CSE 333

Lecture 19 -- C++ final details, networks

Steve Gribble

Department of Computer Science & Engineering

University of Washington

Administrivia

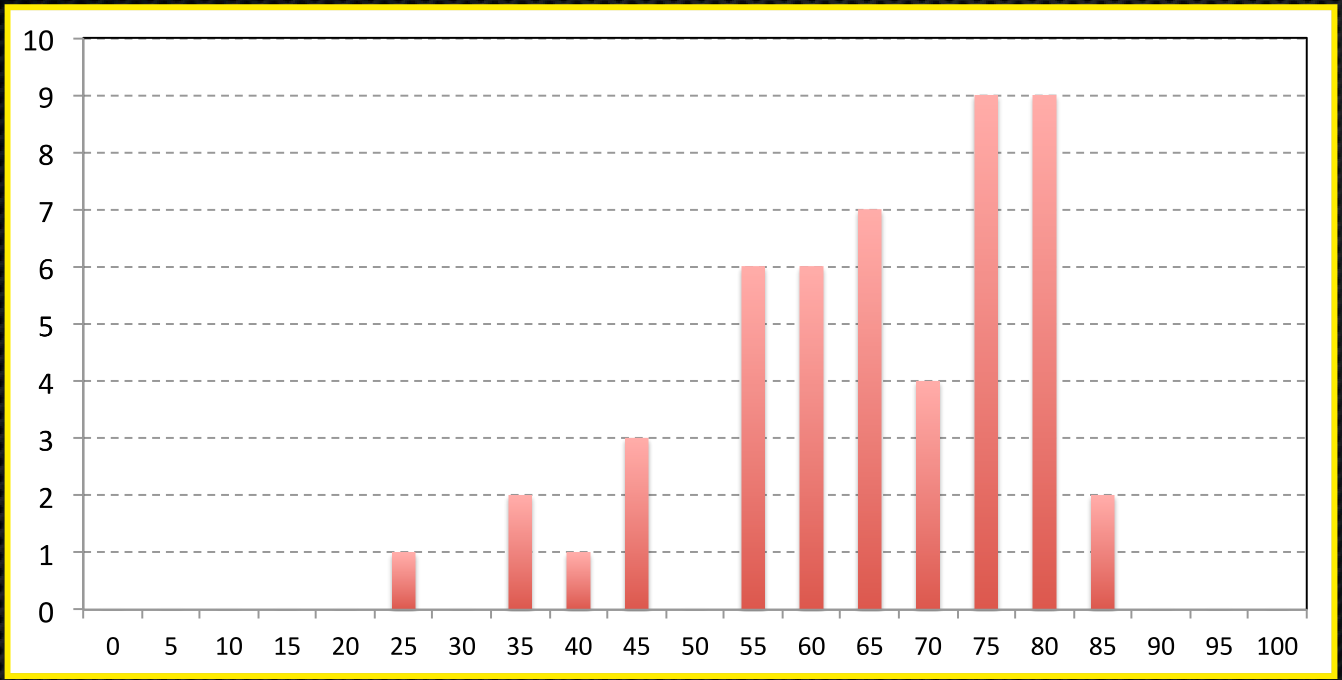
HW3 is due in 5 days!

- we're up to 6 bugs for you to patch in our code :)
- see the sticky thread at the top of the discussion board

Midterm

- handed back today
    - please check our arithmetic
- our grading key is up on the Web (midterm_soln.pdf)

Midterm scores



mean: 66 max: 89 min: 27

Q1: 21   Q2: 8   Q3: 16   Q4: 21

Frequent errors

Q2

- confusion over C-style strings ("foo") and C++ string objects, and how they interrelate
- missed the need for a default constructor

Q3

- lots of confusion over output parameters and pointers
- missing header guards, .h comments
- missing free( ), good test cases

Today

Finish off a couple of final C++ topics

- typecasting, C++ style
- implicit conversion

Start into network programming

Explicit casting in C

C's *explicit typecasting* syntax is simple

```
lhs = (new type) rhs;
```

- C's explicit casting is used to...
  - convert between pointers of arbitrary type
  - forcibly convert a primitive type to another
    - e.g., an integer to a float, so that you can do integer division

```
int x = 5;
int y = 2;
printf("%d\n", x / y);             // prints 2
printf("%f\n", ((float) x) / y);   // prints 2.5
```

C++

You can use C-style casting in C++, but C++ provides an alternative 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)

static_cast

C++'s static_cast can convert:

- pointers to classes **of related type**
  - get a compiler error if you attempt to static_cast between pointers to non-related classes
  - dangerous to cast a pointer to a base class into a pointer to a derived class
- non-pointer conversion
  - float to int, etc.

static_cast is checked at compile time

staticcast.cc

```
class Foo {
 public:
  int x_;
};

class Bar {
 public:
  float x_;
};

class Wow : public Bar {
 public:
  char x_;
};

int main(int argc, char **argv) {
  Foo a, *aptr;
  Bar b, *bptr;
  Wow c, *cptr;

  // compiler error
  aptr = static_cast<Foo *>(&b);

  // OK
  bptr = static_cast<Bar *>(&c);

  // compiles, but dangerous
  cptr = static_cast<Wow *>(&b);
  return 0;
}
```

dynamic_cast

C++'s 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 a full derived object

dynamiccast.cc

```
class Base {
 public:
  virtual int foo() { return 1; }
  float x_;
};

class Deriv : public Base {
 public:
  char x_;
};

int main(int argc, char **argv) {
  Base b, *bptr = &b;
  Deriv d, *dptr = &d;

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

  // OK (run-time check passes).
  dptr = dynamic_cast<Deriv *>(bptr);
  assert(dptr != NULL);

  // Run-time check fails, so the
  // cast returns NULL.
  bptr = &b;
  dptr = dynamic_cast<Deriv *>(bptr);
  assert(dptr != NULL);

  return 0;
}
```

const_cast

Is used to strip or add const-ness

- dangerous!

constcast.cc

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

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

main() {
  int x = 7;
  bar(&x);
}
```

reinterpret_cast

casts between incompatible types

- storing a pointer in an int, or vice-versa
  - works as long as the integral type is “wide” enough
- converting between incompatible pointers
  - dangerous!

Implicit conversion

The compiler tries to infer some kinds of conversions

- when you don't specify an explicit cast, and types are not equal, the compiler looks for an acceptable implicit conversion

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

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

Sneaky implicit conversions

How did the (const char *) --> string conversion work??

- 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
  - can't do int --> Foo --> Baz

implicit.cc

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

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

int main(int argc, char **argv) {
  // The compiler uses Foo's
  // (int x) constructor to make
  // an implicit conversion from
  // the int 5 to a Foo.


  // equiv to return Bar(Foo(5));
  // !!!
  return Bar(5);
}
```

Avoiding sneaky implicits

Declare one-argument constructors as "explicit" if you want to disable them from being used as an implicit conversion path

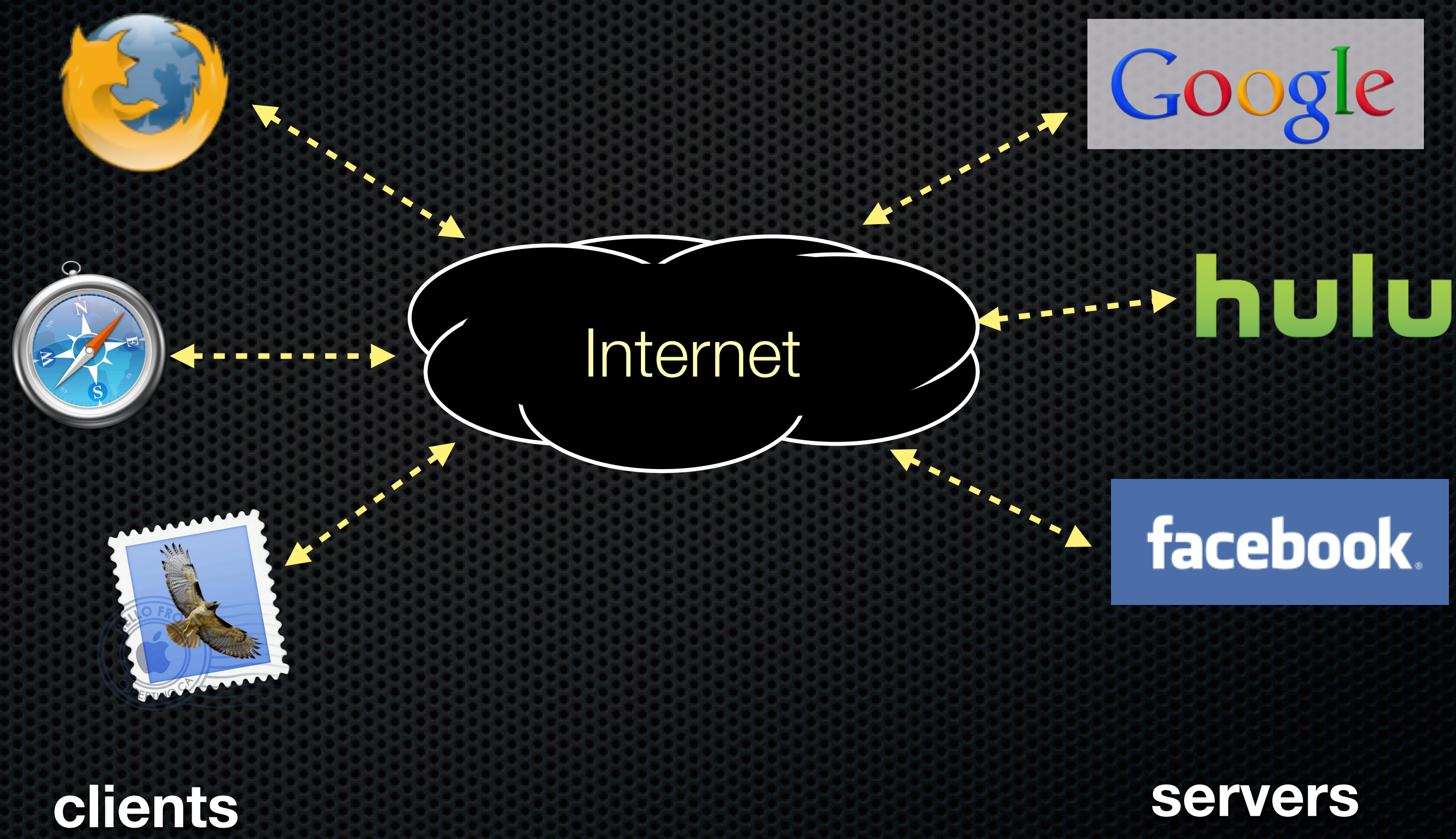
- usually a good idea

explicit.cc

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

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

int main(int argc, char **argv) {
  // The compiler uses Foo's
  // (int x) constructor to make
  // an implicit conversion from
  // the int 5 to a Foo.


  // compiler error
  return Bar(5);
}
```

Networking

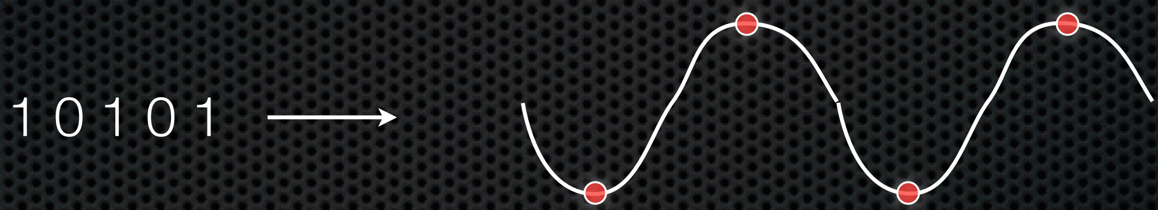
Networks from 10,000ft

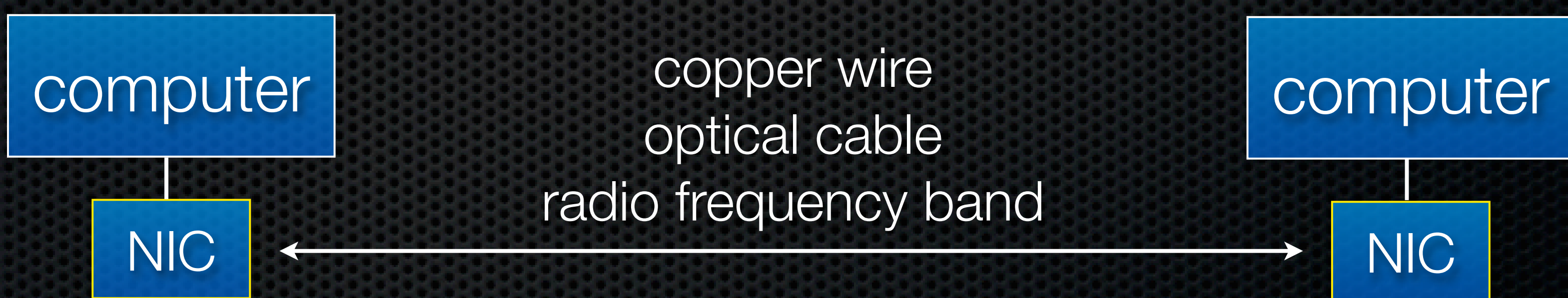


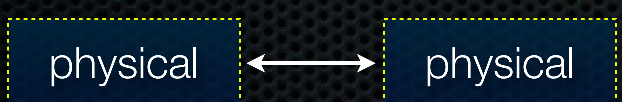
The “physical” layer

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



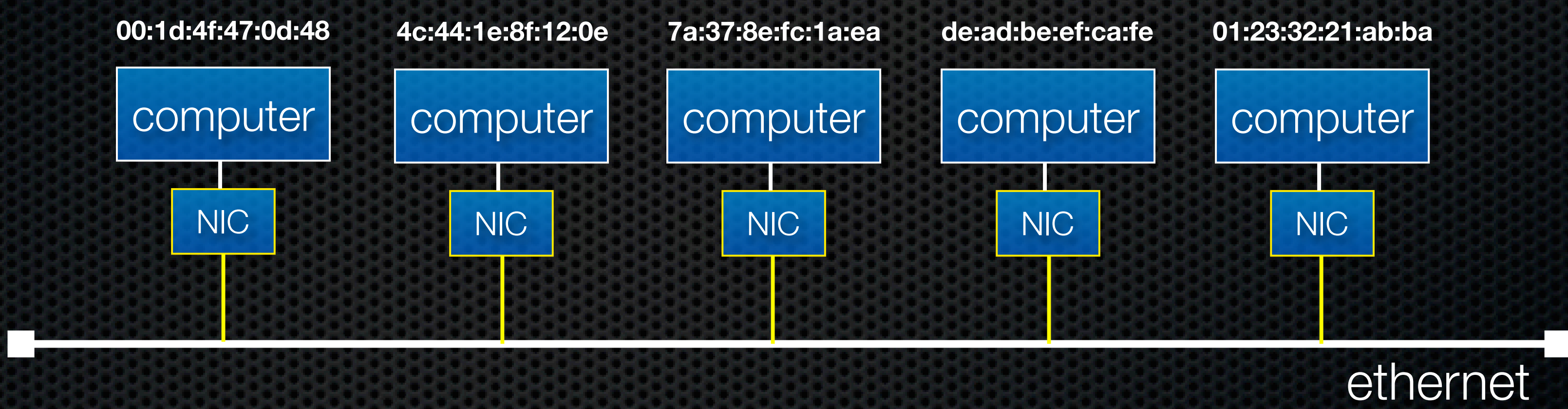


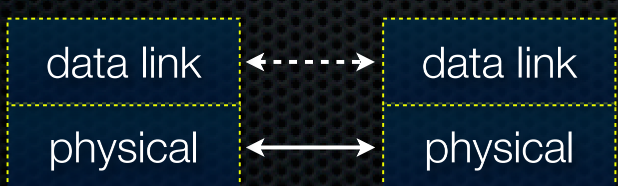


The “data link” layer

Multiple computers on a LAN contend for the network medium

- media access control (MAC) specifies how computers cooperate
- link layer also specifies how bits are packetized and NICs are addressed

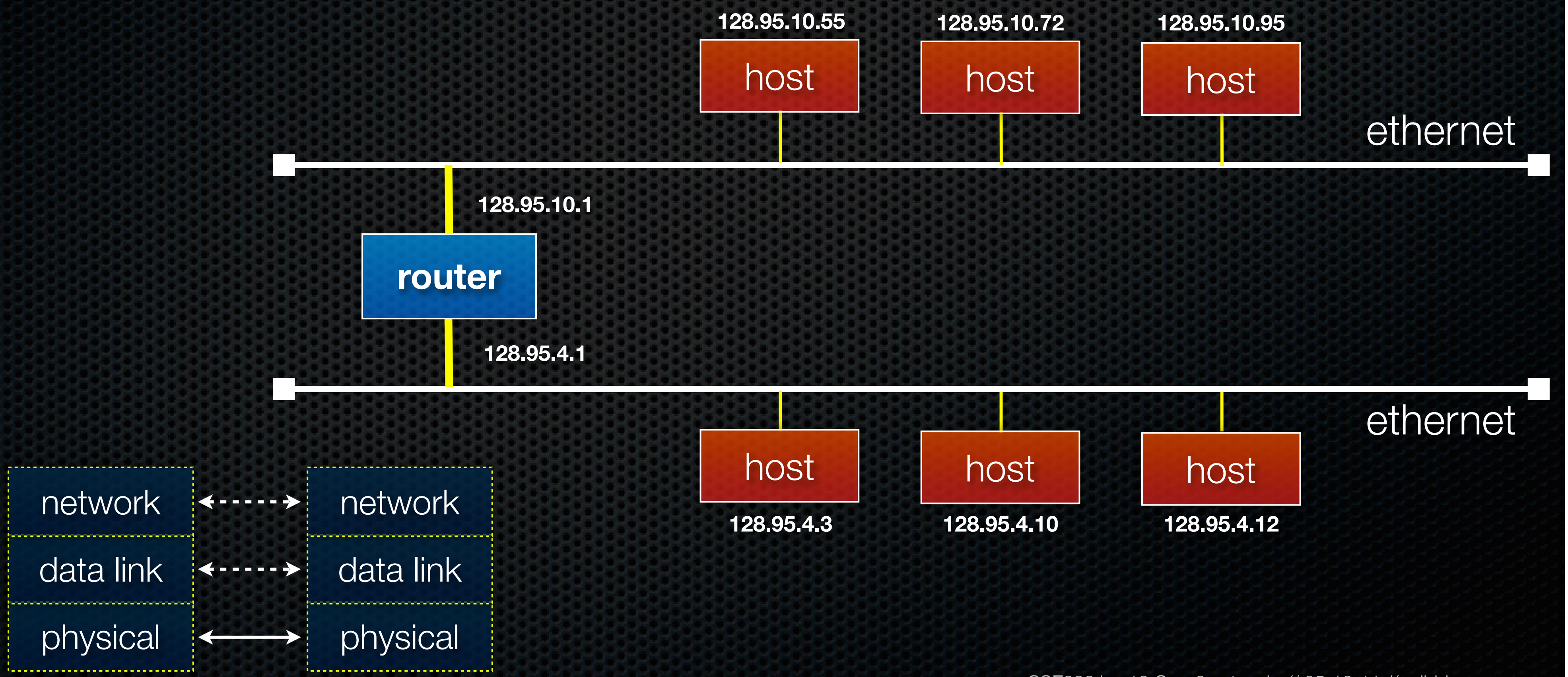




The "network" layer (IP)

The Internet Protocol (IP) routes packets across multiple networks

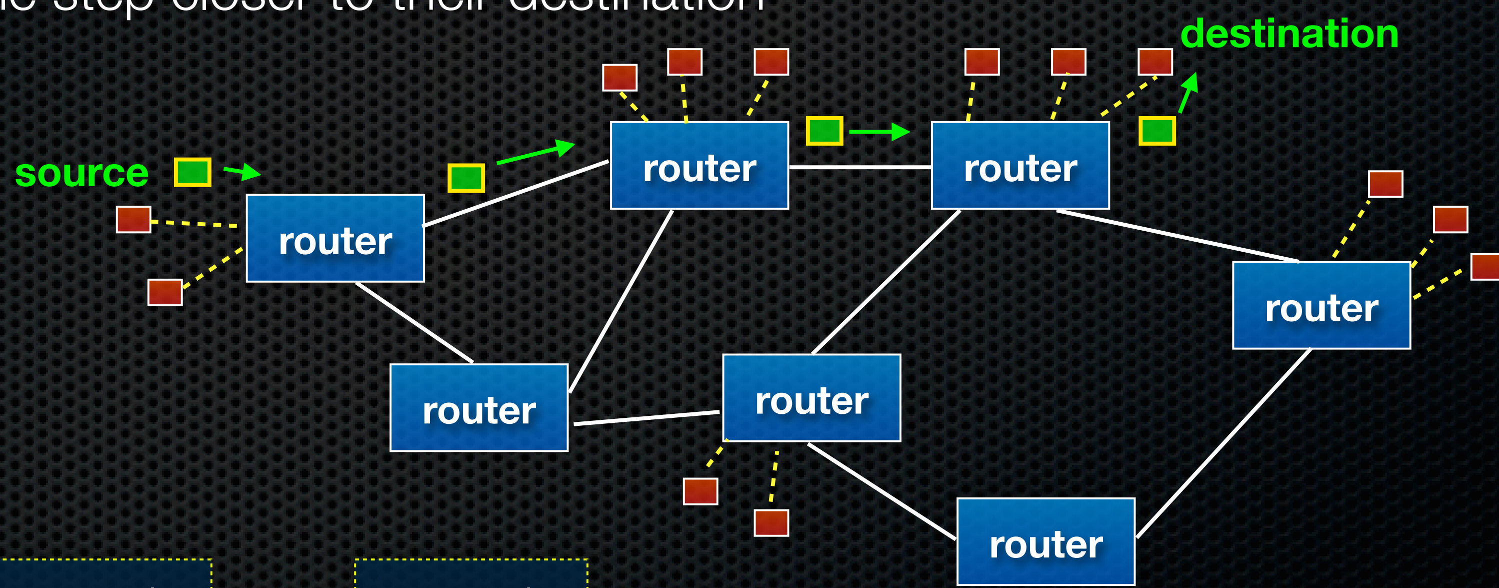
- every computer has a unique Internet address (IP address)
- individual networks are connected by routers that span networks



The “network” layer (IP)

Protocols to:

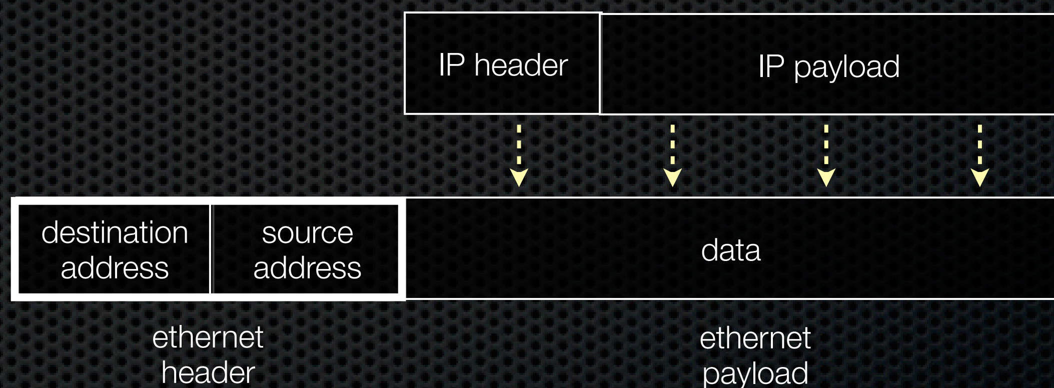
- let a host find the MAC address of an IP address on the same network
- let a router learn about other routers and figure out how to get IP packets one step closer to their destination



The “network” layer (IP)

Packet encapsulation

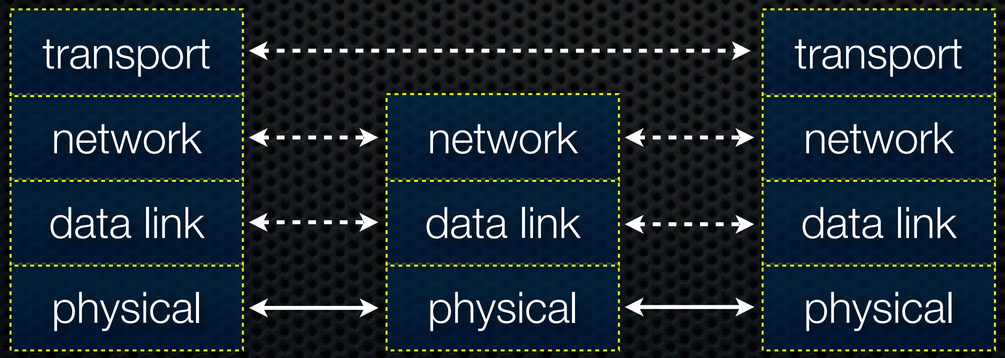
- an IP packet is encapsulated as the payload of an Ethernet frame
- as IP packets traverse networks, routers pull out the IP packet from an ethernet frame and plunk it into a new one on the next network



The “transport” layer (TCP, UDP)

TCP

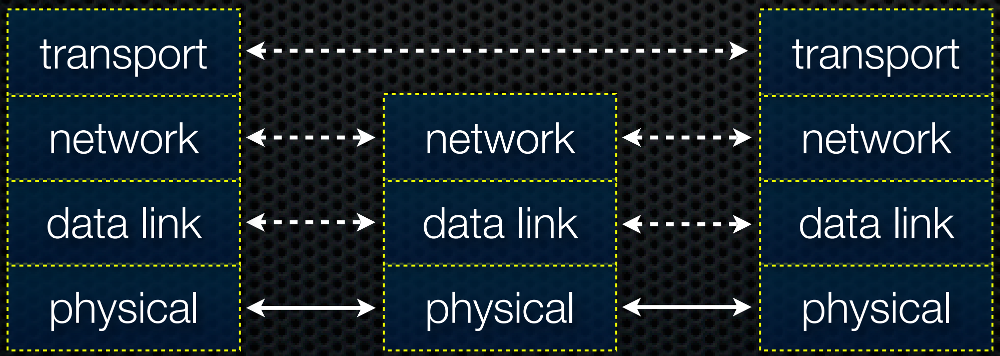
- the “transmission control protocol”
- provides apps with reliable, ordered, congestion-controlled byte streams
- fabricates them by sending multiple IP packets, using sequence numbers to detect missing packets, and retransmitting them
- a single host (IP address) can have up to 65,535 “ports”
  - kind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



The “transport” layer (TCP, UDP)

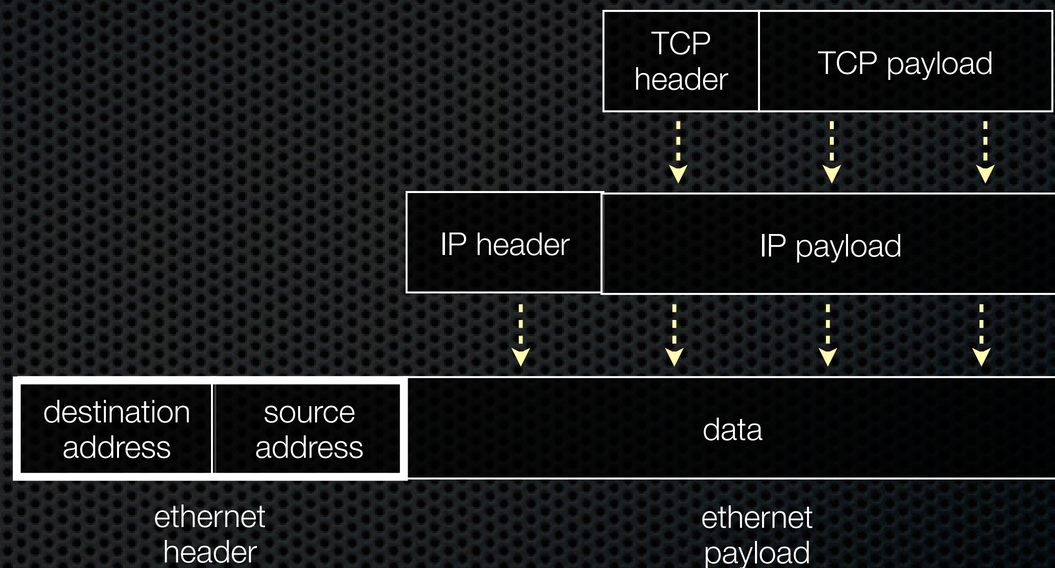
TCP

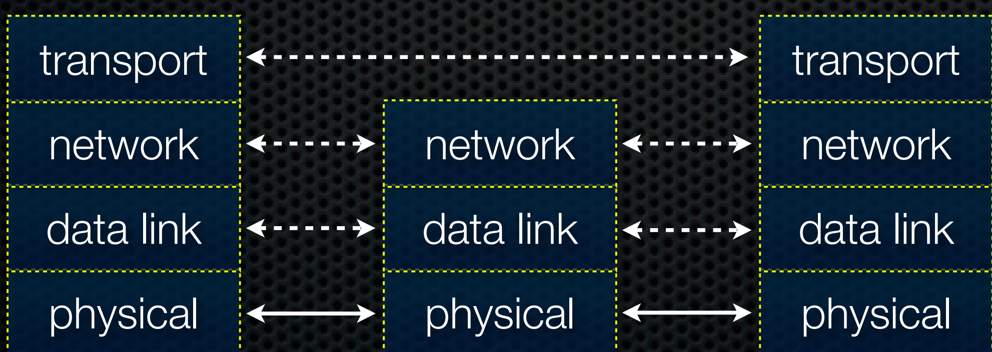
- useful analogy: how would you send a book by mail via postcards?
- split the book into multiple postcards, send each one by one, including sequence numbers that indicate the assembly order
- receiver sends back postcards to acknowledge receipt and indicate which got lost in the mail



The "transport" layer (TCP)

Packet encapsulation -- same as before!

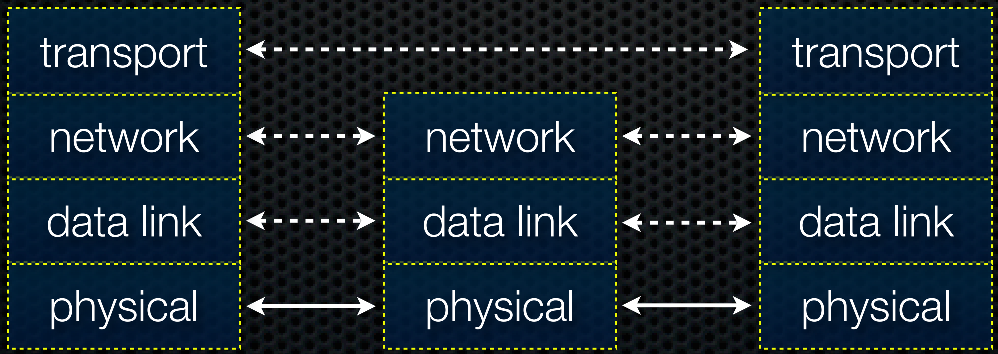




The "transport" layer (TCP)

Applications use OS services to establish TCP streams

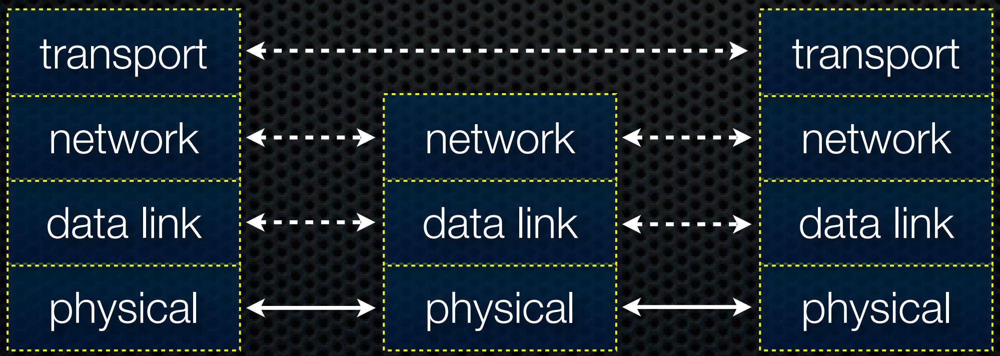
- the "Berkeley sockets" API -- a set of OS system calls
- clients ***connect( )*** to a server IP address + application port number
- servers ***listen( )*** for and ***accept( )*** client connections
- clients, servers ***read( )*** and ***write( )*** data to each other



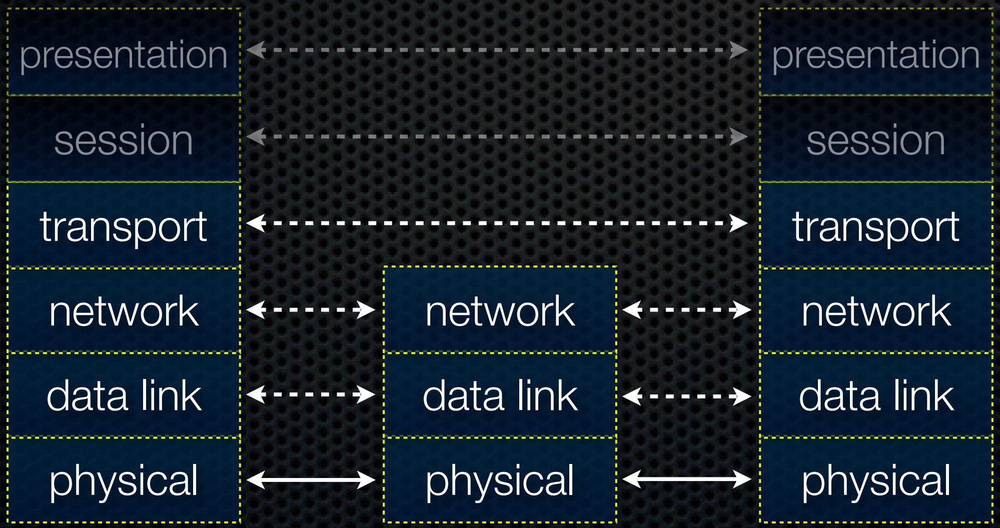
The "transport" layer (UDP)

UDP

- the "user datagram protocol"
- provides apps with unreliable packet delivery
- UDP datagrams are fragmented into multiple IP packets
  - UDP is a really thin, simple layer on top of IP



The (mostly missing) layers 5,6



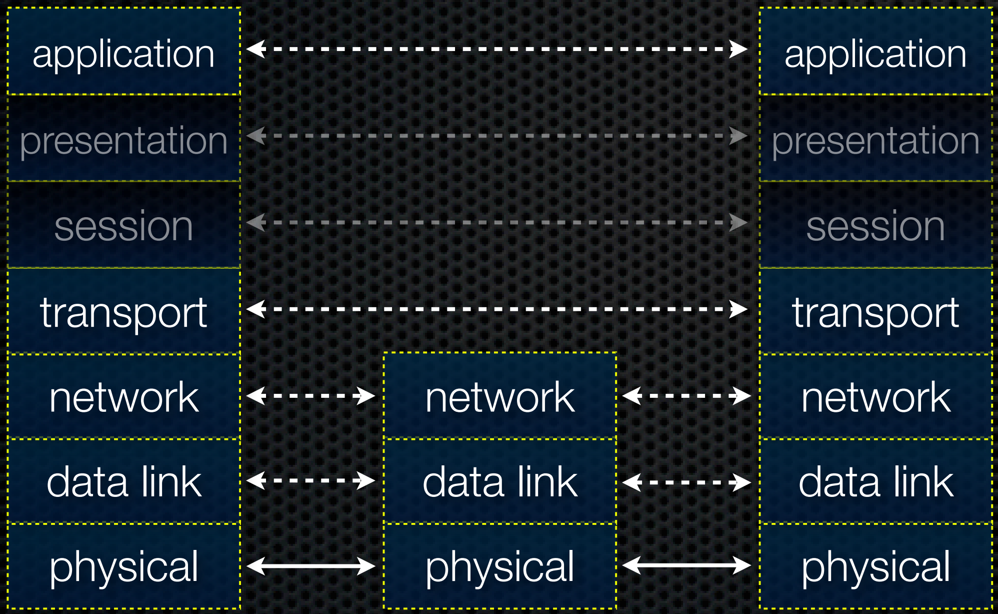
Layer 5: session layer

- 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

The "application" layer

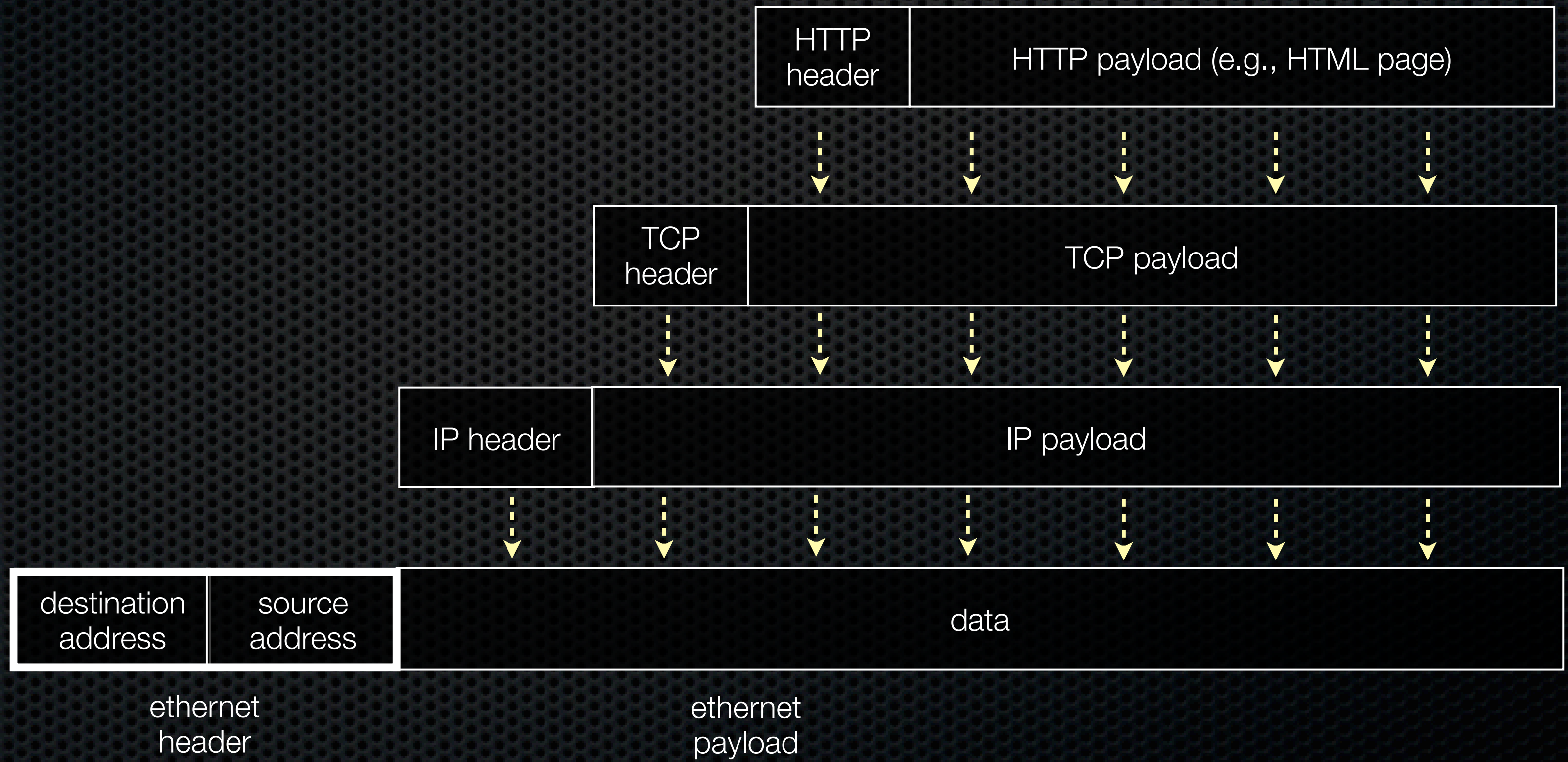


Application protocols

- 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

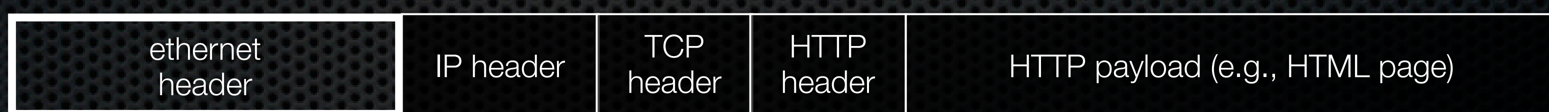
The "application" layer

Packet encapsulation -- same as before!



The "application" layer

Packet encapsulation -- same as before!



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

See you on Monday!