Poll Everywhere

pollev.com/cse333j

About how long did Exercise 13 take you?

A. [0, 2) hours
B. [2, 4) hours
C. [4, 6) hours
D. [6, 8) hours
E. 8+ Hours
F. I didn't submit / I prefer not to say

Systems Programming
Networks Introduction

Instructors:

Justin Hsia Amber Hu

Teaching Assistants:

Ally Tribble	Blake Diaz	Connor Olson
Grace Zhou	Jackson Kent	Janani Raghavan
Jen Xu	Jessie Sun	Jonathan Nister
Mendel Carroll	Rose Maresh	Violet Monserate

Relevant Course Information

- Bit of a break from exercises – Exercises 15 & 16 released on Friday

- Homework 3 is due Thursday (2/23)
  - Debug using small custom test directories
  - Make use of the solution binaries to double-check your work

- Rest of the quarter: networking, concurrency, processes

Lecture Outline (1/4)

- **C++ Inheritance (finish)**
  - **Constructors and Destructors**
  - **Assignment**
- C++ Casting
- C++ Conversions
- Networks Introduction

- Reference: *C++ Primer* §4.11.3, 19.2.1

Assignment and Inheritance

- C++ allows you to assign the value of a derived class to an instance of a base class
  - Known as object slicing
    - It's legal since b = d passes type checking rules
    - But b doesn't have space for any extra fields in d

slicing.cc

```
class Base {
 public:
  Base(int xi) : x(xi) { }
  int x;
};

class Der1 : public Base {
 public:
  Der1(int yi) : Base(16), y(yi) { }
  int y;
};

void Foo() {
  Base b(1);
  Der1 d(2);

  d = b;  //
  b = d;  //
}
```

STL and Inheritance (1/2)

- Recall: STL containers store **copies of values**
  - What happens when we want to store mixes of object types in a single container? (*e.g.*, `Stock` and `DividendStock`)
  - You get sliced ☹

```
#include <list>
#include "Stock.h"
#include "DividendStock.h"

int main(int argc, char** argv) {
  Stock s;
  DividendStock ds;
  list<Stock> li;

  li.push_back(s);    // OK
  li.push_back(ds);   // OUCH!

  return EXIT_SUCCESS;
}
```

STL and Inheritance (2/2)

- Instead, store **pointers to heap-allocated objects** in STL containers
  - No slicing! ☺
  - `sort()` does the wrong thing ☹
  - You have to remember to `delete` your objects before destroying the container ☹
    - Unless you use smart pointers!

Lecture Outline (2/4)

- C++ Inheritance (finish)
  - Constructors and Destructors
  - Assignment
- **C++ Casting**
- C++ Conversions
- Networks Introduction

- Reference: *C++ Primer* §4.11.3, 19.2.1

Explicit Casting in C

- Simple syntax: `lhs = (new_type) rhs;`
- Used to:
  - Convert between pointers of arbitrary type
    - Doesn't change the data, but treats it differently
  - Forcibly convert a primitive type to another
    - Actually changes the representation

- You *can* still use C-style casting in C++, but sometimes the intent is not clear
  - You *should not* use C-style casting in C++.

Casting in C++

STYLE TIP

- C++ provides an alternative casting style that is more informative:
  - `static_cast<to_type>(expression)`
  - `dynamic_cast<to_type>(expression)`
  - `const_cast<to_type>(expression)`
  - `reinterpret_cast<to_type>(expression)`

- Always use these in C++ code
  - Intent is clearer
  - Easier to find in code via searching

static_cast

staticcast.cc

```
class A {
 public:
  int x;
};

class B {
 public:
  float x;
};

class C : public B {
 public:
  char x;
};
```

- `static_cast` can convert:
  - Pointers to classes **of related type**
    - Compiler error if classes are not related
    - Dangerous to cast *down* a class hierarchy
  - Casting between `void*` and `T*`
  - Non-pointer conversion
    - *e.g.*, `float` to `int`
- `static_cast` is checked at <u>compile time</u>

```
void Foo() {
  B b; C c;

  // compiler error
  A* aptr = static_cast<A*>(&b);
  // OK
  B* bptr = static_cast<B*>(&c);
  // compiles, but dangerous
  C* cptr = static_cast<C*>(&b);
}
```

dynamiccast.cc

dynamic_cast

- `dynamic_cast` can convert:
  - Pointers to classes **of related type**
  - References to classes **of related type**
- `dynamic_cast` is checked at both compile time and run time
  - Casts between unrelated classes fail at compile time
  - Casts from base to derived fail at run time if the pointed-to object is not the derived type

```
class Base {
 public:
  virtual void Foo() { }
  float x;
};

class Der1 : public Base {
 public:
  char x;
};
```

```
void Bar() {
  Base b; Der1 d;

  // OK (run-time check passes)
  Base* bptr = dynamic_cast<Base*>(&d);
  assert(bptr != nullptr);

  // OK (run-time check passes)
  Der1* dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);

  // Run-time check fails, returns nullptr
  bptr = &b;
  dptr = dynamic_cast<Der1*>(bptr);
  assert(dptr != nullptr);
}
```

const_cast

- `const_cast` adds or strips const-ness
  - Dangerous (!)

```
void Foo(int* x) {
  *x++;
}

void Bar(const int* x) {
  Foo(x);                      // compiler error
  Foo(const_cast<int*>(x));   // succeeds
}

int main(int argc, char** argv) {
  int x = 7;
  Bar(&x);
  return EXIT_SUCCESS;
}
```

`reinterpret_cast`

- `reinterpret_cast` casts between *incompatible* types
  - Low-level reinterpretation of the bit pattern
  - *e.g.*, storing a pointer in an `int`, or vice-versa
    - Works as long as the integral type is "wide" enough
  - Converting between incompatible pointers
    - Dangerous (**!**)
    - This is used (carefully) in hw3
  - Use any other C++ cast if you can!

Casting Style Considerations

STYLE TIP

- From the "Casting" and "Run-Time Type Information (RTTI)" sections of the Google C++ Style Guide:
  - When the logic of a program guarantees that a given instance of a base class is, in fact, an instance of a particular derived class, then a `dynamic_cast` may be used freely on the object.
    - Usually one can use a `static_cast` as an alternative in such situations
  - Only use `reinterpret_cast` if you know what you are doing and you understand the aliasing issues
    - For *unsafe conversions* of pointer types to and from integer and other pointer types, including `void*`

Lecture Outline (3/4)

- C++ Inheritance (finish)
  - Constructors and Destructors
  - Assignment
- C++ Casting
- **C++ Conversions**
- Networks Introduction

- Reference: *C++ Primer* §4.11.3, 19.2.1

Implicit Conversion

- The compiler tries to infer some kinds of conversions
  - When types are not equal and you don't specify an explicit cast, the compiler looks for an acceptable implicit conversion

```
void Bar(std::string x);

void Foo() {
  int x = 5.7;  // conversion, float -> int
  char c = x;   // conversion, int -> char
  Bar("hi");    // conversion, (const char*) -> string
}
```

Sneaky Implicit Conversions

- (`const char*`) to `string` conversion?
  - If a class has a constructor with a single parameter, the compiler will exploit it to perform implicit conversions
  - At most, one user-defined implicit conversion will happen
    - Can do `int` → `Foo`, but not `int` → `Foo` → `Baz`

```
class Foo {
 public:
  Foo(int xi) : x(xi) { }
  int x;
};

int Bar(Foo f) {
  return f.x;
}

int main(int argc, char** argv) {
  return Bar(5);  // equivalent to return Bar(Foo(5));
}
```

Avoiding Sneaky Implicits

STYLE TIP

- Declare one-argument constructors as `explicit` if you want to disable them from being used as an implicit conversion path
  - Usually a good idea

```
class Foo {
 public:
  explicit Foo(int xi) : x(xi) { }
  int x;
};

int Bar(Foo f) {
  return f.x;
}

int main(int argc, char** argv) {
  return Bar(5);  // compiler error
}
```

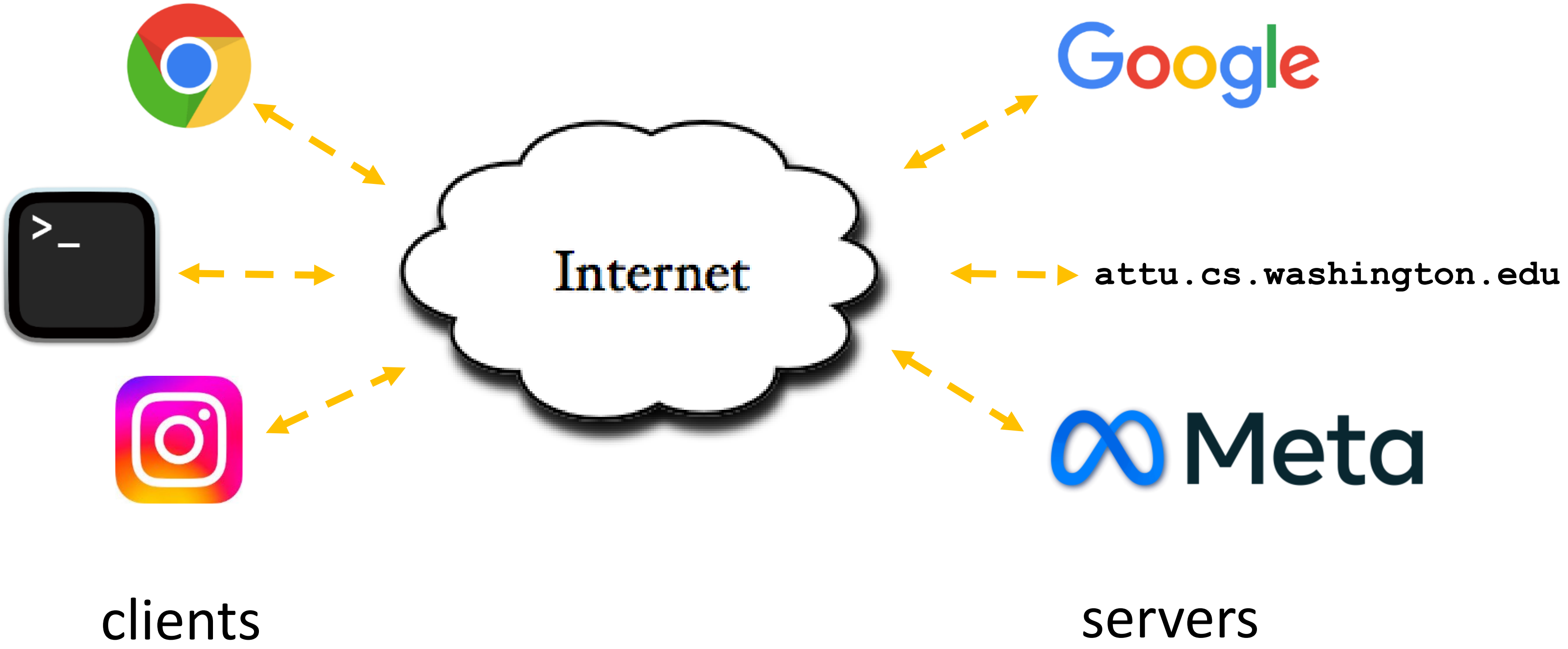
Lecture Outline (4/4)

- C++ Inheritance (finish)
  - Constructors and Destructors
  - Assignment
- C++ Casting
- C++ Conversions
- **Networks Introduction**

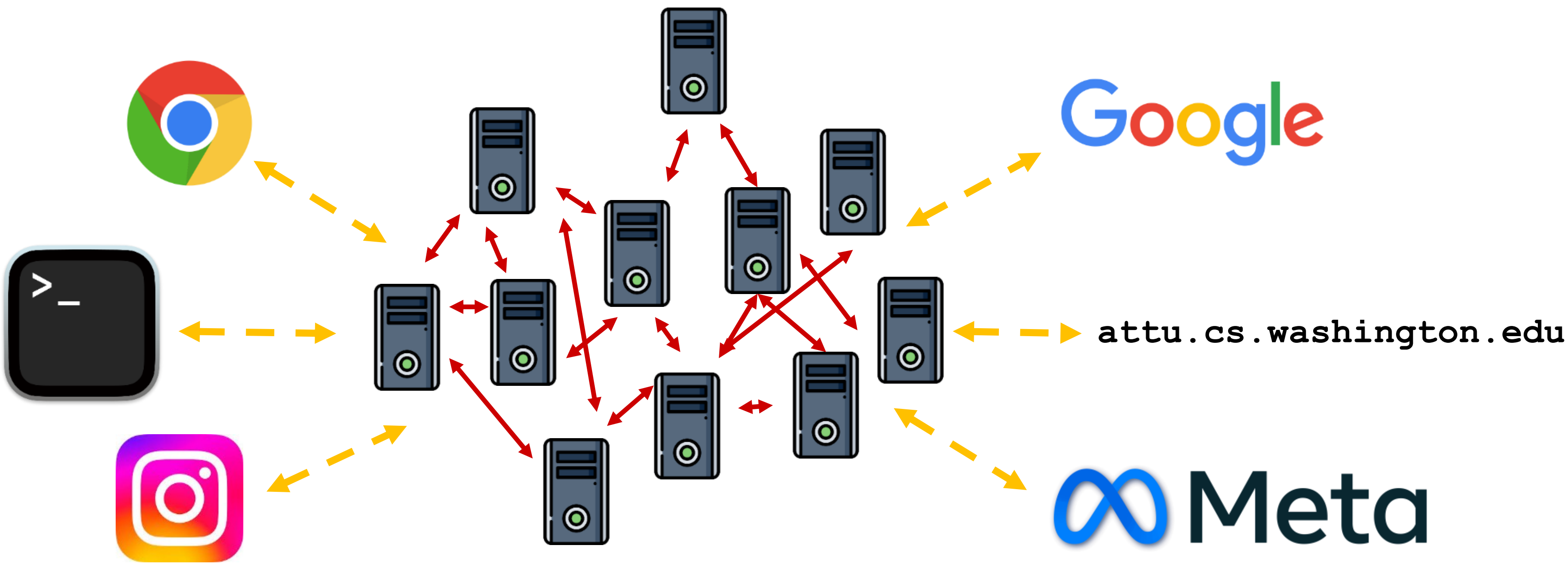
Today's Goals

- Networking is a very common programming feature
  - You will likely have to create a program that will read/write over the network at some point in your career
- We want to give you a basic, high-level understanding of how networks work before you use them
  - Lecture will be more "story-like;" we will purposefully skip over most of the details, but hopefully you will learn something new about the Internet today!
  - Take CSE 461 if you want to know more about the implementations of networks (the course is pretty cool ☺)

Networks From 10,000 ft

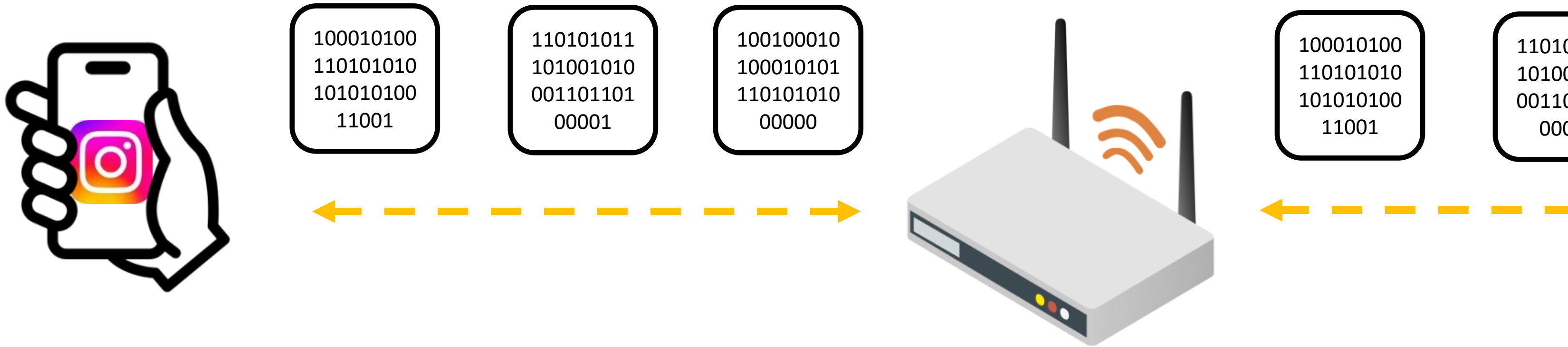


Networks From 1,000 ft



clients network backbone servers

Networks From 10 ft



We send data from one software system to another by packaging it into **data packets** and then asking the **operating system** to transport them for us.

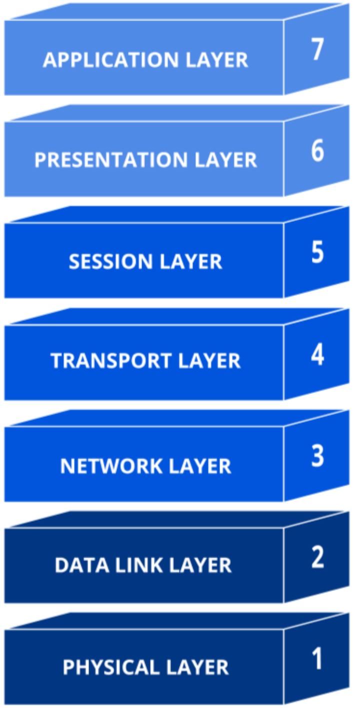
Packet Contents

- Layers…
  - Upon layers…
    - Upon layers…
      - upon layers…
        - upon layers…
          - upon layers…
            - …upon layers!

The OSI Model

- *Conceptual* reference model for components of a communication system

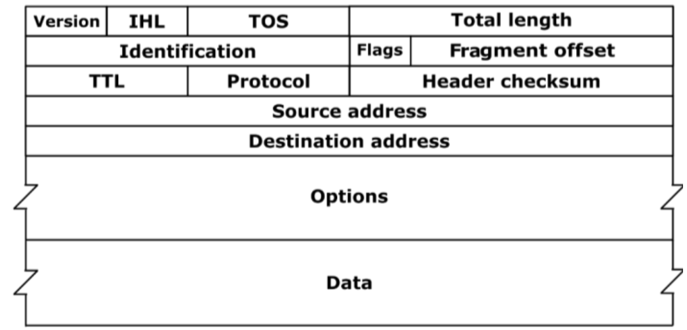
- The software (and hardware) that gets your packets from one side of the internet to the other is organized into "**layers**"
  - At each step of the way, each layer is present and doing some work to get your bytes to the next hop



The OSI Model Analogy

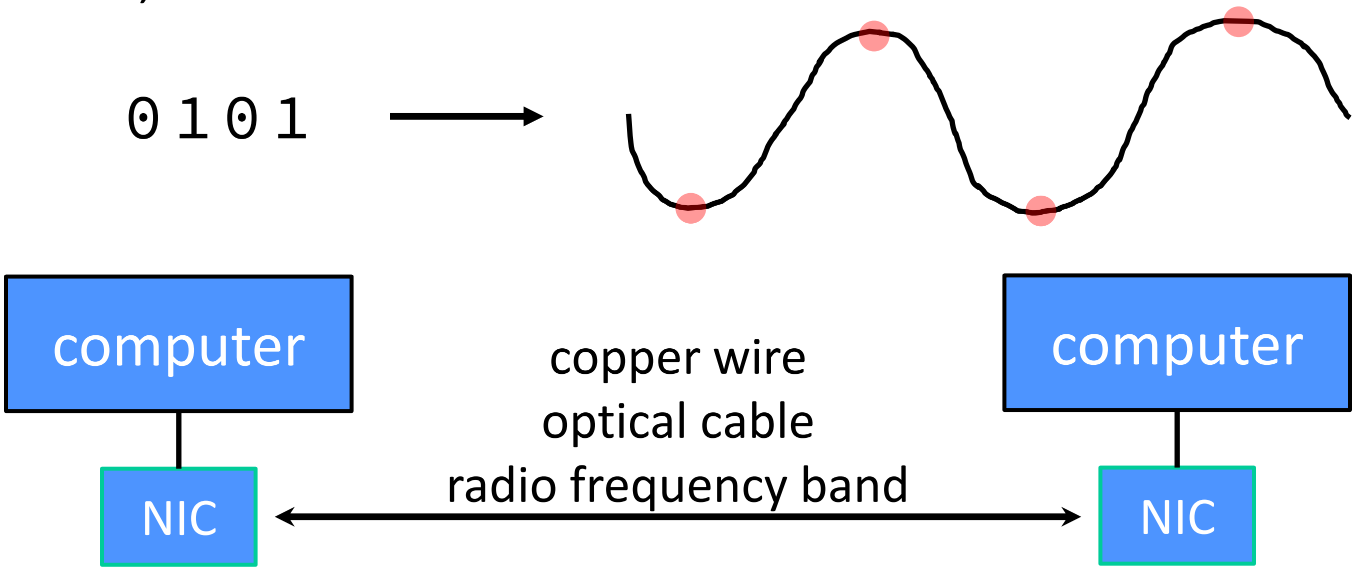
Protocols and Headers

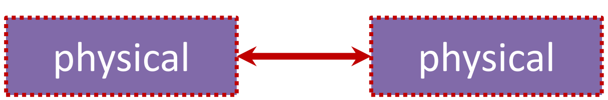
- Your data packet includes some metadata bytes (**packet headers**) for each layer so the next computer will know what to do with it
- We **standardize** those headers, and the behavior of the software reading them, in things called **network protocols**



The Physical Layer

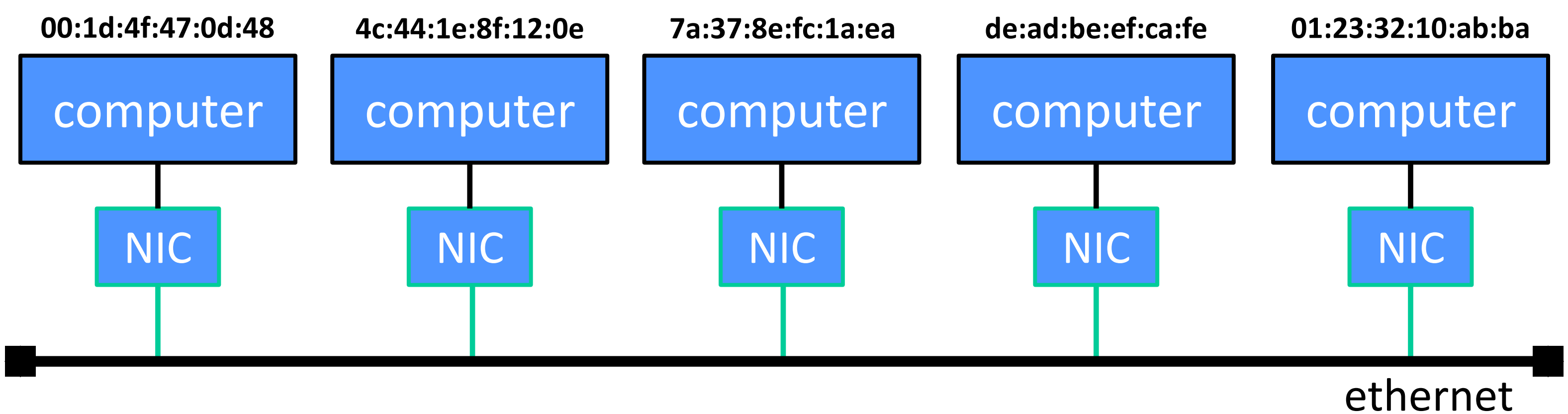
- Individual bits are transmitted over a physical medium
  - Physical layer specifies how bits are encoded at a signal level
  - Many choices, *e.g.*, encode “1” as +1v, “0” as -0v; or “0”=+1v, “1”=-1v, …

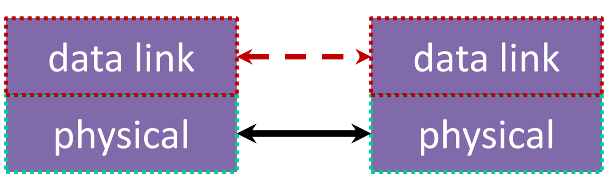


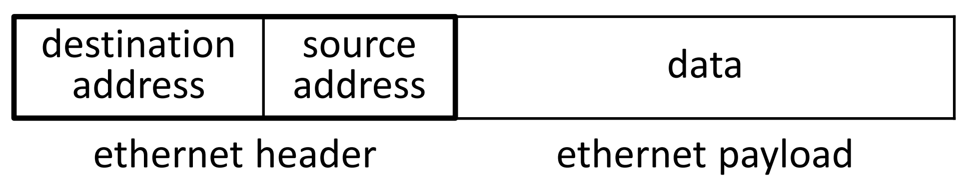


The Data Link Layer

- Multiple computers on a LAN contend for the medium
  - Media access control (MAC) specifies how computers cooperate
  - Link layer also specifies how bits are "packetized" and network interface controllers (NICs) are addressed

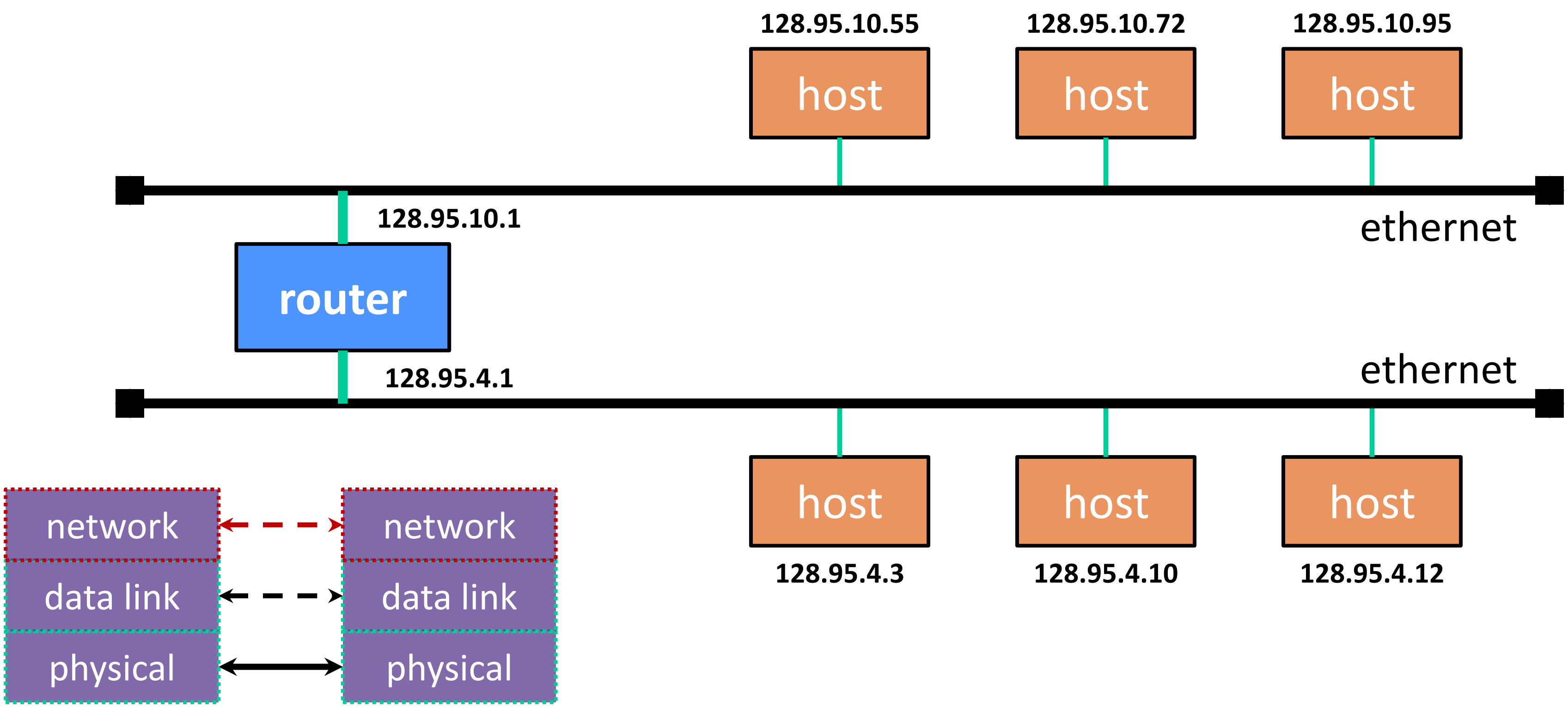






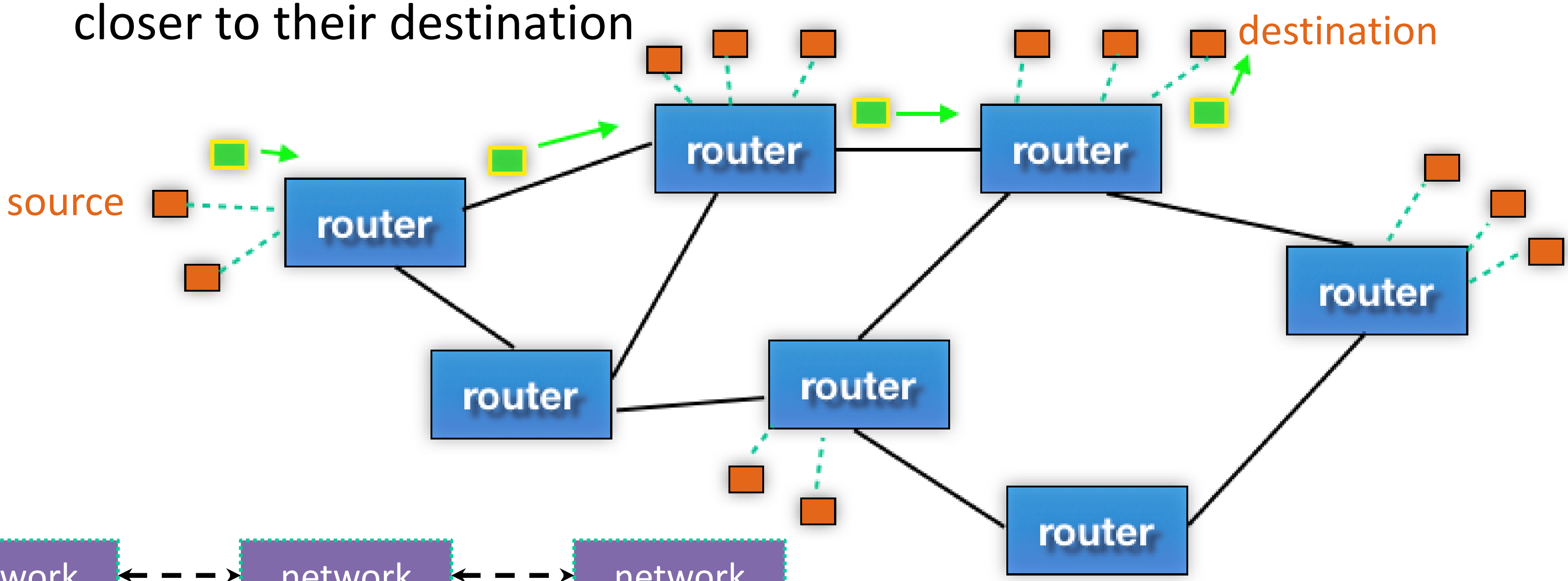
The Network Layer (IP)

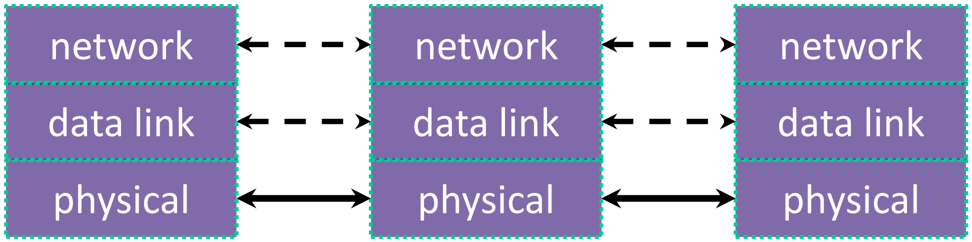
- Internet Protocol routes packets across multiple networks
  - Every computer has a unique IP address
  - Individual networks are connected by routers that span networks



The Network Layer (IP): Routing

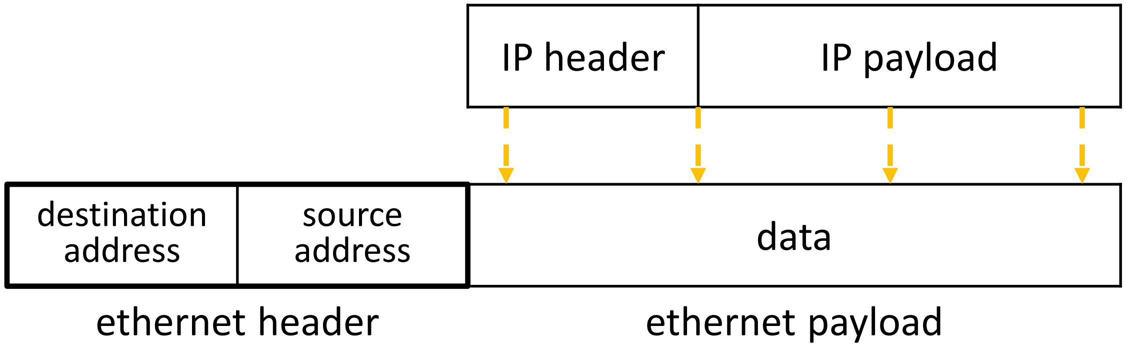
- There are protocols to:
  - Let a host map an IP to MAC address on the same network
  - Let a router learn about other routers to get IP packets one step closer to their destination

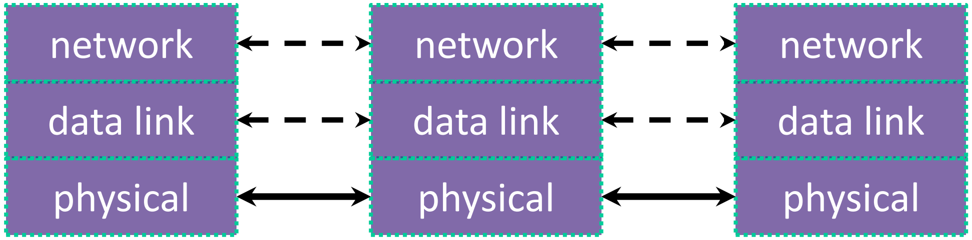




The Network Layer (IP): Packets

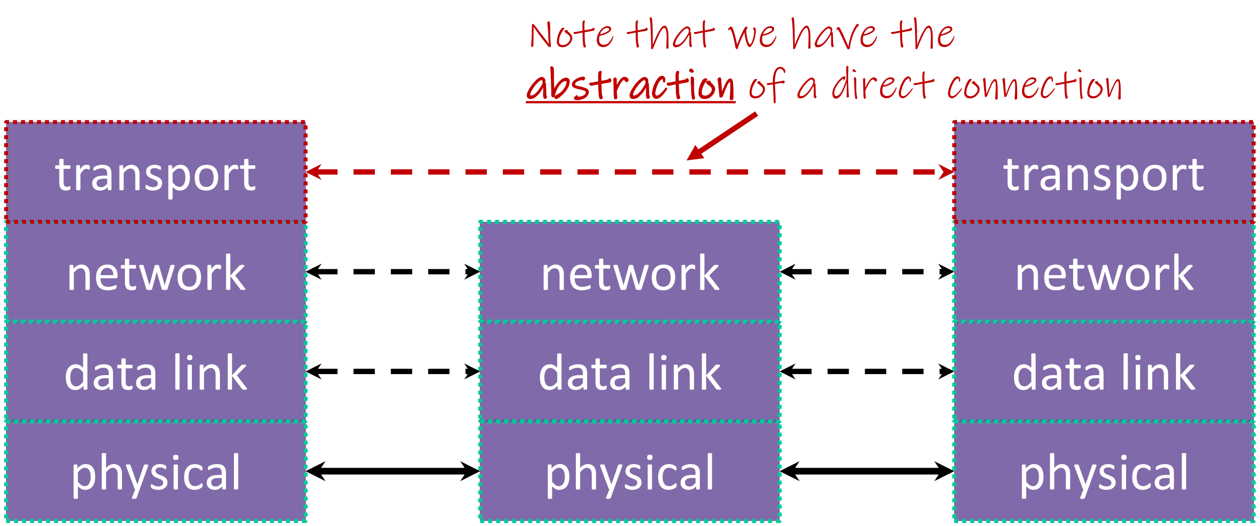
- Packet encapsulation:
  - An IP packet is encapsulated as the payload of an Ethernet frame
  - As IP packets traverse networks, routers pull out the IP packet from an Ethernet frame and plunk it into a new one on the next network





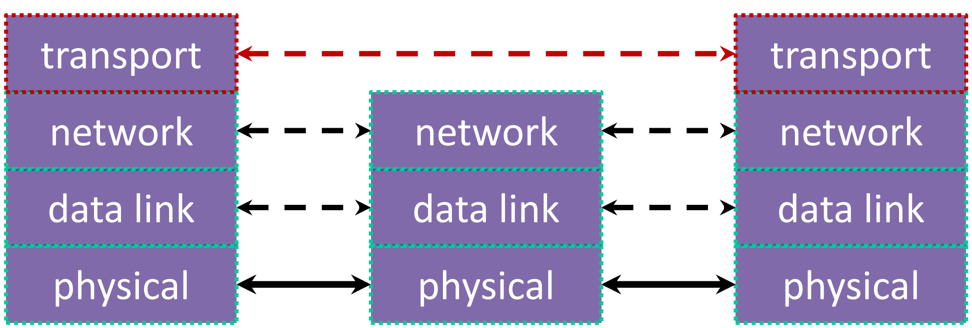
The Transport Layer

- Provides interface to treat the network as a **data stream**
- Provides different *protocols* to interface between source and destination:
  - *e.g.*, Transmission Control Protocol (TCP), User Datagram Protocol (UDP)
  - These protocols still work with packets, but manage their order, reliability, multiple applications using the network…



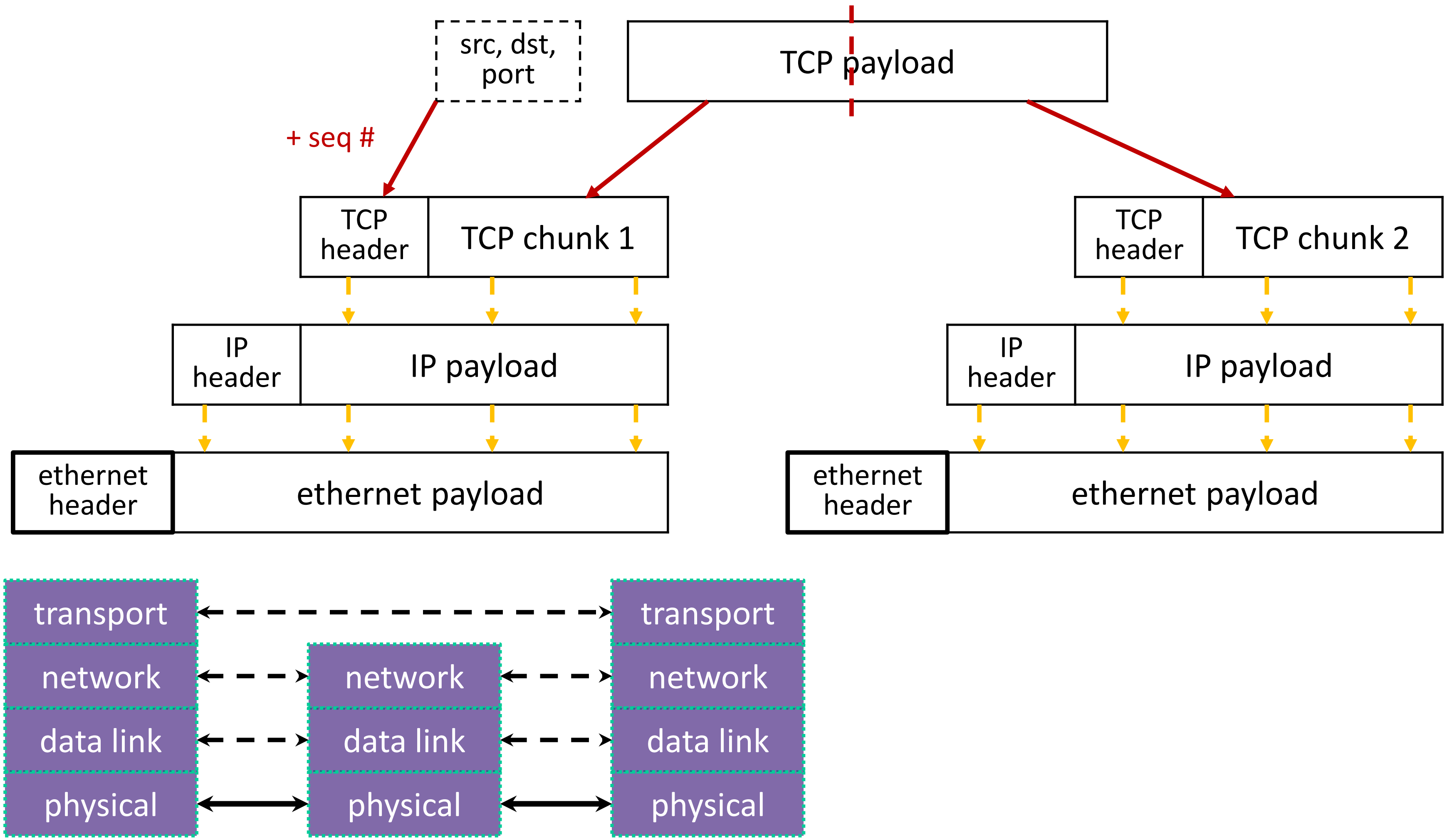
The Transport Layer: TCP

- Transmission Control Protocol (TCP):
  - Provides applications with reliable, ordered, congestion-controlled byte streams
    - Sends stream data as multiple IP packets (differentiated by sequence numbers) and retransmits them as necessary
    - When receiving, puts packets back in order and detects missing packets
  - A single host (IP address) can have up to 216 = 65,535 “ports”
    - Kind of like an apartment number at a postal address (your applications are the residents who get mail sent to an apt. #)



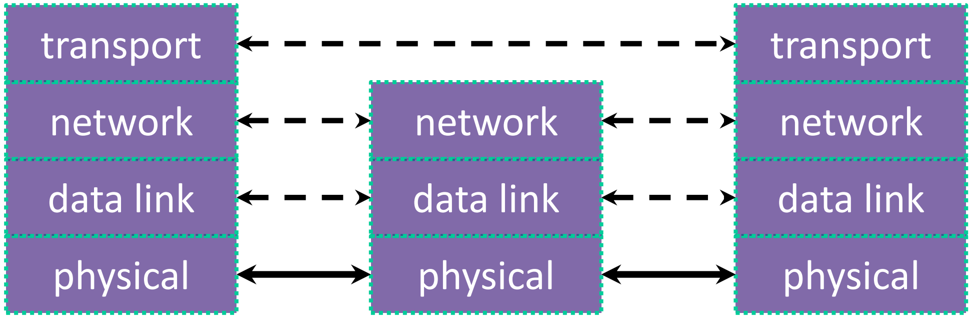
The Transport Layer: TCP Packets

- Packet encapsulation – one more nested layer!



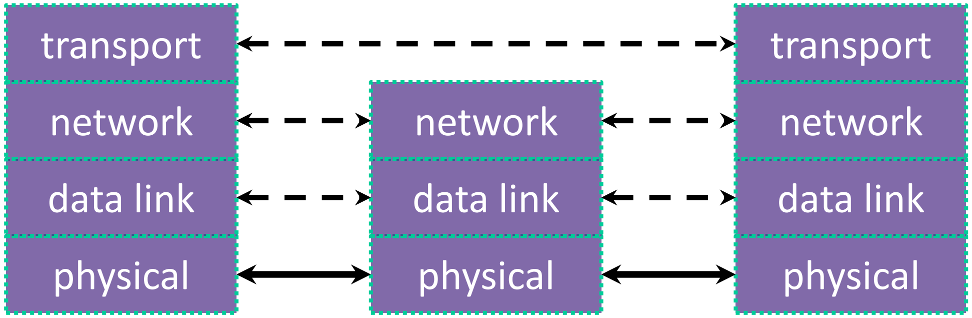
The Transport Layer: TCP API

- 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 and servers **`read`**`()` and **`write`**`()` data to each other



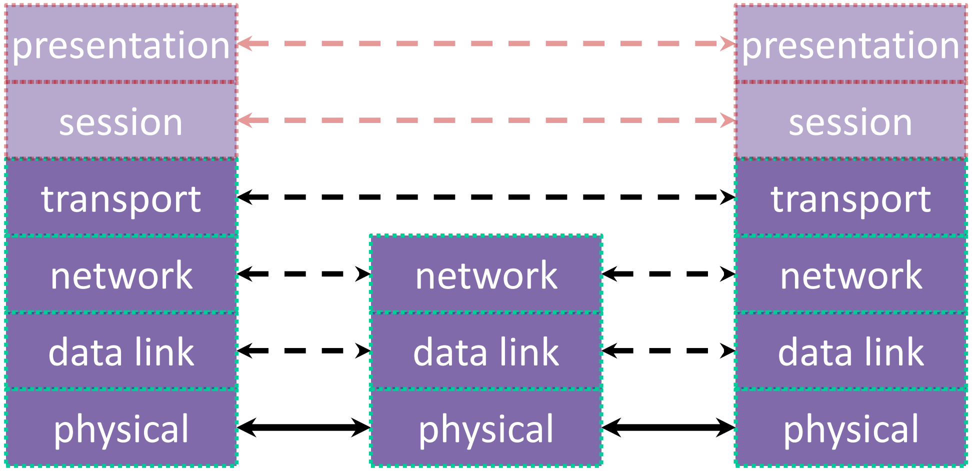
The Transport Layer: UDP

- User Datagram Protocol (UDP):
  - Provides applications with *unreliable* packet delivery
  - UDP is a really thin, simple layer on top of IP
    - Datagrams still are fragmented into multiple IP packets



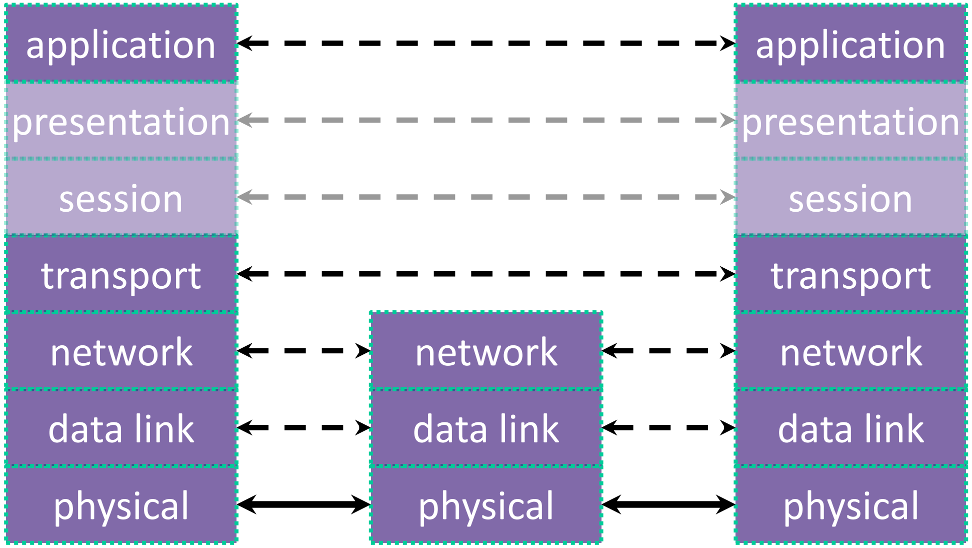
The (Mostly Missing) Layers 5 & 6

- Layer 5: Session Layer
  - Would handle establishing & terminating application sessions
  - Remote Procedure Call (RPC) kind of fits in here
- Layer 6: Presentation Layer
  - Would map application-specific data units into a more network-neutral representation
  - Encryption (SSL) kind of fits in here



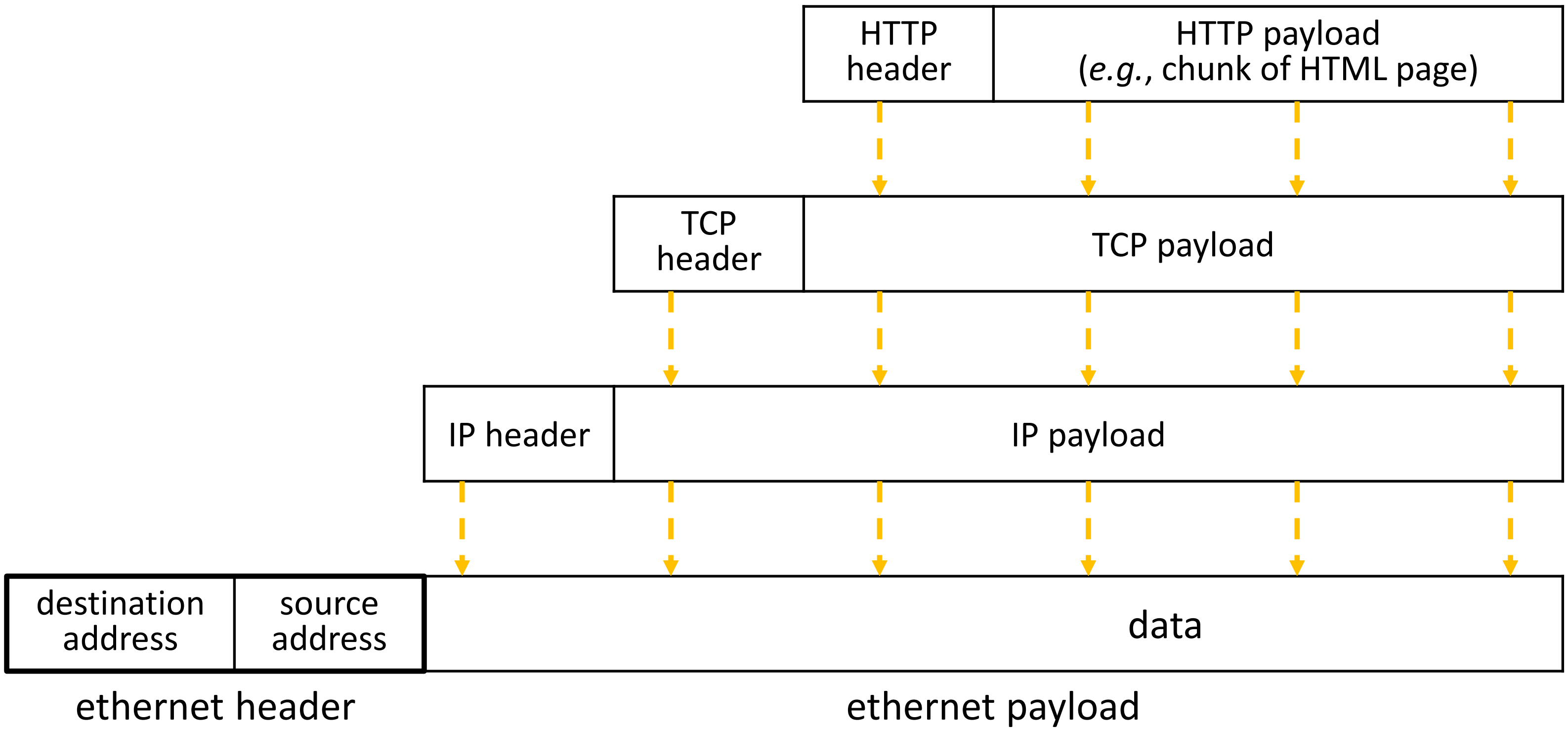
The Application Layer

- 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



The Application Layer: Packets (1/2)

- Packet encapsulation:



The Application Layer: Packets (2/2)

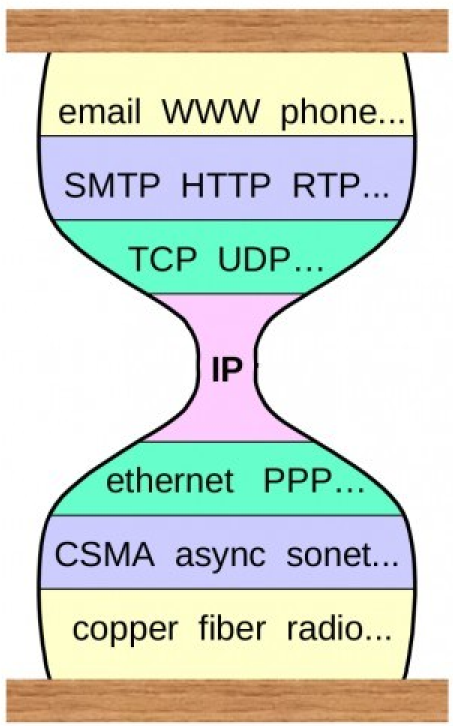
- Packet encapsulation:

ethernet header	IP header	TCP header	HTTP header	HTTP payload (*e.g.*, chunk of HTML page)

The Application Layer

- Popular application-level protocols:
  - **DNS:** translates a domain name (e.g., www.google.com) into one or more IP addresses (e.g., 74.125.197.106)
    - Domain Name System
    - An hierarchy of DNS servers cooperate to do this
  - **HTTP:** web protocols
    - Hypertext Transfer Protocol
  - **SMTP, IMAP, POP:** mail delivery and access protocols
    - Secure Mail Transfer Protocol, Internet Message Access Protocol, Post Office Protocol
  - **SSH:** secure remote login protocol
    - Secure Shell
  - **bittorrent:** peer-to-peer, swarming file sharing protocol

The "Narrow Waist"



- Hahah that's a lot of protocols. How are we gonna do this 🥴 ...
- The bulk of the machinery running the internet **only deals with IP, TCP and UDP** ("TCP/IP")
  - How is the link layer handled? That's a **driver** problem
  - How does Instagram work? That's an **application** problem.
- This is called "**the narrow waist**" of the internet

netcat demo (if time)

- netcat (`nc`) is "a computer networking utility for reading from and writing to network connections using TCP or UDP"
  - https://en.wikipedia.org/wiki/Netcat

  - Listen on port: `nc -l <port>`
  - Connect: `nc <IPaddr> <port>`
    - Local host: `127.0.0.1`