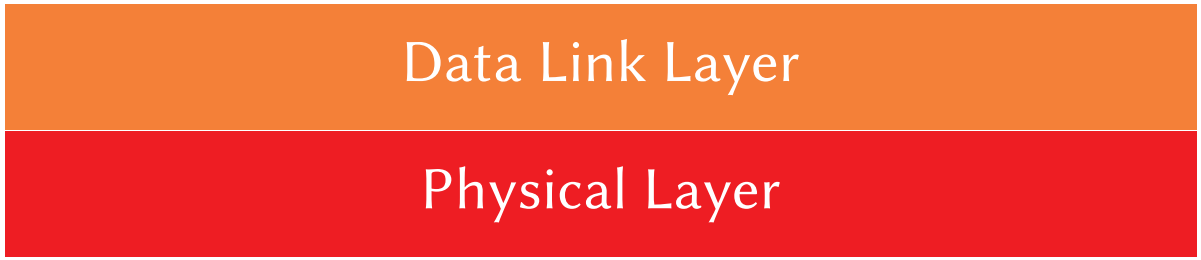
- Wires (on poles or underground) in your neighborhood
- Connect to your house
- Wire goes through the outside wall and into a jack in your living room
- Cable runs from the jack in the wall to a cable modem
- Cable goes from the cable modem to a router
- Ethernet cable goes from the router to my machine–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

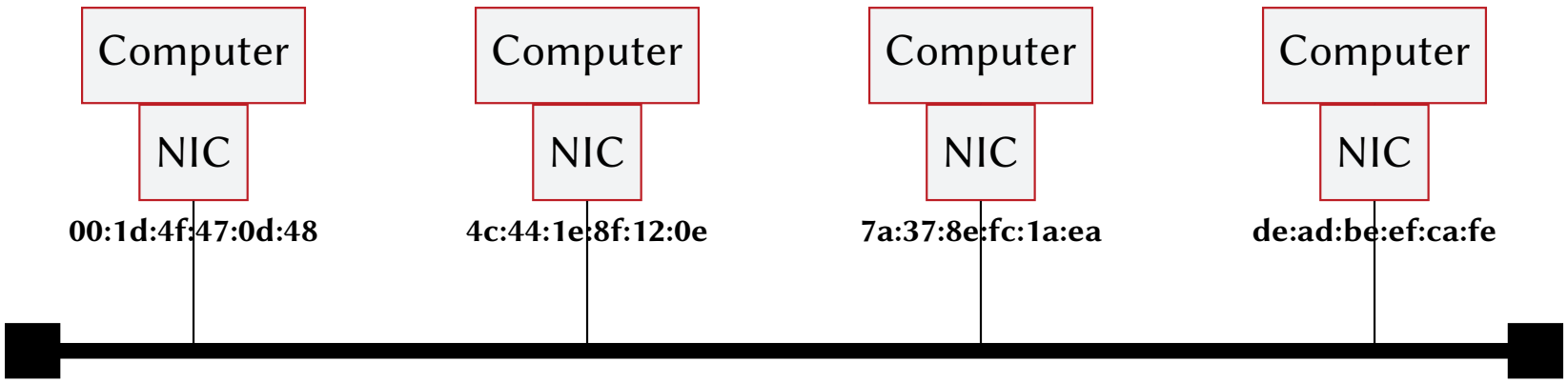
OSI Model

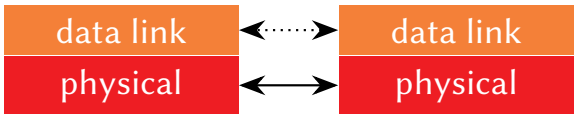


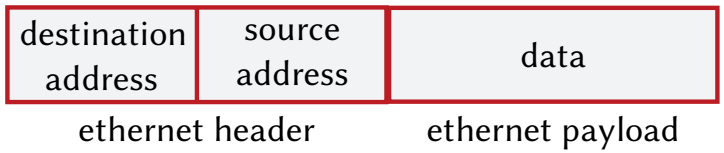
Data Link Layer

Multiple computers on a LAN share the network medium

- media access control (MAC) specifies how computers cooperate



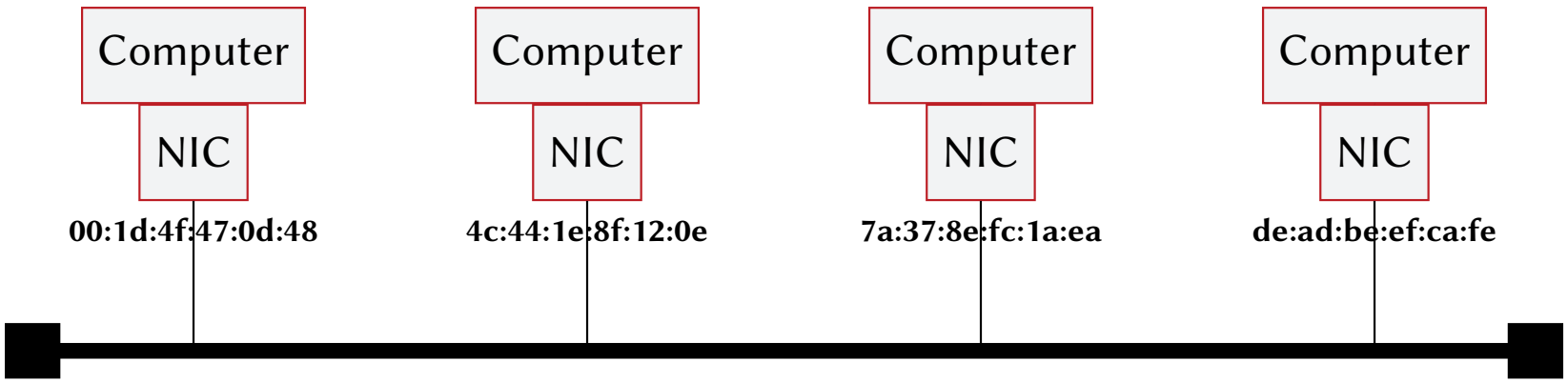


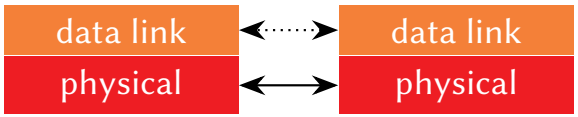


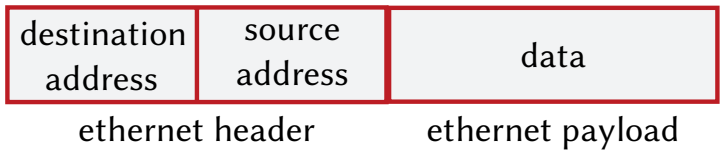
Data Link Layer

Multiple computers on a LAN share the network medium

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



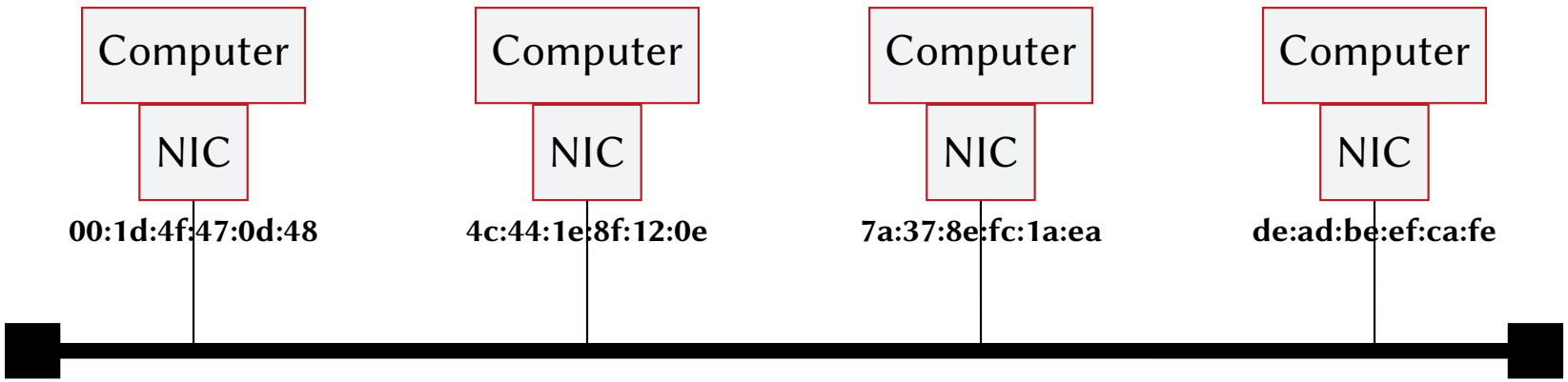


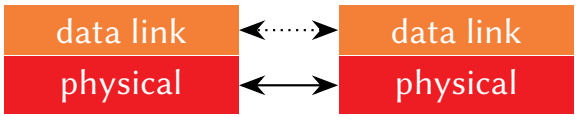


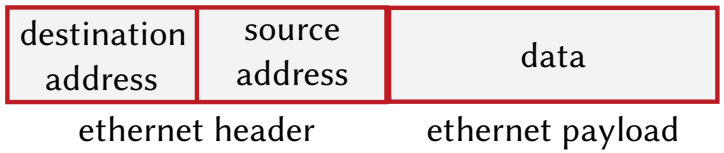
Data Link Layer

Multiple computers on a LAN share the network medium

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



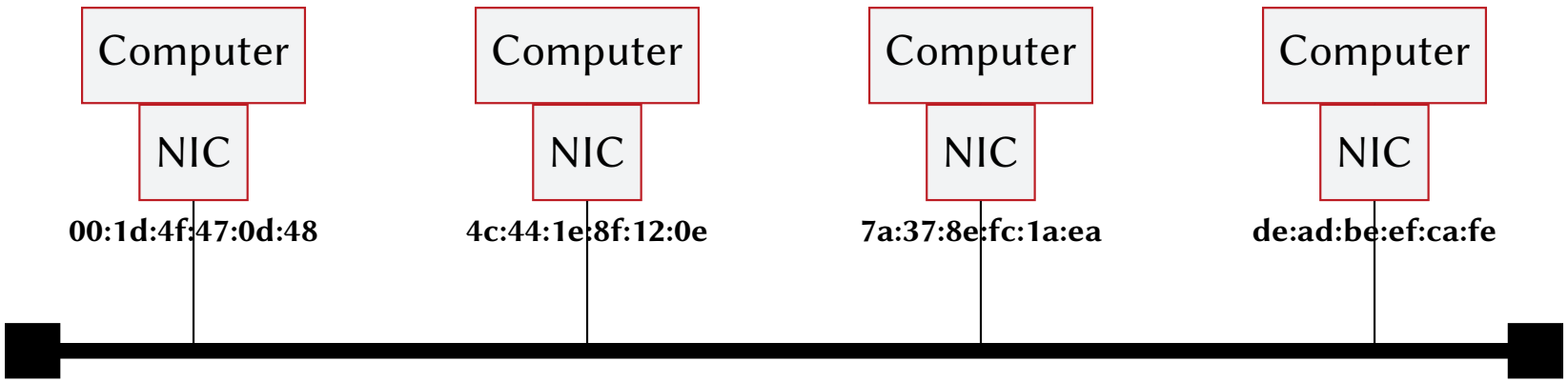


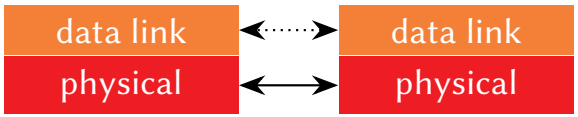


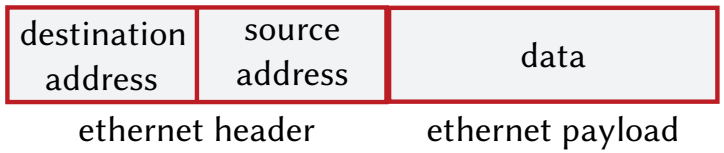
Data Link Layer

Multiple computers on a LAN share the network medium

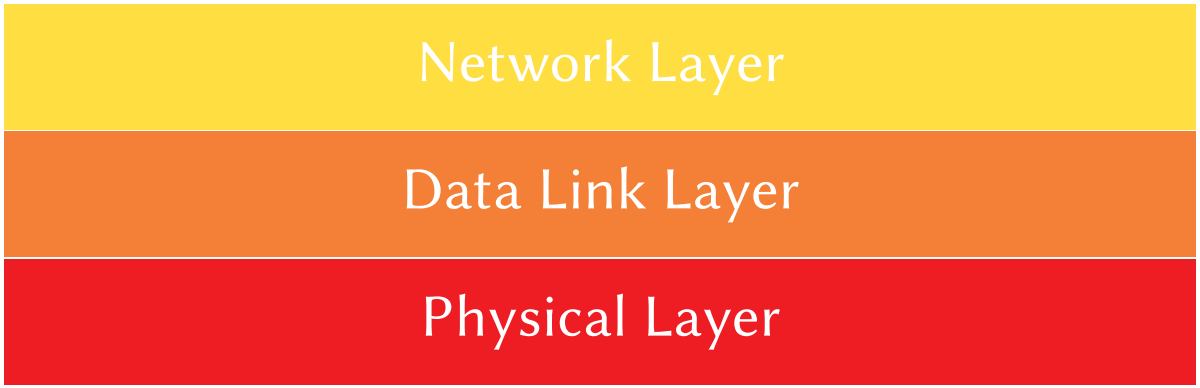
- media access control (MAC) specifies how computers cooperate
  - Every NIC card has it's own MAC address
  - If you have multiple NICs in a machine (ethernet, Wi-Fi), you have multiple MACs
- link layer also specifies how bits are packetized and NICs are addressed







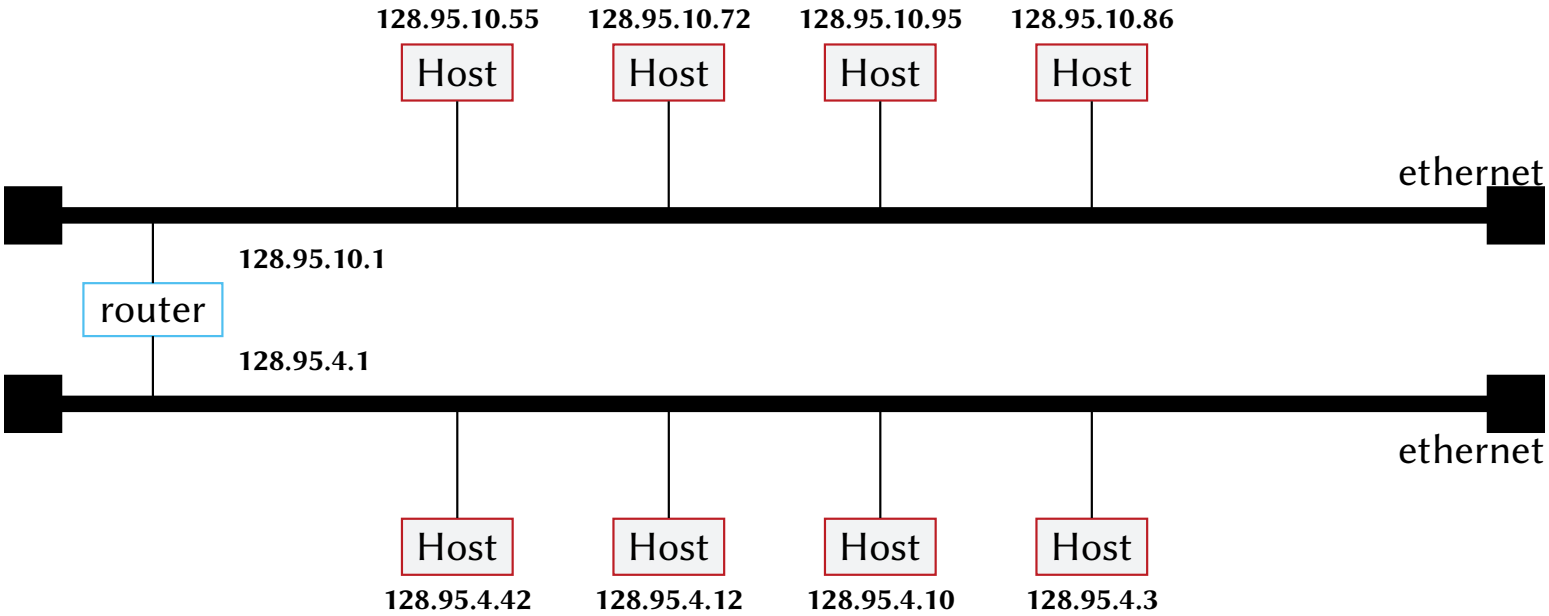
OSI Model

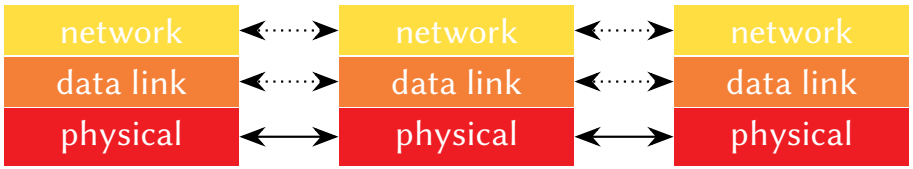


Network Layer

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

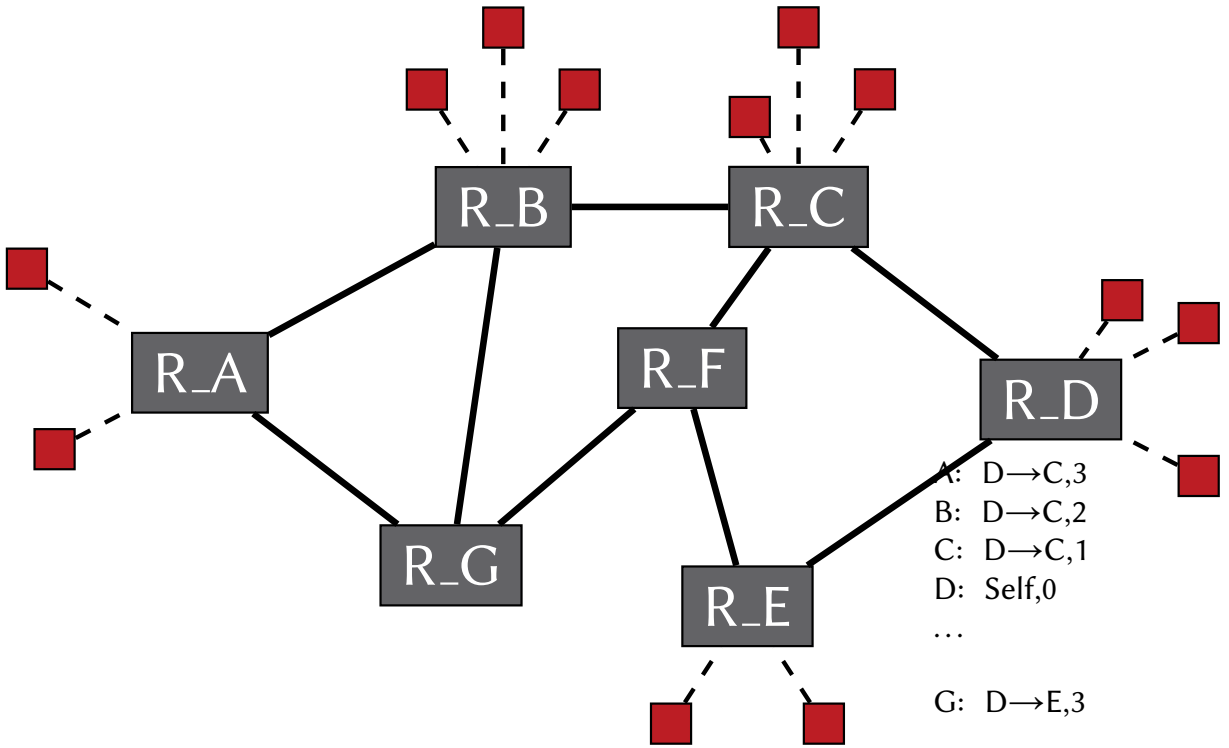




The Network Layer (IP)

Protocols to:

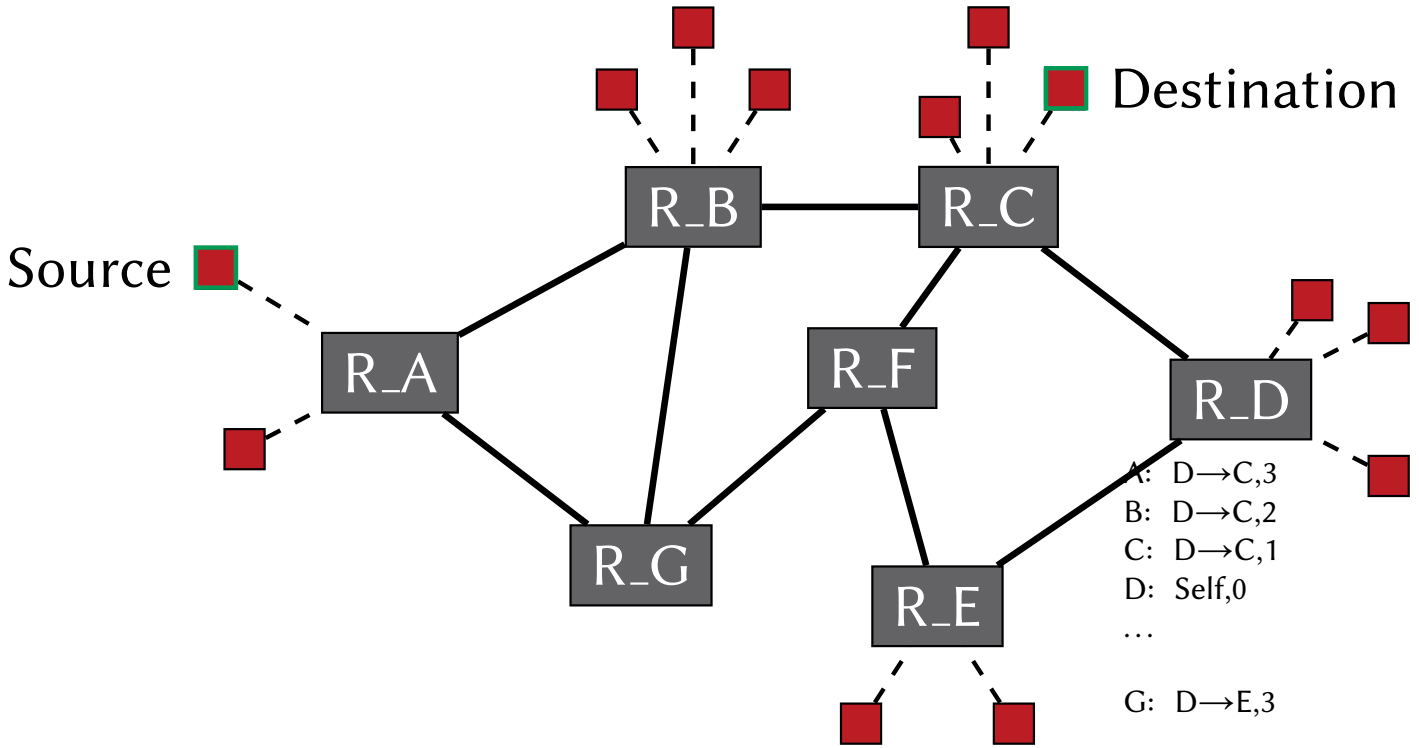
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



The Network Layer (IP)

Protocols to:

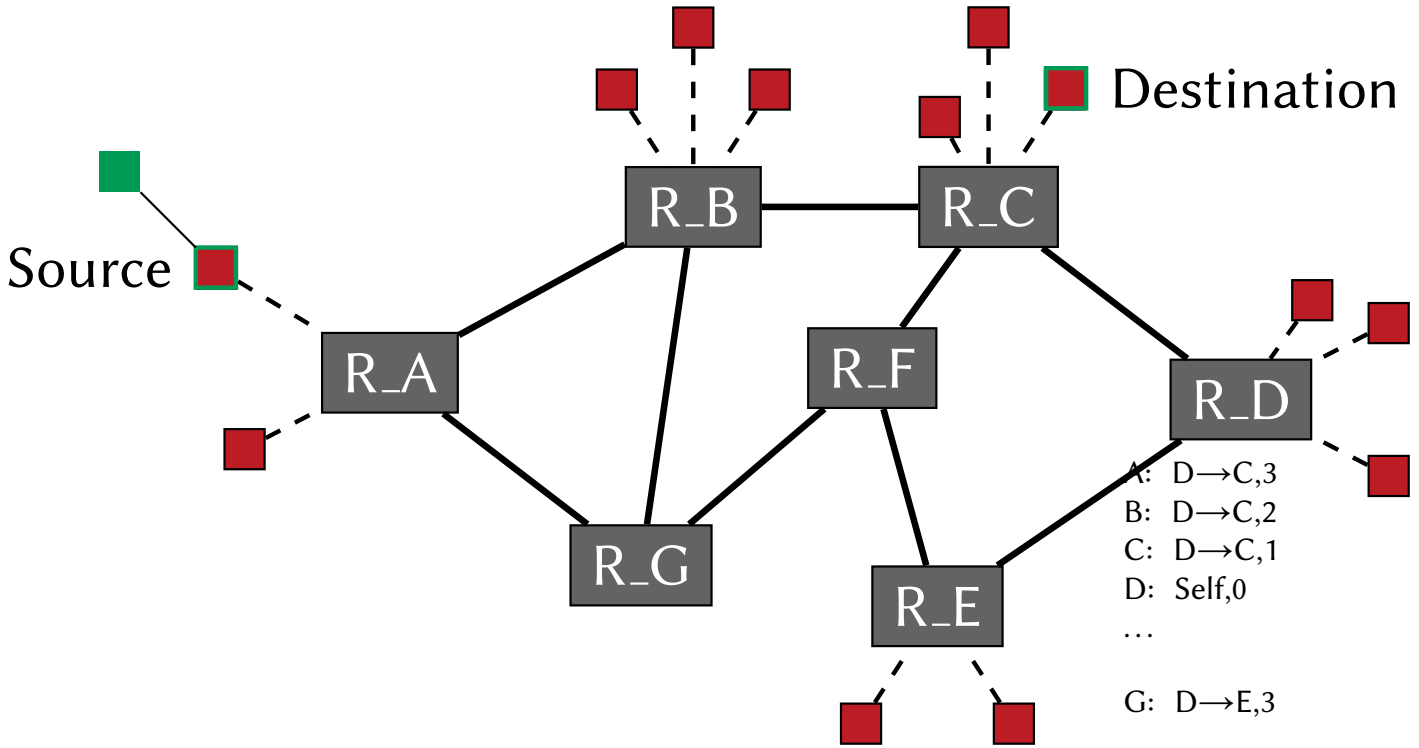
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



The Network Layer (IP)

Protocols to:

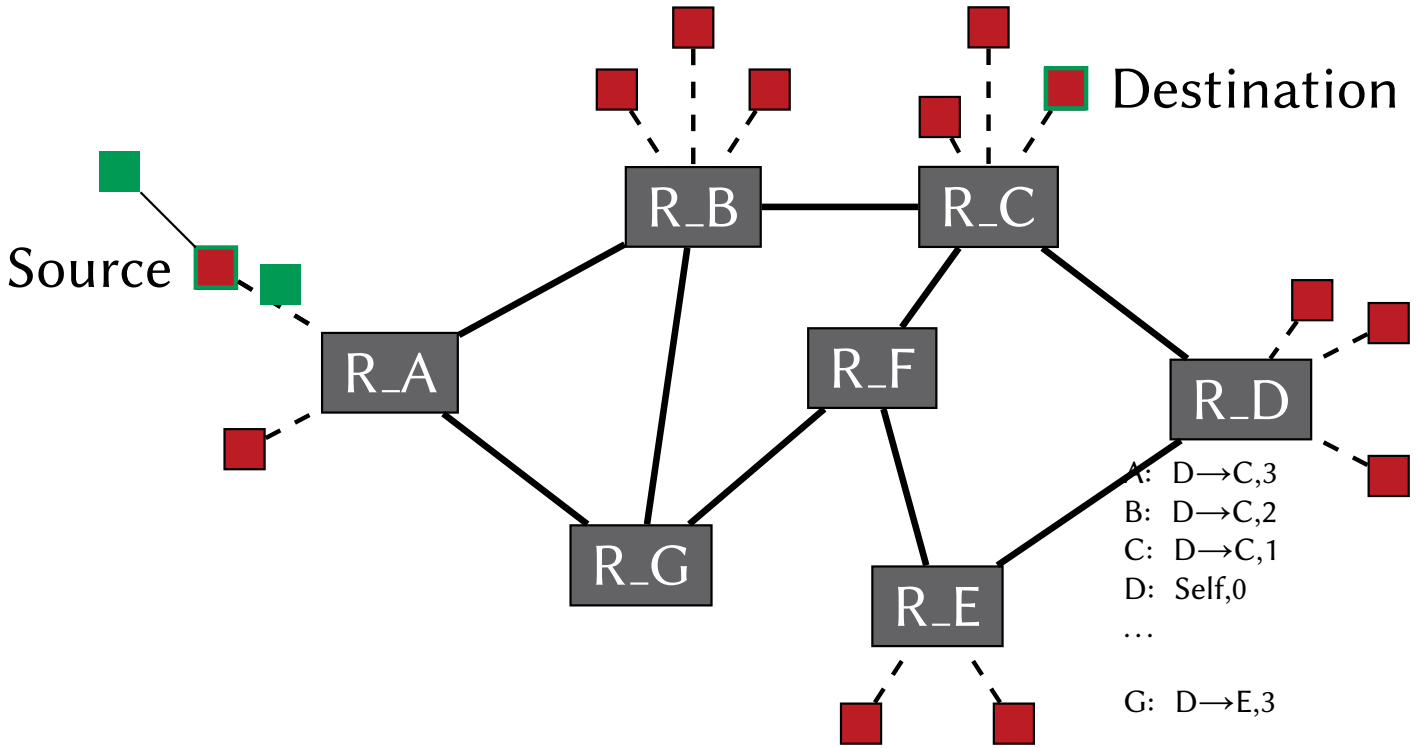
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



The Network Layer (IP)

Protocols to:

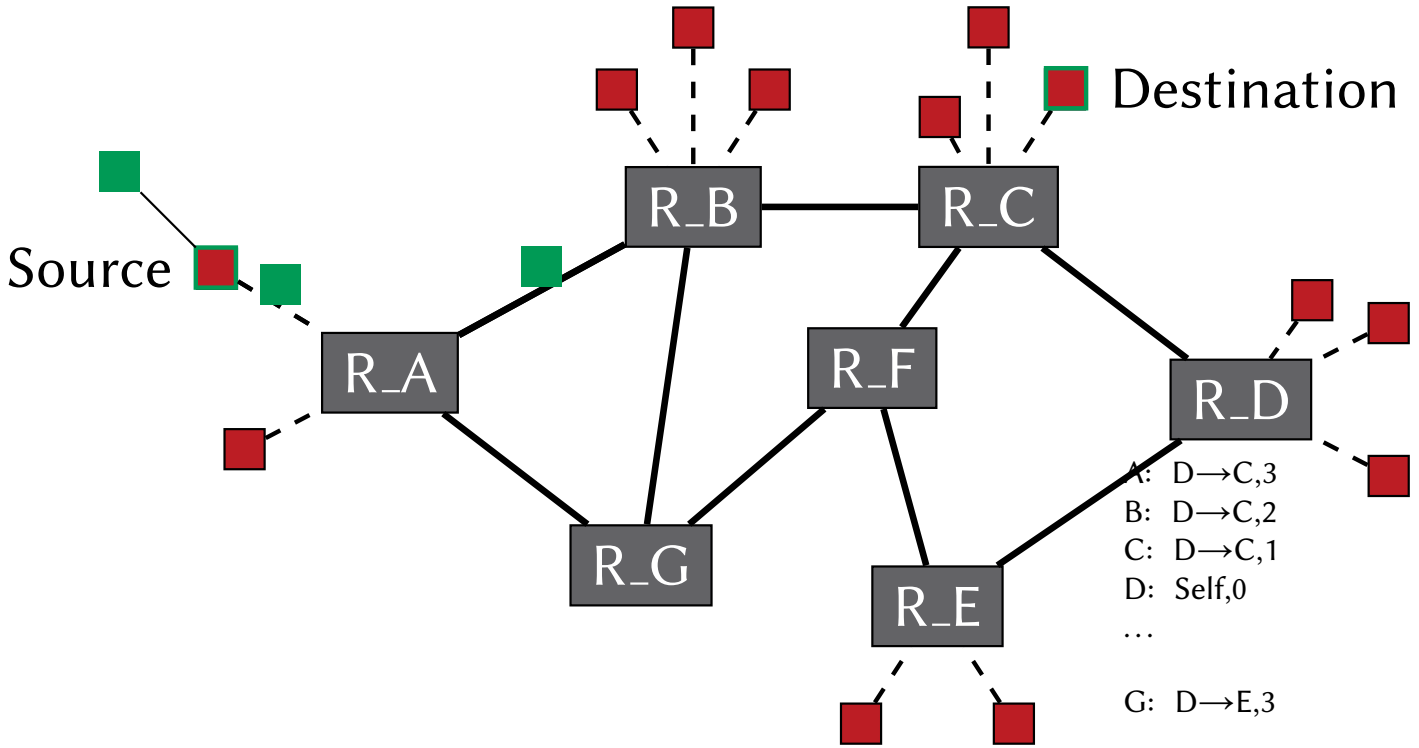
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



The Network Layer (IP)

Protocols to:

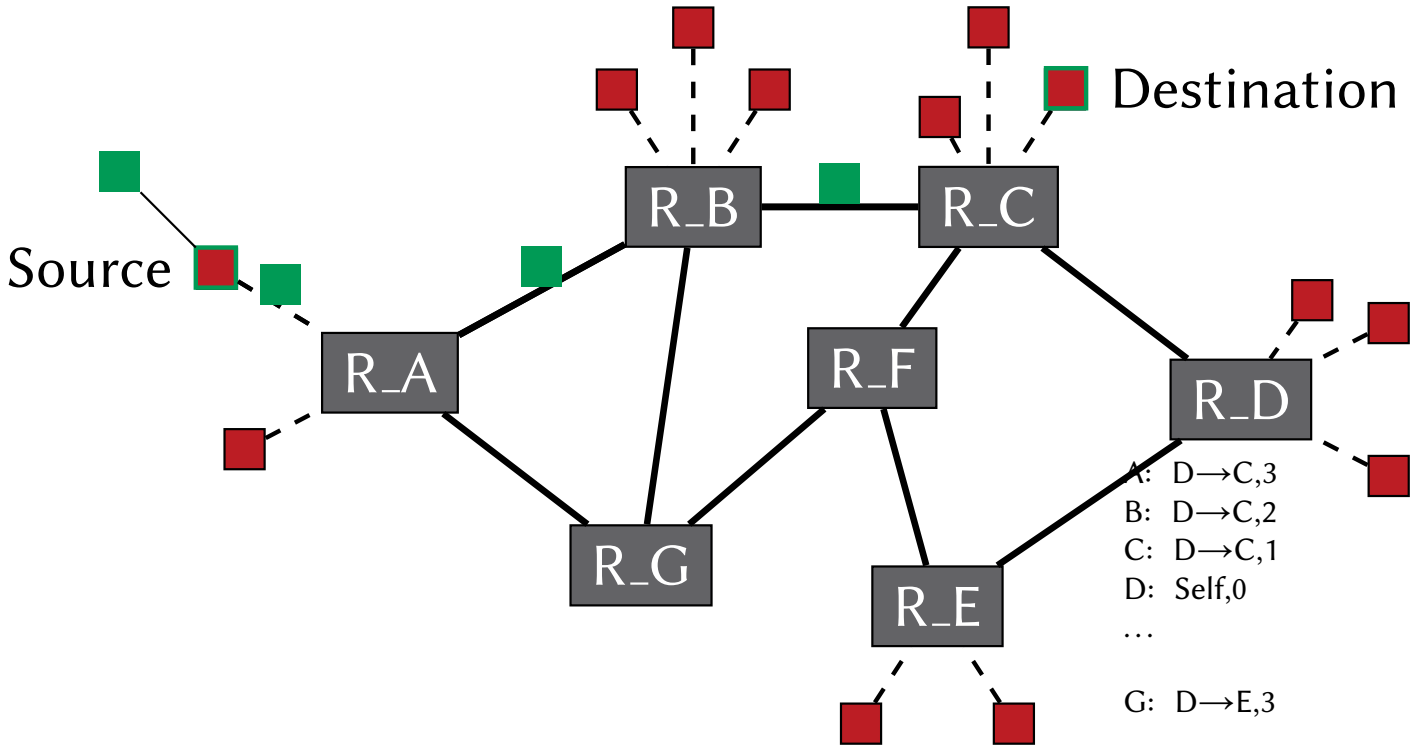
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



The Network Layer (IP)

Protocols to:

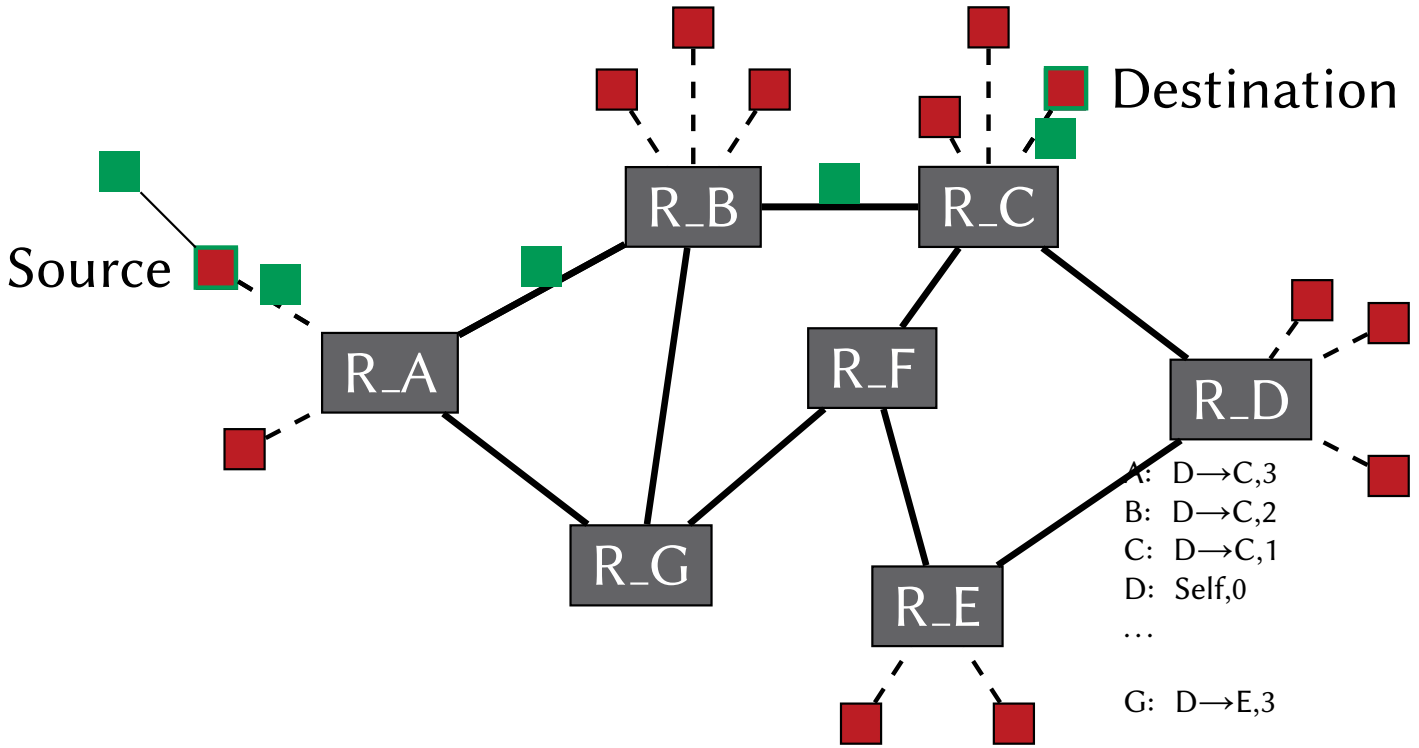
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



The Network Layer (IP)

Protocols to:

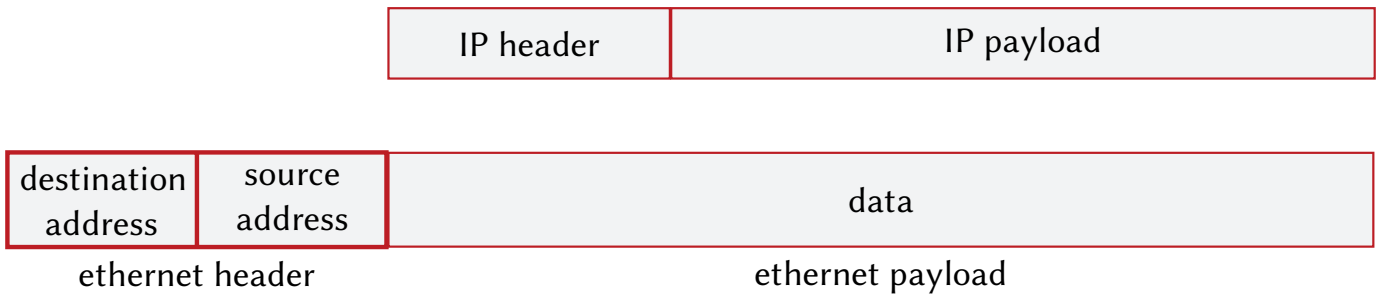
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination

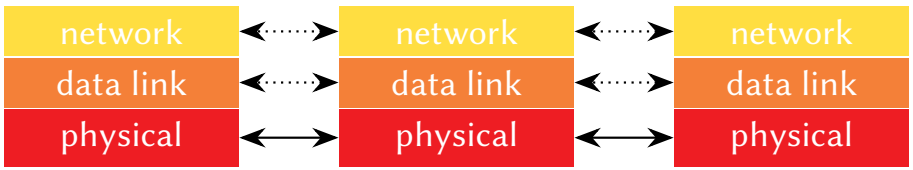


The Network Layer

Packet encapsulation

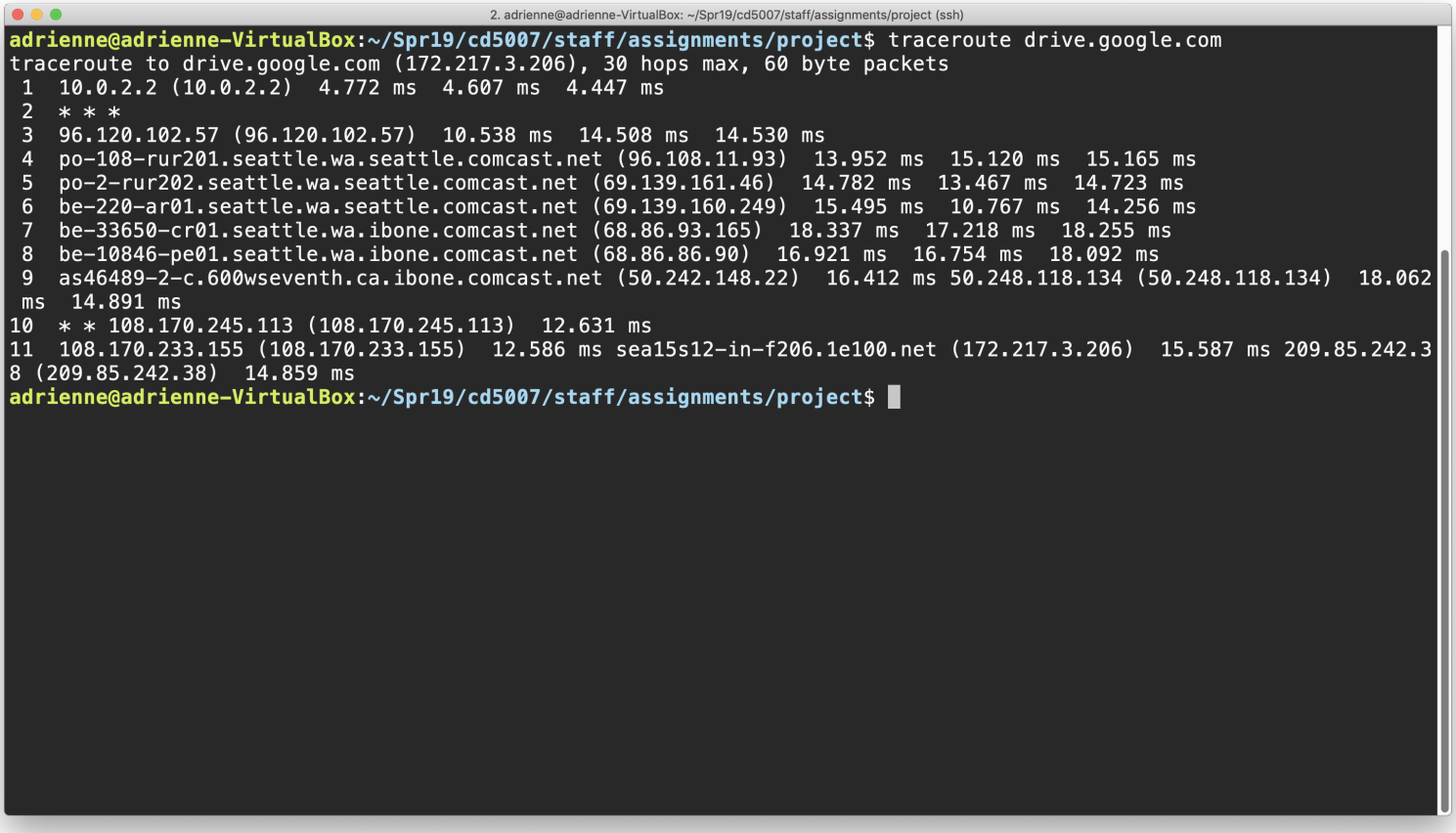
- an IP packet is encapsulated as the payload of an Ethernet frame
- as IP packets traverse networks, routers pull out the IP packet from an ethernet frame and plunk it into a new one on the next network





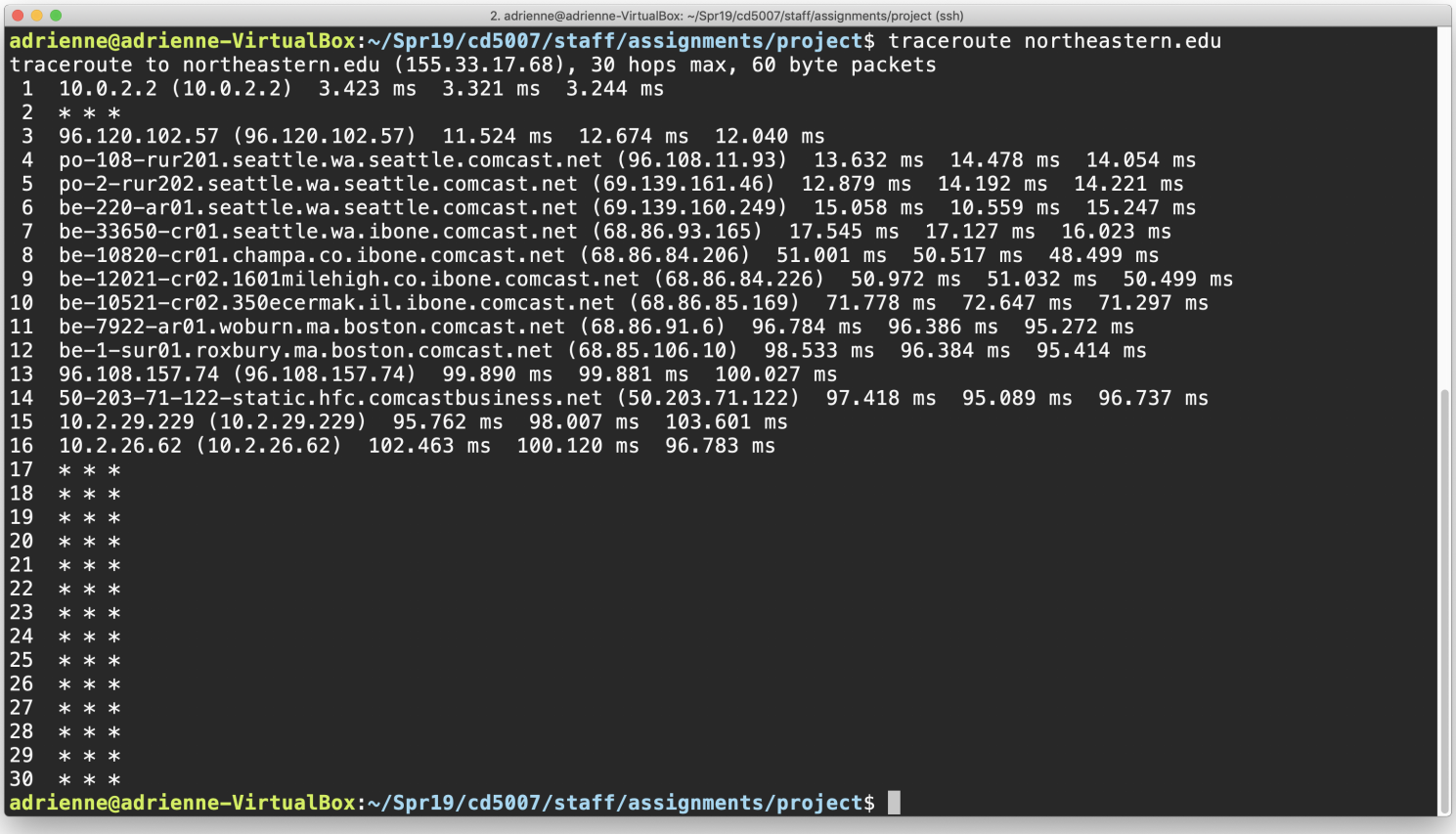
A fun thing to do

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



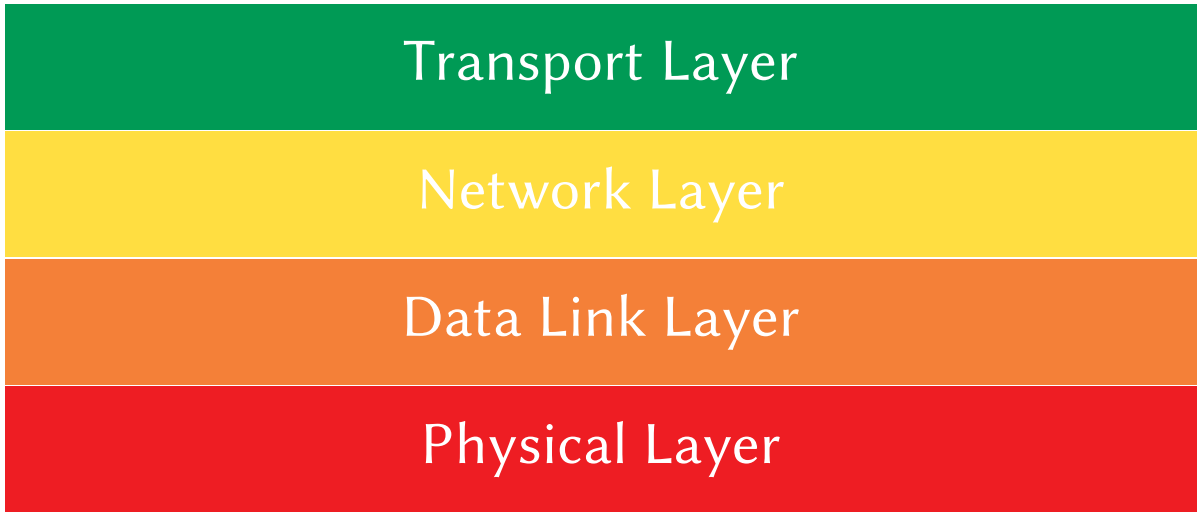
A fun thing to do

`traceroute`[2] let's you see the path a request takes to get to a destination:



[2] https://community.spiceworks.com/networking/articles/2531-traceroute-request-timed-out-why-traceroute-is-broken

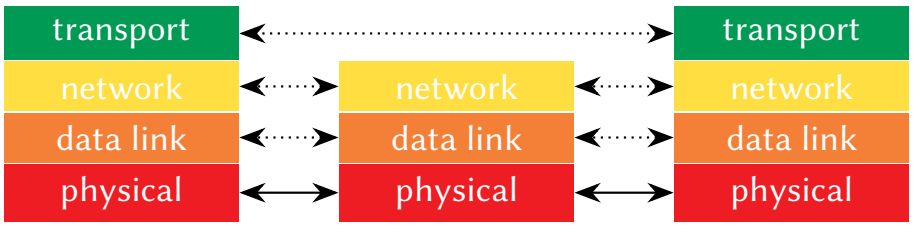
OSI Model


Transport Layer
Network Layer
Data Link Layer
Physical Layer

Transport Layer

TCP

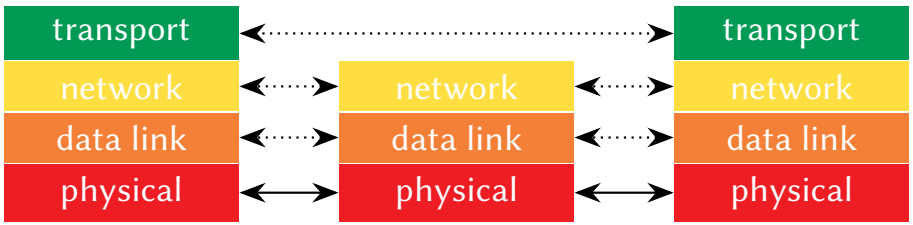
- the “transmission control protocol”
- provides apps with reliable, ordered, congestion-controlled byte streams
- fabricates them by sending multiple IP packets, using sequence numbers to detect missing packets, and retransmitting them
- a single host (IP address) can have up to 65,535 “ports”
- kind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



Transport Layer

TCP

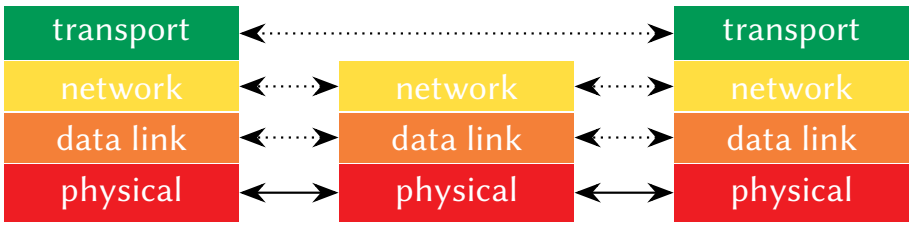
- useful analogy: how would you send a book by mail via postcards?



Transport Layer

TCP

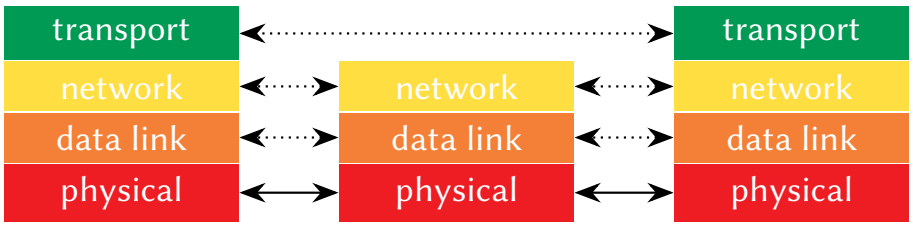
- useful analogy: how would you send a book by mail via postcards?
- split the book into multiple postcards, send each one by one, including sequence numbers that indicate the assembly order



Transport Layer

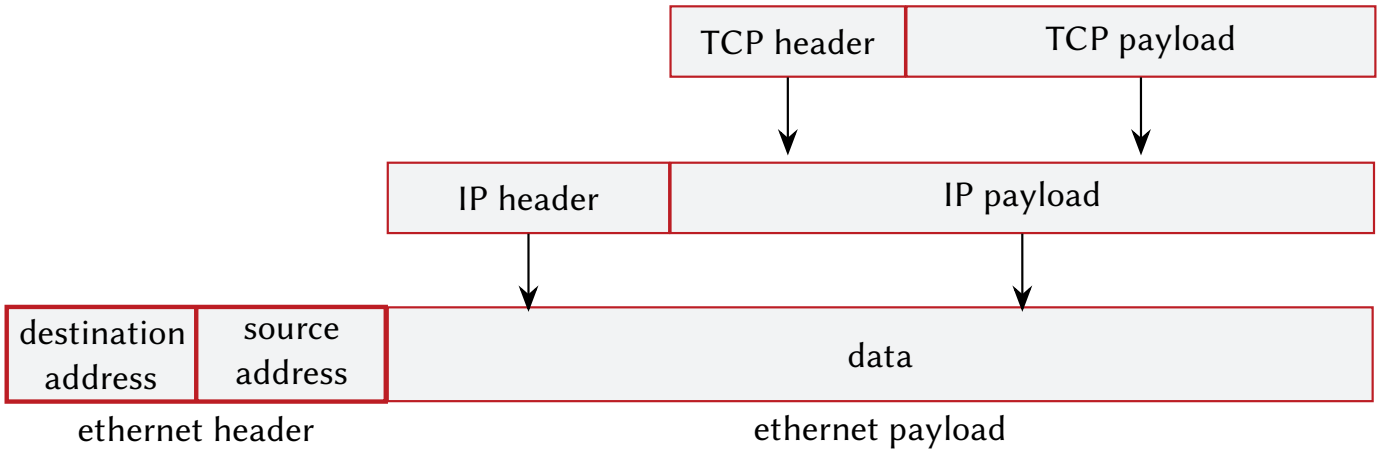
TCP

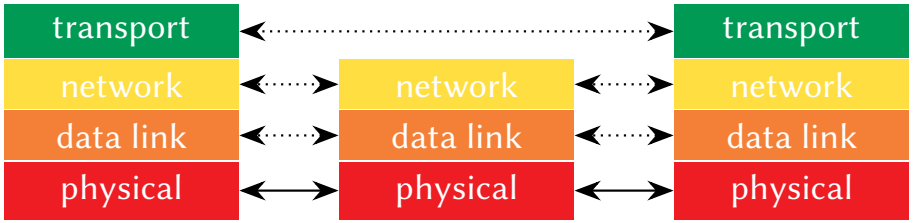
- useful analogy: how would you send a book by mail via postcards?
- split the book into multiple postcards, send each one by one, including sequence numbers that indicate the assembly order
- receiver sends back postcards to acknowledge receipt and indicate which got lost in the mail



The Transport Layer (TCP)

Packet encapsulation!



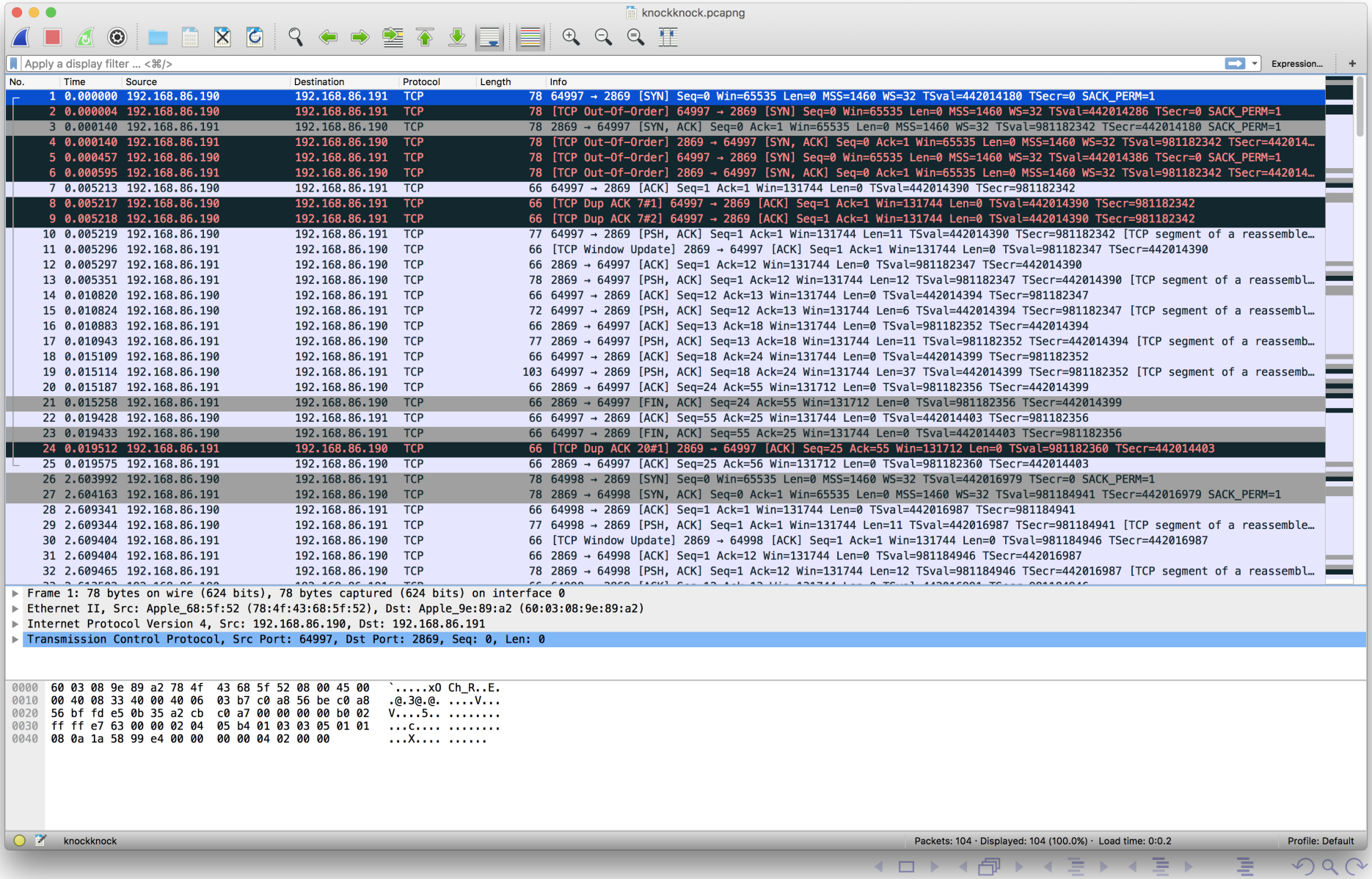


Transport Layer (TCP)

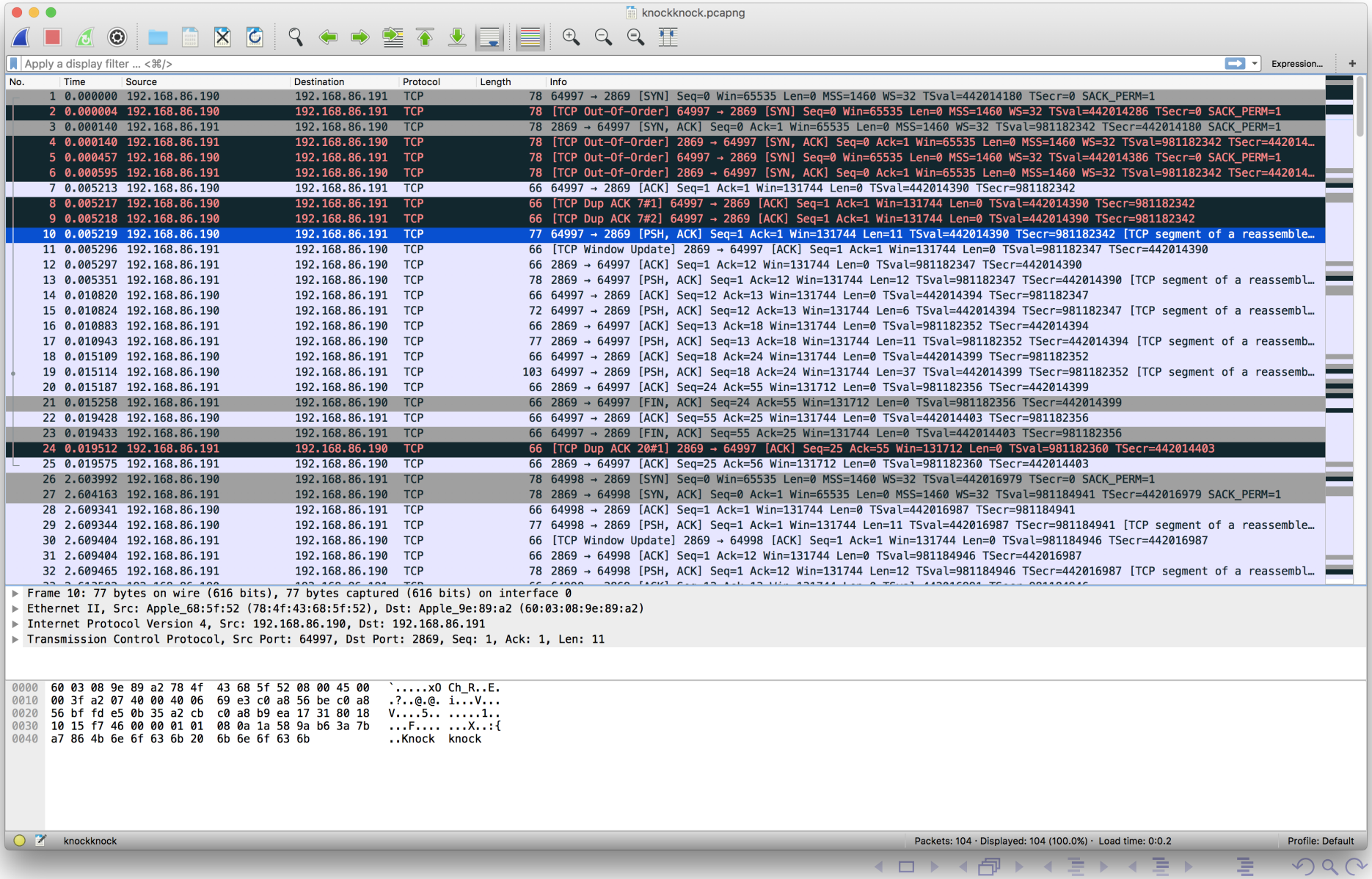
Applications use OS services to establish TCP streams

- 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



Sniffing TCP traffic



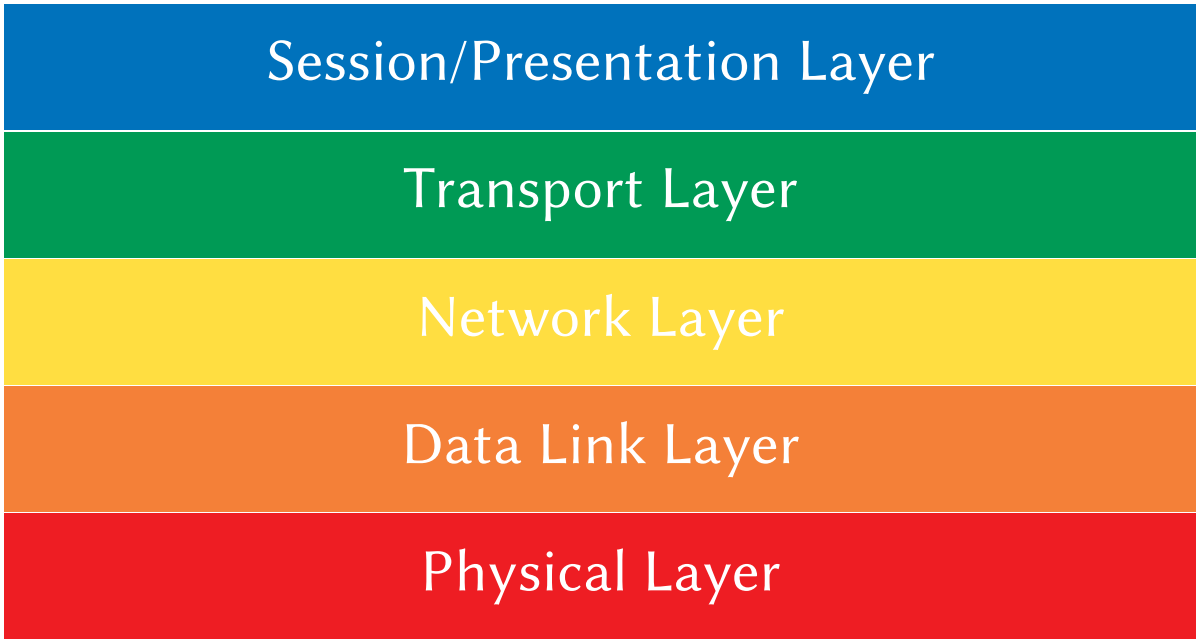
Transport Layer (UDP)

UDP

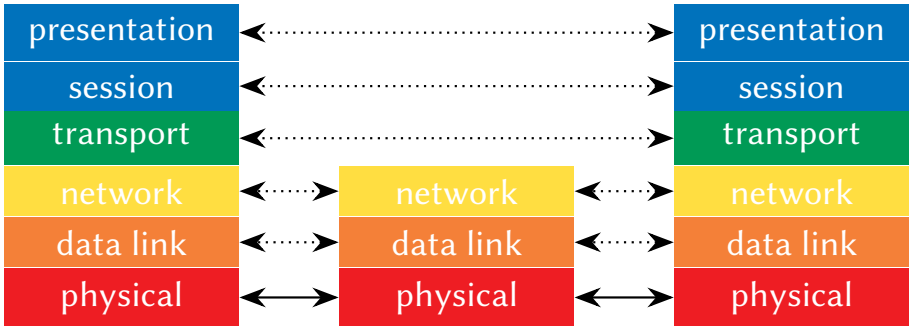
- the “user datagram protocol”
- UDP is faster because error recovery is not attempted. It is a “best effort” protocol.
- There is no guarantee that the messages or packets sent would reach at all.
- UDP datagrams are fragmented into multiple IP packets
  - UDP is a really thin, simple layer on top of 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 Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer

Session, Presentation Layers



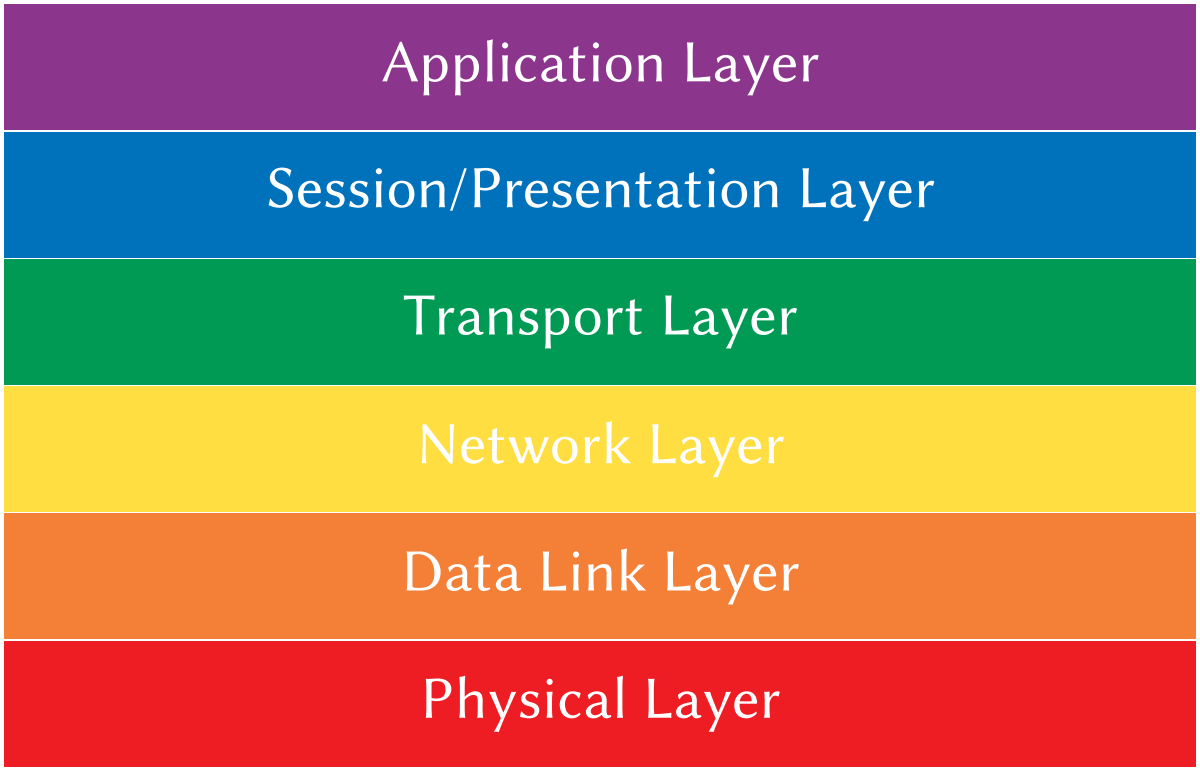
Layer 5: session layer

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

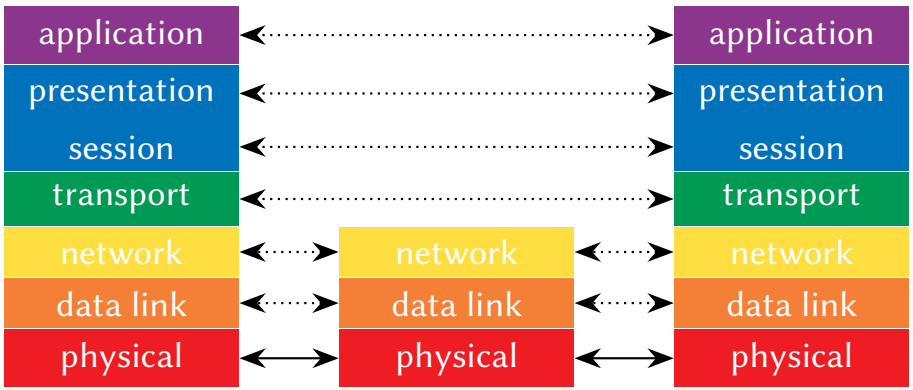
Layer 6: presentation layer

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

OSI Model


Application Layer
Session/Presentation Layer
Transport Layer
Network Layer
Data Link Layer
Physical Layer

Application Layer

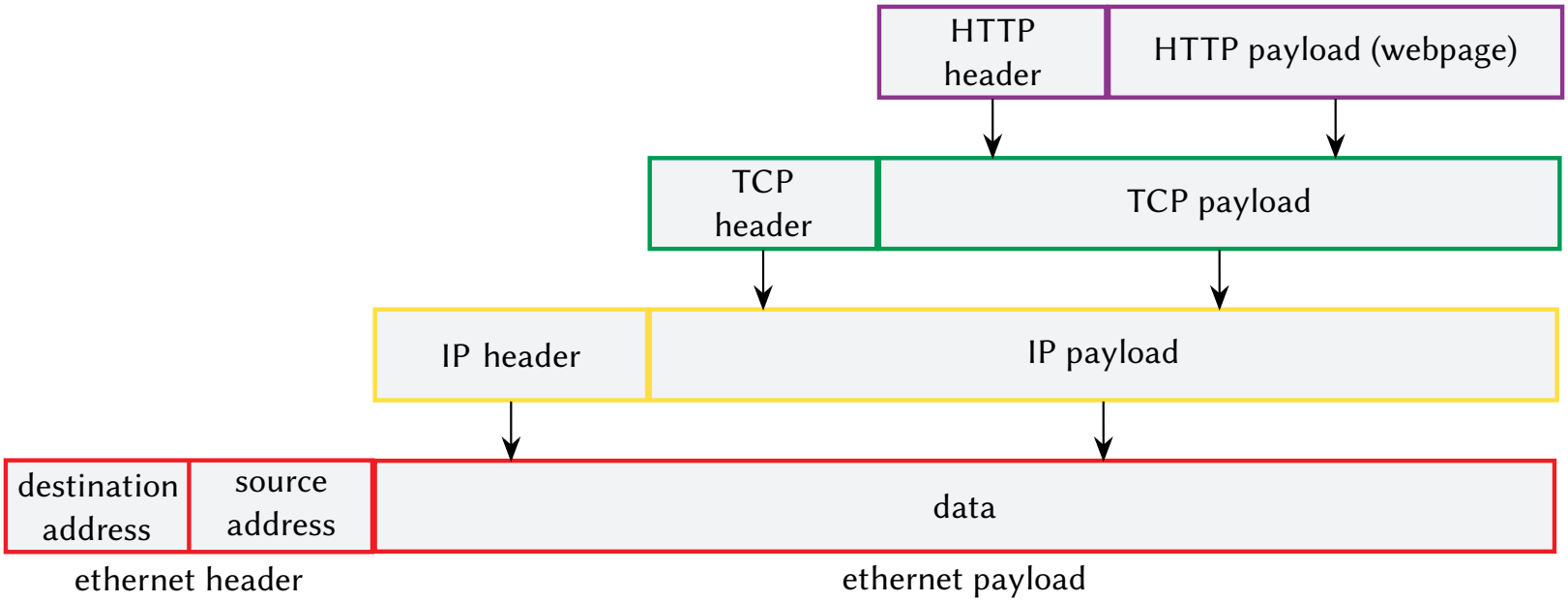


Application protocols

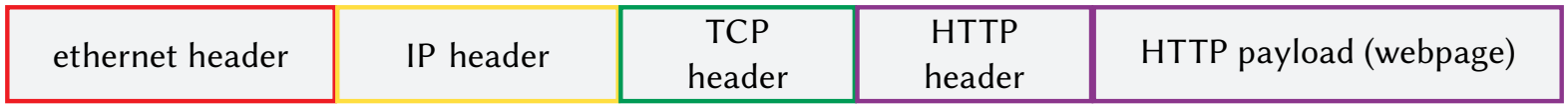
- the format and meaning of messages between application entities
- e.g., HTTP is an application level protocol that dictates how web browsers and web servers communicate
  - HTTP is implemented on top of TCP streams

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:

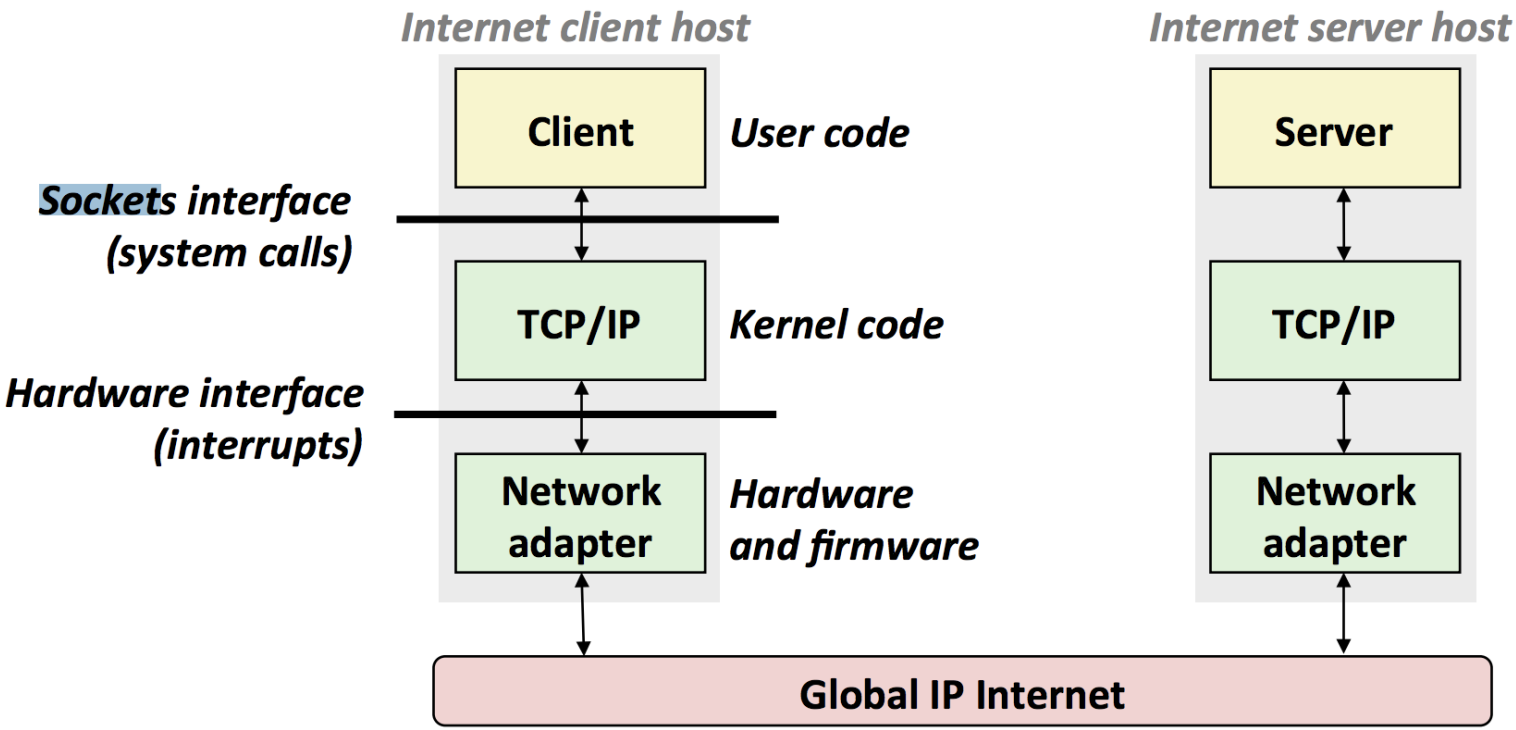
- DNS: translates a DNS name (www.google.com) into one or more IP addresses (74.125.155.105, 74.125.155.106, ...)
  - a hierarchy of DNS servers cooperate to do this
- HTTP: web protocols
- SMTP, IMAP, POP: mail delivery and access protocols
- ssh: remote login protocol

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"

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

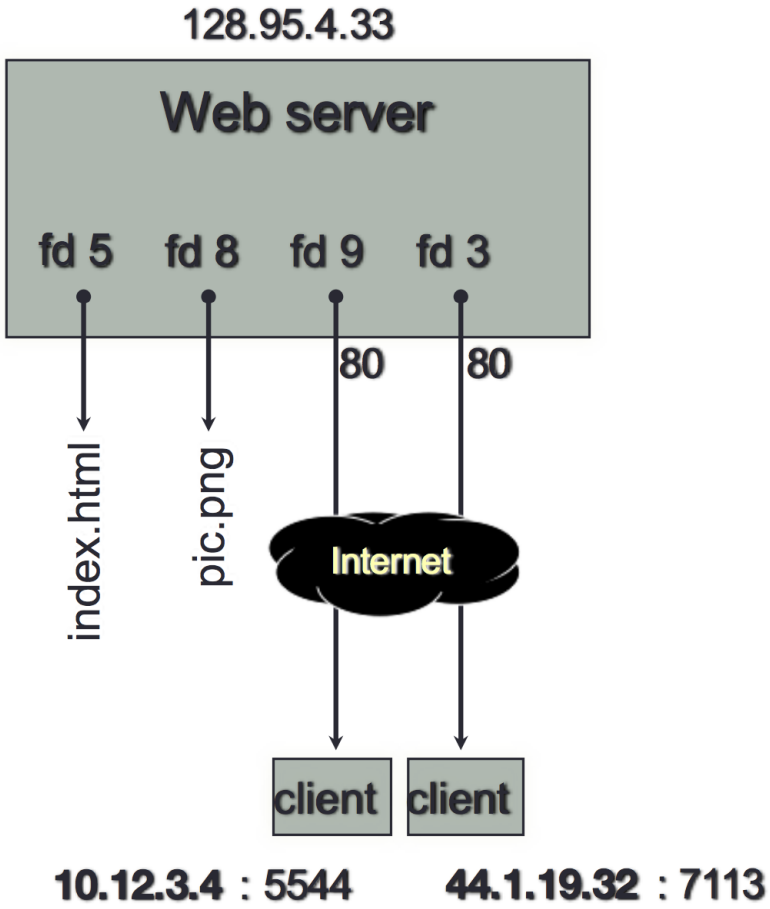
Remember open, read, write, and close?

- 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 bytestreams
  - 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

Client

- an application that establishes a connection to a server

Server

- an application that receives connections from clients

1. Establish connection
2. Communicate
3. Close connection

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?

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

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[3]

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

struct addrinfo hints, *infoptr;

int main() {
  hints.ai_family = AF_INET; // AF_INET means IPv4 only addresses

  int result = getaddrinfo("www.northeastern.edu", NULL, &hints, &infoptr);
  if (result) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(result));
    exit(1);
  }

  struct addrinfo *p;
  char host[256];

  for(p = infoptr; p != NULL; p = p->ai_next) {

    getnameinfo(p->ai_addr, p->ai_addrlen, host, sizeof(host), NULL, 0, NI_NUMERICHOST);
    puts(host);
  }

  freeaddrinfo(infoptr);
  return 0;
}
```

[3] 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[4]

```c
int main(int argc, char **argv)
{
  int s;
  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

  struct addrinfo hints, *result;
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET; /* IPv4 only */
  hints.ai_socktype = SOCK_STREAM; /* TCP */

  s = getaddrinfo("www.illinois.edu", "80", &hints, &result);
  if (s != 0) {
          fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
          exit(1);
  }

  if(connect(sock_fd, result->ai_addr, result->ai_addrlen) == -1){
                perror("connect");
                exit(2);
        }

        // For this trivial demo just assume write() sends all bytes in one go and is not
     interrupted
  write(sock_fd, "hi!", 3);
  char resp[1000];
  int len = read(sock_fd, resp, 999);
  resp[len] = '\0';
  printf("%s\n", resp);

    return 0;
}
```

[4] 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("===\n");
21         // For this trivial demo just assume write() sends all bytes in one go and is not
       interrupted
22   write(sock_fd, buffer, strlen(buffer));
23         // For this trivial demo just assume write() sends all bytes in one go and is not
       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;
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:

1. 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     printf("Waiting for connection...\n");
14     int client_fd = accept(sock_fd, NULL, NULL);
15     printf("Connection made: client_fd=%d\n", client_fd);
16
17     char buffer[1000];
18     int len = read(client_fd, buffer, sizeof(buffer) - 1);
19     buffer[len] = '\0';
20
21     printf("Read %d chars\n", len);
22     printf("===\n");
23     printf("%s\n", buffer);
24
25     return 0;
26 }
```

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...

```c
//Includes omitted for space
int main(int argc, char **argv)
{
  int s;
  // Step 1: Get address stuff
  struct addrinfo hints, *result;
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  s = getaddrinfo(NULL, "1234", &hints, &result);
  if (s != 0) {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
      exit(1);
  }
  // Step 2: Open socket
  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);
  // Step 3: Bind socket
  if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
    perror("bind()");
    exit(1);
  }
  // Step 4: Listen on the socket
  if (listen(sock_fd, 10) != 0) {
    perror("listen()");
    exit(1);
  }
  struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
  printf("Listening on file descriptor %d, port %d\n", sock_fd, ntohs(result_addr->sin_port));
  // Step 5: Accept connection
  printf("Waiting for connection...\n");
  int client_fd = accept(sock_fd, NULL, NULL);
  printf("Connection made: client_fd=%d\n", client_fd);
  // Step 6: Read, then write if you want
  char buffer[1000];
  int len = read(client_fd, buffer, sizeof(buffer) - 1);
  buffer[len] = '\0';
  printf("Read %d chars\n", len);
  printf("===\n");
  printf("%s\n", buffer);
  // Step 7: Close
  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()`
  - `recieve_message()`
  - `send_message()`

```c
int do_open(){
  int s;
  // Step 1: Get Address stuff
  struct addrinfo hints, *result;

  // Setting up the hints struct...
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET;
  hints.ai_socktype = SOCK_STREAM;
  hints.ai_flags = AI_PASSIVE;
  s = getaddrinfo(NULL, "1414", &hints, &result);
  if (s != 0) {
    fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
    exit(1);
  }

  // Step 2: Create the socket
  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

  // Step 3: Bind the socket
  if (bind(sock_fd, result->ai_addr, result->ai_addrlen) != 0) {
    perror("bind()");
    exit(1);
  }

  // Step 4: Listen
  if (listen(sock_fd, 10) != 0) {
    perror("listen()");
    exit(1);
  }

  struct sockaddr_in *result_addr = (struct sockaddr_in *) result->ai_addr;
  printf("Listening on file descriptor %d, port %d\n", sock_fd, ntohs(result_addr->sin_port));

  // Step 5: Accept a connection
  printf("Waiting for connection...\n");
  int client_fd = accept(sock_fd, NULL, NULL);
  printf("Connection made: client_fd=%d\n", client_fd);

  return client_fd;
}
```

```c
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>
#include <arpa/inet.h>

int do_open(){
  // See def on previous slide...
}

void recieve_message(int client_fd){
  char buffer[1000];
  int len = read(client_fd, buffer, sizeof(buffer) - 1);
  buffer[len] = '\0';
  printf("SERVER RECEIVED: %s \n", buffer);
}

void send_message(char *msg, int sock_fd){
  printf("SERVER SENDING: %s", msg);
  printf("===\n");
  write(sock_fd, msg, strlen(msg));
}

int main(int argc, char **argv){

  int client_socket_id = do_open();

  // Step 6: Read and write
  recieve_message(client_socket_id);

  send_message("Who's there?", client_socket_id);

  // Step 7: Close
  close(client_socket_id);

  return 0;
}
```

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()`

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

int do_connect(char *host, char *port){
  int s;
  int sock_fd = socket(AF_INET, SOCK_STREAM, 0);

  struct addrinfo hints, *result;

  // Allows "global"
  memset(&hints, 0, sizeof(struct addrinfo));
  hints.ai_family = AF_INET; /* IPv4 only */
  hints.ai_socktype = SOCK_STREAM; /* TCP */

  s = getaddrinfo(host, port, &hints, &result);

  // If I can't get the address, write an error.
  if (s != 0) {
      fprintf(stderr, "getaddrinfo: %s\n", gai_strerror(s));
      exit(1);
  }

  // Try to connect; if I can't, write an error.
  if(connect(sock_fd, result->ai_addr, result->ai_addrlen) == -1){
    perror("connect");
    exit(2);
  }
  else{
    printf("Connection is good!\n");
  }

  return sock_fd;
}
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>
#include <unistd.h>

int do_connect(char *host, char *port){
  // See implementation on the previous slide
}

void send_message(char *msg, int sock_fd){
  printf("SENDING: %s", msg);
  printf("===\n");
  write(sock_fd, msg, strlen(msg));
}

void read_response(int sock_fd){
  // Response
  char resp[1000];
  int len = read(sock_fd, resp, 999);
  resp[len] = '\0';

  printf("RECEIVED: %s\n", resp);
}

int main(int argc, char **argv){

  int sock_fd = do_connect("localhost", "1414");

  send_message("Knock knock", sock_fd);

  read_response(sock_fd);

  close(sock_fd);

  return 0;
}
```

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 commuication
- 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