Week 9 – Friday

COMP 3400

Last time

- What did we talk about last time?
- Thread safety
- POSIX threads

Questions?

Assignment 5

Review

Exam 2 format

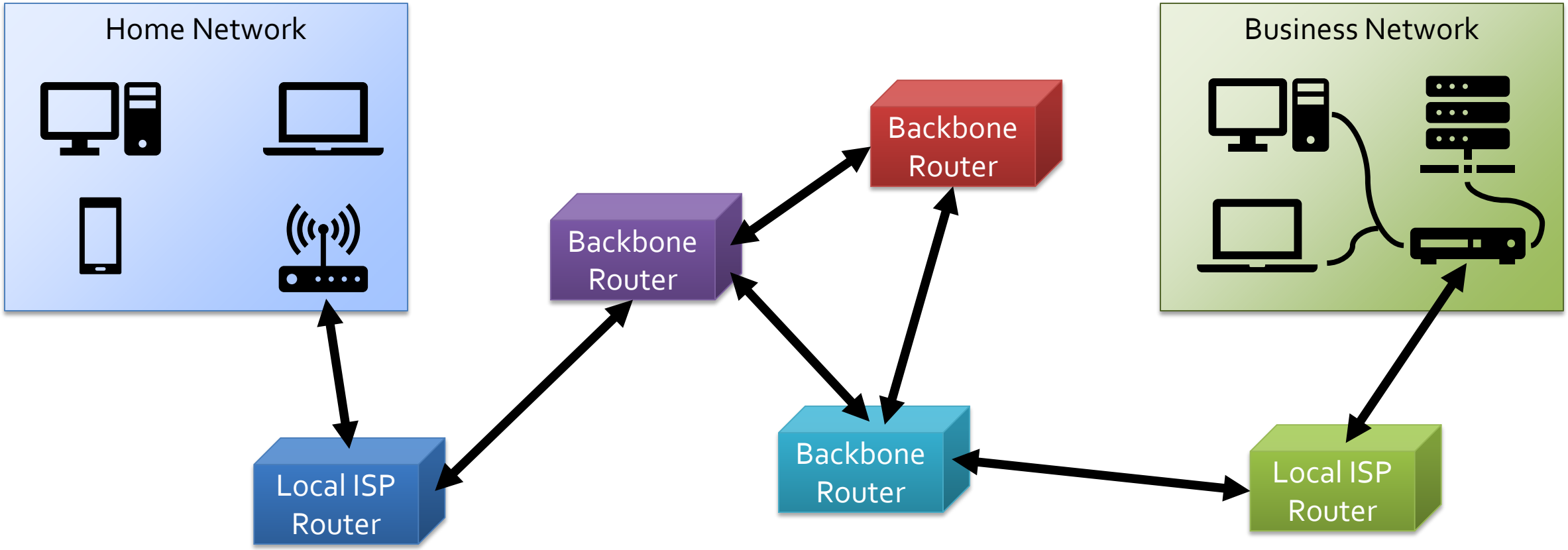
- Exam 2 will be similar in format to Exam 1:
  - Mostly short answer questions
  - Possibly a matching problem or two
  - Probably one or two short programming problems
  - Possibly a debugging problem
- The focus is networking
- Threading isn't on the exam

Networking

Networking

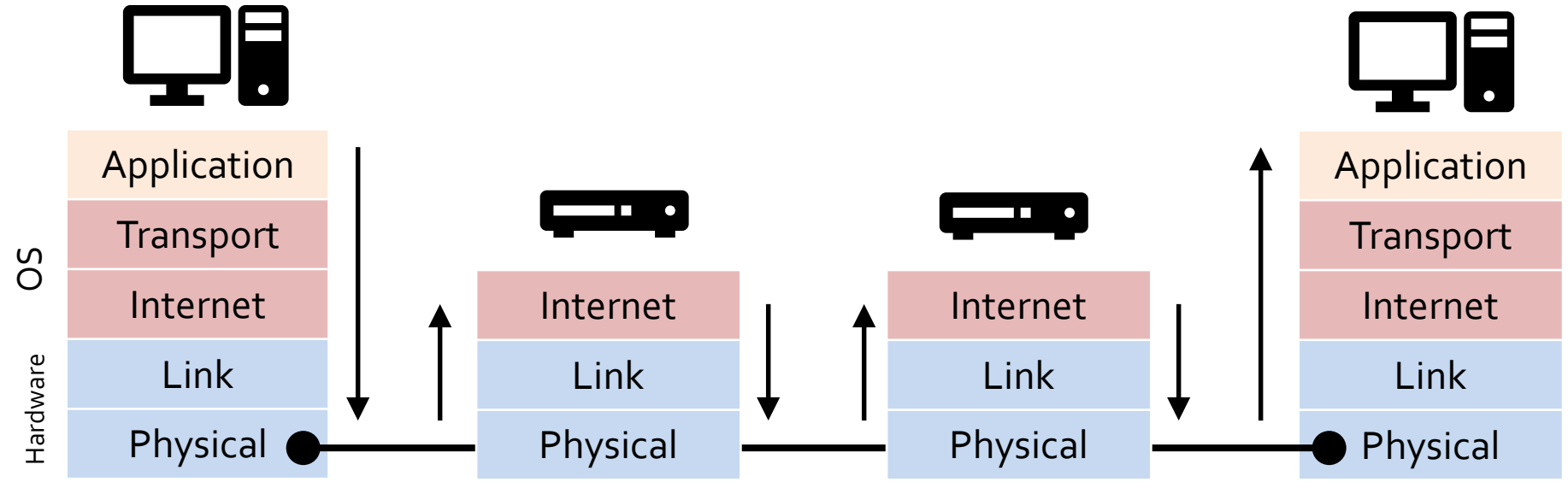
- IPC can be extended to processes working on *different* machines
  - The goal is still to do more complex computation than would be possible or convenient with a single process
- However, communicating between processes on different machines is much less **reliable**
- This unreliability is particularly strong on the Internet
  - Servers can crash
  - Network outages can cause machines to be unreachable
  - General chaos means that everything can be working reasonably well and yet messages are sometimes lost

Visualization



Layer models

- Networking always involves layers
- Each layer talks to the one above and below it and can often be swapped out for different protocols that provide similar services
- For this class, we'll be talking about a five layer Internet model
  - Simpler than the 7 layer OSI model
  - Remember that the purpose of models is to understand complex systems
- Different people use different names for the same layers



Layers

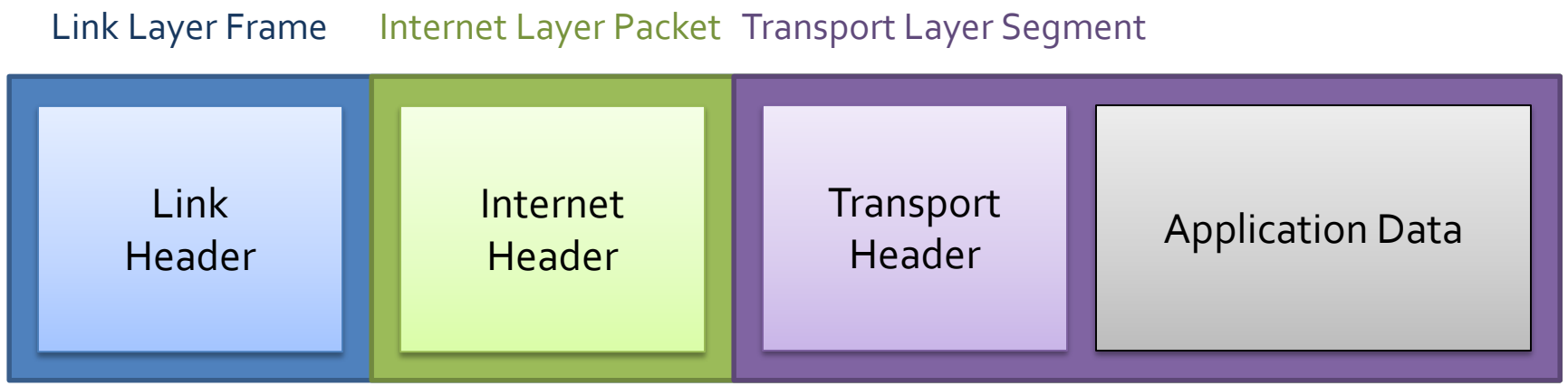
- Application layer
  - Logical endpoints of communication
  - Actual processes that are talking to each other
  - Example protocols: HTTP, FTP, SSH
- Transport layer
  - Implemented as sockets, the software endpoints of communication
  - Provides message passing system calls
  - Breaks down messages into fixed-size segments
  - Demultiplexes: Takes all messages arriving at the machine and sends them to appropriate processes by port number
  - Example protocols: TCP and UDP

Layers continued

- Internet layer
  - Provides point-to-point communication between hosts and routers
  - Uses addresses to determine the logical location of hosts
  - Determines the route that packets will travel along
- Link layer
  - Sends packets between devices on the same network
  - Closely tied to hardware
  - Example protocols: Ethernet, WiFi, Bluetooth
- Physical layer
  - Actual hardware
  - Interprets electrons or radio waves as bits

Packets

- Every system of computer networking we'll talk about uses packets
  - A **packet** is chunk of data with header information like ports and addresses
- Each layer encapsulates the data from the layer above it for transmission
  - The application data is usually bigger than the headers, but we don't draw it to scale
- When being technical, each layer calls its packets different things:
  - Transport layer: **segments**
  - Internet layer: **packets**
  - Link layer: **frames**
- **Datagram** is also used as a synonym for packet
- Be aware that all these terms get thrown around



Naming and addressing

- Local IPC used a name that often mapped to a path in the file system
- Networked IPC usually needs more information:
  - Host to connect to
  - Sometimes port
  - File or resource being requested
- A standard form for this information is a uniform resource identifier (URI):

```
URI = scheme:[//authority]path[?query][#fragment]
```

  - Note that brackets mark optional entities

Sockets

Sockets

- The most basic element of the networking arsenal is the **socket**
- A socket is half of a two-way connection between hosts
- We create a socket with a call to **`socket()`**

```
int socket (int domain, int type, int protocol);
```

  - Returns an **`int`**, essentially a file descriptor
  - Is similar to calling **`open()`** on a file
  - We can call **`read()`** and **`write()`** on socket file descriptors

IPv4

- Old-style IP addresses are often written in this form:
  - `74.125.67.100`
- 4 numbers between 0 and 255, separated by dots
- That's a total of 256^4 = 4,294,967,296 addresses
- But there are 8 billion people on earth …

IPv6

- IPv6 are the new IP addresses that are beginning to be used by modern hardware
  - Often written as 8 groups of 4 hexadecimal digits each
  - **`2001:0db8:85a3:0000:0000:8a2e:0370:7334`**
  - 1 hexadecimal digit has 16 possibilities
  - How many different addresses is this?
  - $16^{32} = 2^{128} \approx 3.4 \times 10^{38}$ is enough to have 500 trillion addresses for every cell of every person's body on Earth

Details for socket()

```
int socket (int domain, int type, int protocol);
```

- **domain**
  - What the socket will be used for
  - Typical values are IPv4, IPv6, or local communication
- **type**
  - Determines the transport layer
  - Usually TCP or UDP for this class
- **protocol**
  - Usually not used and set to **0**
  - Can be used for special raw sockets used for packer sniffers

Field	Constant	Purpose
domain	AF_INET	IPv4 addresses
	AF_INET6	IPv6 addresses
	AF_LOCAL	Unix domain socket for IPC
	AF_NETLINK	Netlink socket for kernel messages
	AF_PACKET	Raw socket type
type	SOCK_STREAM	Byte-stream communication, for TCP transport
	SOCK_DGRAM	Fixed-size messages, for UDP transport
	SOCK_RAW	Raw data that is not processed by transport layer
protocol	IPPROTO_RAW	IP datagrams without transport-layer processing
	ETH_P_ALL	Ethernet frames without network-layer processing

Networking data structures

- Different data structures are needed to specify addresses depending on what kind of networking is being done
- Since C doesn't have inheritance, structs with the same size are treated interchangeably and then cast to each other when appropriate
- One of these is **`struct sockaddr`**, which is 16 bytes in size

```
// generic address structure
struct sockaddr {
  sa_family_t sa_family; // two bytes: AF_INET, etc.
  char sa_data[14];
};
```

IPv4 socket addresses

- The structure for holding IPv4 addresses is identical in size to **`struct sockaddr`**

```
// IPv4 address structure
struct sockaddr_in {
  sa_family_t sin_family;
  in_port_t sin_port;
  struct in_addr sin_addr;
  char sin_zero[8];
};
struct in_addr {
  in_addr_t s_addr; // in_addr_t is an alias for uint32_t
};
```

Type	struct sockaddr															
Fields	sa_family		sa_data													
Data	02	00	00	50	5d	b8	d8	22	00	00	00	00	00	00	00	00
Fields	sin_family		sin_port		sin_addr				sin_zero							
Type	struct sockaddr_in															

IPv6 socket addresses

- IPv6 addresses are longer and consequently require bigger (and stranger looking) structs

```
// IPv6 address structure
struct sockaddr_in6 {
  sa_family_t sin6_family;
  in_port_t sin6_port;
  uint32_t sin6_flowinfo;
  struct in6_addr sin6_addr;  // IPv6 addresses are 128-bit
  uint32_t sin6_scope_id;
};

struct in6_addr {
  union {
    uint8_t  __u6_addr8[16];  // aliased as s6_addr
    uint16_t __u6_addr16[8];  // aliased as s6_addr16
    uint32_t __u6_addr32[4];  // aliased as s6_addr32
  } __u6_addr;
};
```

Endian conversion

- Rather than try to keep straight what the endianness of our machine and the endianness of the network is, we use a family of functions:
  - **`hton`**: host to network endianness
  - **`ntoh`**: network to host endianness
  - They come in **`l`** (long) versions (for 32-bit integers) or **`s`** (short) versions (for 16-bit integers)

```
uint32_t htonl (uint32_t hostlong);   // 32-bit from host to network
uint16_t htons (uint16_t hostshort); // 16-bit from host to network
uint32_t ntohl (uint32_t netlong);    // 32-bit from network to host
uint16_t ntohs (uint16_t netshort);   // 16-bit from network to host
```

Getting addresses from a host name

- DNS converts a host name to an IP address
- The **`getaddrinfo()`** function lets us get a linked list of matching addresses

```
int getaddrinfo (const char *name, const char *service,
const struct addrinfo *hints, struct addrinfo **results)
```

- The only annoying bit is that we have to fill out a hints structure
- A utility function **`freeaddrinfo()`** is provided to free the linked list structure when done with it

```
void freeaddrinfo (struct addrinfo *info);
```

The addrinfo struct

- The result of **`getaddrinfo()`** is stored into the pointer given by the last argument

```
struct addrinfo {
  int ai_flags;
  int ai_family;
  int ai_socktype;
  int ai_protocol;
  socklen_t ai_addrlen;
  char *ai_canonname;
  struct sockaddr *ai_addr;   // Pointer to address we need
  struct addrinfo *ai_next;   // Pointer to next addrinfo in linked list
};
```

Client side: connecting

- After all the madness is done getting the **`sockaddr`**, a client can connect to a listening server with the **`connect()`** function

```
int connect (int socket, const struct sockaddr *address, socklen_t address_len);
```

- The **`connect()`** function is a blocking call that will eventually succeed or fail to connect the socket file descriptor to an actual network connection
- If successful, we can read and write from that file descriptor

Server side: options

- The server side is more complicated
- It's useful to set some options on the socket using the (confusing) **`setsockopt()`** function

```
int setsockopt (int socket, int level, int option, const void *value, socklen_t lengeth);
```

  - Reusing the port, allowing reuse of the same port, even after crashing
  - Timing out on read messages
- After creating the socket:

```
//Allow port reuse
int on = 1;
setsockopt (socketfd, SOL_SOCKET, SO_REUSEADDR, (const void *) &on, sizeof (int));
// Set a 5-second timeout when waiting to receive
struct timeval timeout = { 5, 0 };
setsockopt (socketfd, SOL_SOCKET, SO_RCVTIMEO, (const void *) &timeout,
            sizeof (timeout));
```

Server side: binding and listening

- After creating the server socket (and maybe setting options), the next step is to bind the server to a port

```
int bind (int socket, const struct sockaddr *address, socklen_t address_len);
```

- For UDP, the server is then ready to receive messages
- For TCP, it has to listen on the socket

```
int listen (int socket, int backlog);
```

  - The backlog gives how many clients can queue up when trying to connect to the server

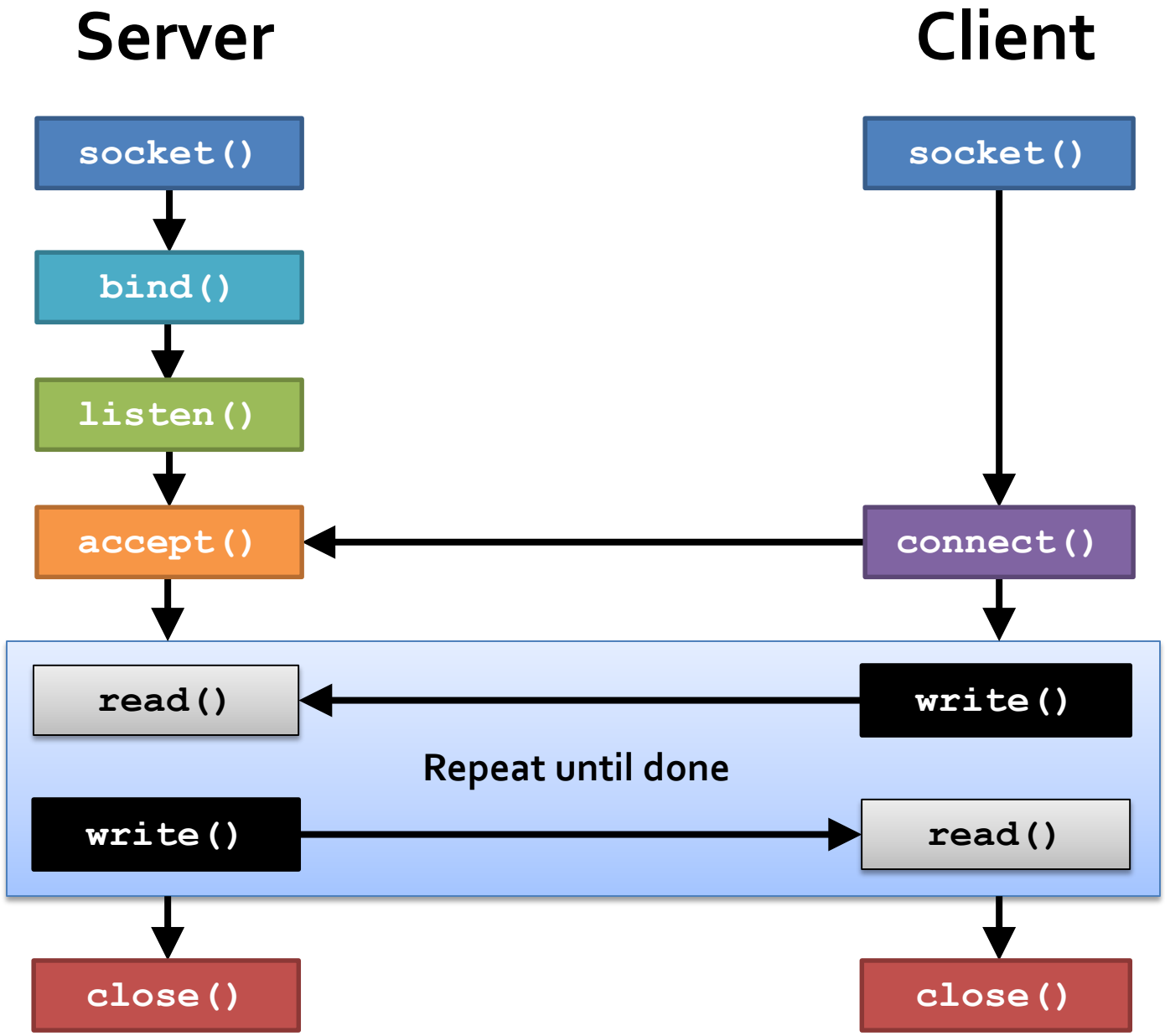
Server side: accepting

- For TCP connections, after listening, the server can call **`accept()`**

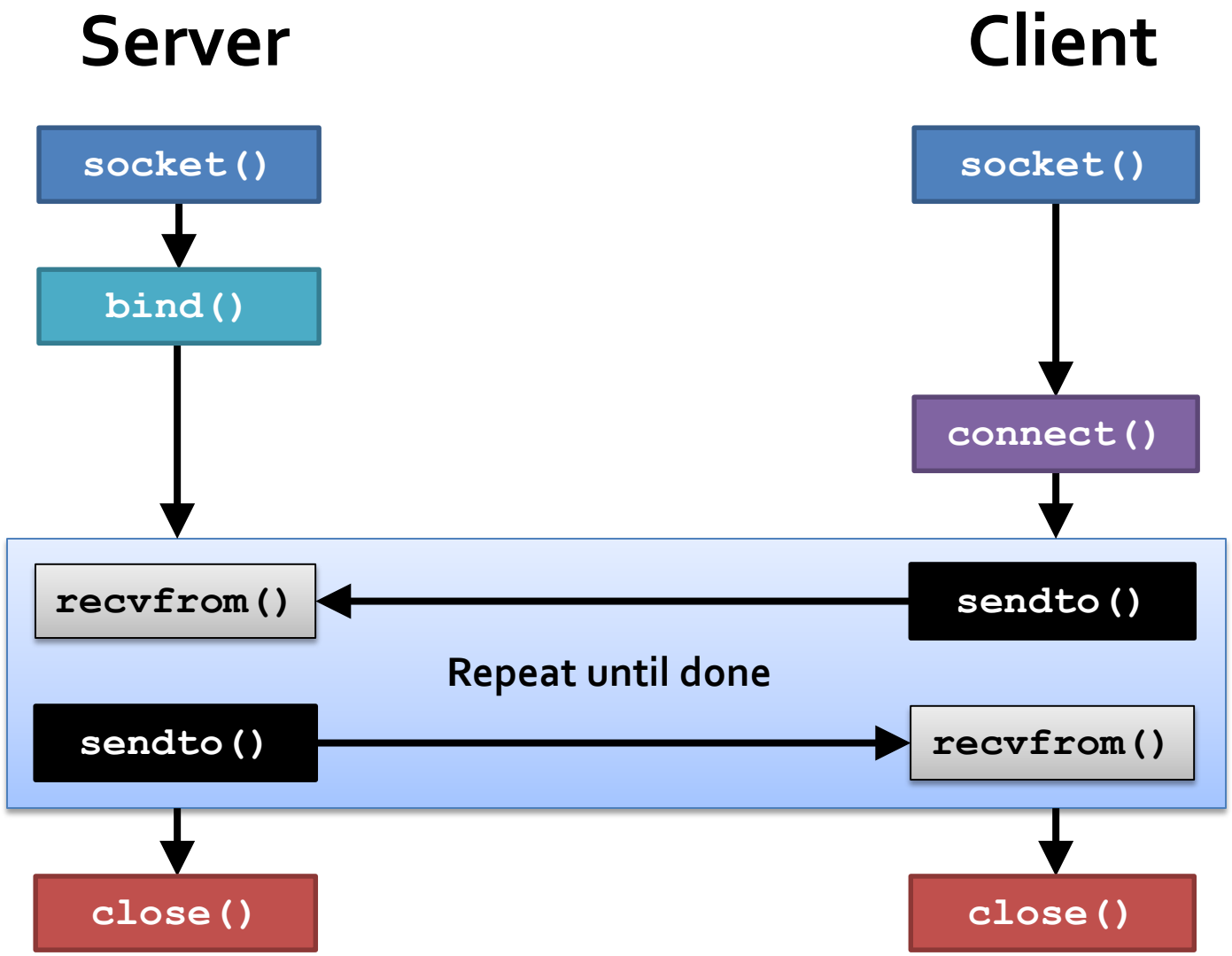
```
int accept (int socket, struct sockaddr *address, socklen_t *address_len);
```

  - Blocking function
  - Will wait until a client tries to connect
  - Then, messages can be sent and received
  - Doing so sets up a TCP session, expecting a series of packets from the connecting client

TCP Communication



UDP Communication



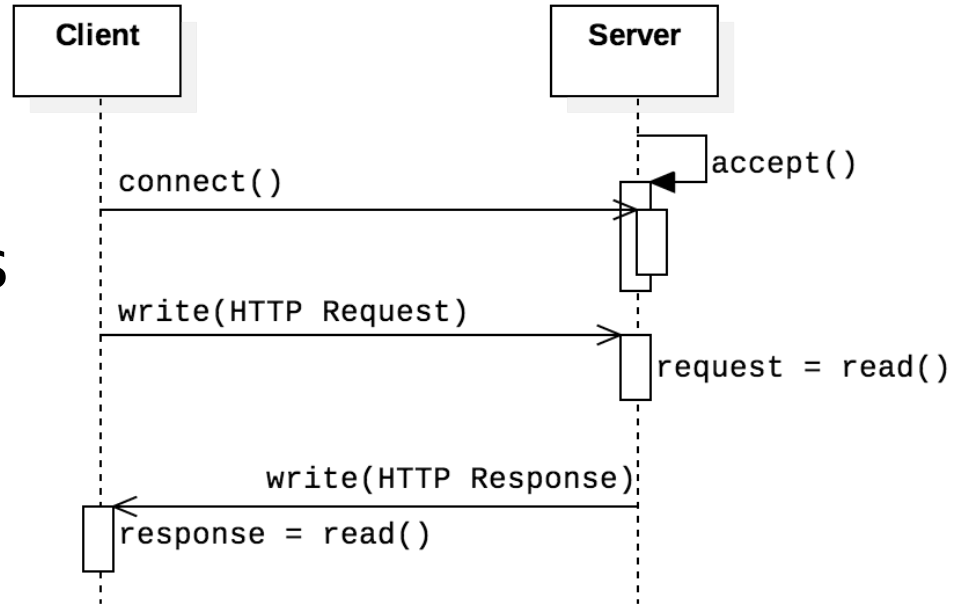
TCP Socket Programming

TCP communication

- The biggest differences between single-machine and networked IPC:
  - Networked IPC typically employs **protocols** so that machines agree on how data should be formatted
  - Networked IPC is less reliable
- It's hard to talk about TCP communication without examples that use some particular application layer protocol
- We're going to use HTTP because:
  - It's easy to understand
  - It's really important
  - There are lots of servers in the world we can talk to without any credentials

HTTP

- **Hypertext Transfer Protocol** (HTTP) is the protocol for (non-encrypted) web page communication
- It's a request-response protocol
  - Shown in the sequence diagram on the right
- HTTP itself is stateless: no information is preserved between requests
- Other features built around HTTP (cookies, server-side scripting, and databases) overcome this stateless limitation



Sample request

- HTTP requests and responses start with header lines
  - Each ends with CRLF (**\r\n**), with an extra CRLF after all headers
  - Each **\r\n** would simply look like a newline, but we show them below for clarity
- The most common client request is GET
- It must have a line like the following:

```
GET /path HTTP/version\r\n
```

  - **path** is the file being requested
  - **version** is the HTTP version, usually 1.0, 1.1, or 2

```
GET /index.html HTTP/1.0\r\n
Accept: text/html\r\n
Accept-Encoding: gzip, deflate, br\r\n
Accept-Language: en-US,en;q=0.5\r\n
User-Agent: Mozilla/5.0\r\n
\r\n
```

Sample response

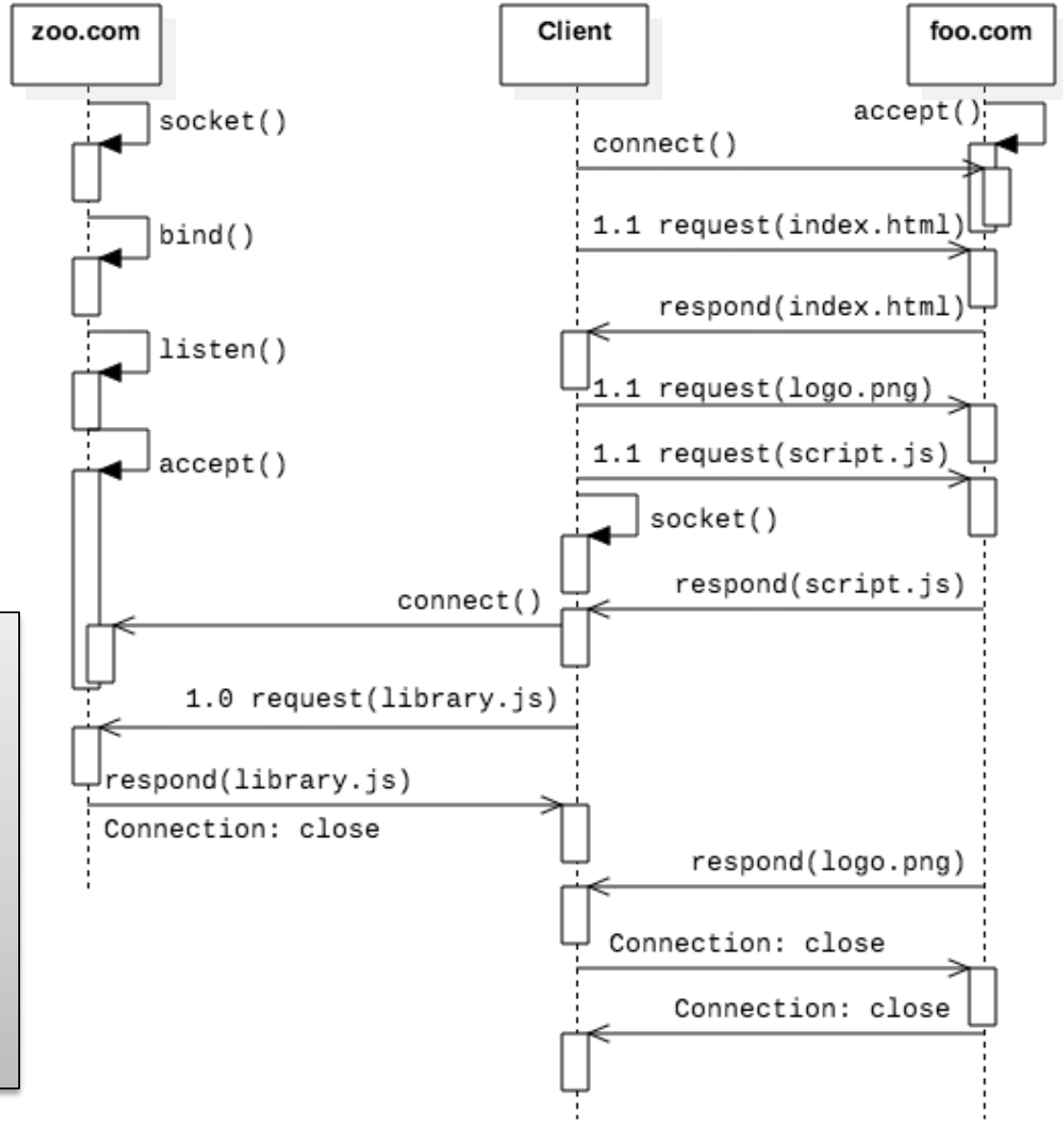
- After sending that data, the response will look something like the following:

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html; charset=UTF-8\r\n
Date: Sun, 28 Feb 2021 22:20:28 GMT\r\n
Content-Length: 1256\r\n
Connection: close\r\n
\r\n
<!doctype html>\n
<html>\n
<head>\n
    <title>Example Domain</title>\n
</head>\n
<body>\n
<div>\n
    <h1>Example Domain</h1>\n
    <p>This domain is for use in illustrative examples in documents.</p>\n
</div>\n
</body>\n
</html>\n
```

Persistent connections

- HTTP/1.0 was one and done
- HTTP/1.1 allows for persistent connections, so that multiple requests can be made over the same TCP connection
- HTML that requires multiple requests is below
- The sequence diagram showing the communication is on the right

```
<html>
<head>
<script src="http://zoo.com/library.js" />
<script src="script.js" />
</head>
<body><img src="logo.png" /></body>
</html>
```



Processing headers

- After reading headers we can look through each one
- One critical thing is to find the length of the content, so we can allocate enough space for it

```
char *line = buffer;
char *eol = strstr (line, "\r\n");
size_t body_length = 0;
while (eol != NULL)  // While there are more CRLFs
  {
    eol[0] = '\0';   // Null-terminate each line
    printf ("HEADER LINE: %s\n", line);

    // Find content length
    if (! strncmp (line, "Content-Length: ", 16))
      {
        char *len = strchr (line, ' ') + 1;
        content_length = strtol (len, NULL, 10);
      }

    line = eol + 2; // Move to the next line
    eol = strstr (line, "\r\n");
  }
```

Getting the content

- On the previous slide, we found the length of the content
- It's possible that the content was so small we read it into our 8 KB buffer
- Otherwise, we'll need to allocate more space

```
int length = strlen(eoh + 4);
char *content = malloc(length + 1);
strcpy(content, eoh + 4);
if (content_length > length) // if false, all data received
  {
    // Increase the content size and read additional data
    // Bytes needed is the Content-Length minus bytes already received
    content = realloc (content, content_length);
    bytes = read (socketfd, content + length, content_length - length);
  }
```

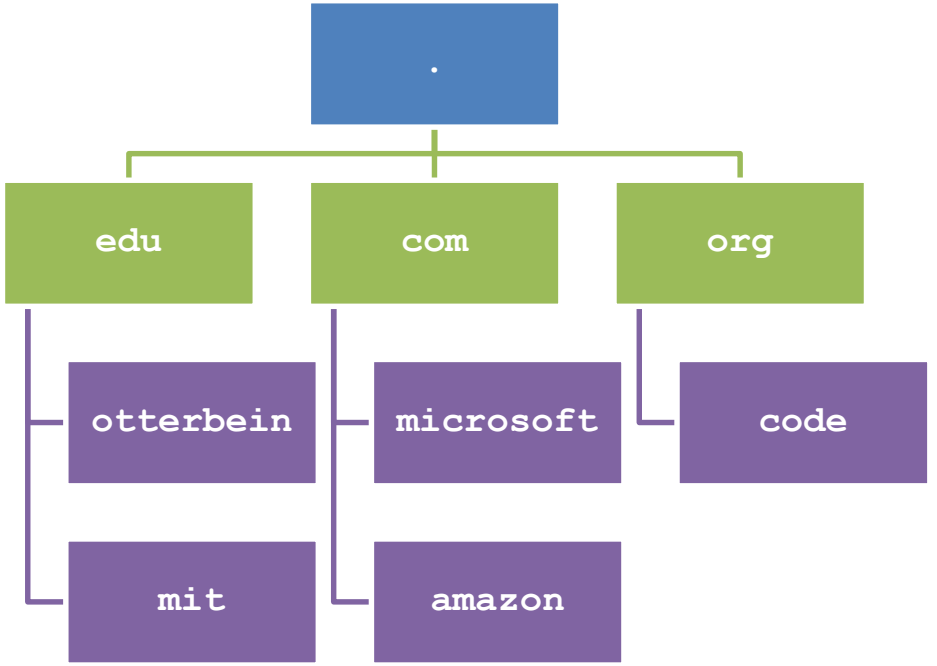
UDP Socket Programming

UDP socket programming

- As with TCP, it's hard to give meaningful examples of code without using some application-level protocol
- For TCP, we did HTTP
- For UDP, we'll do DNS
- **DNS**, the **Domain Name System**, is the distributed network of servers that translates domain names into IP addresses

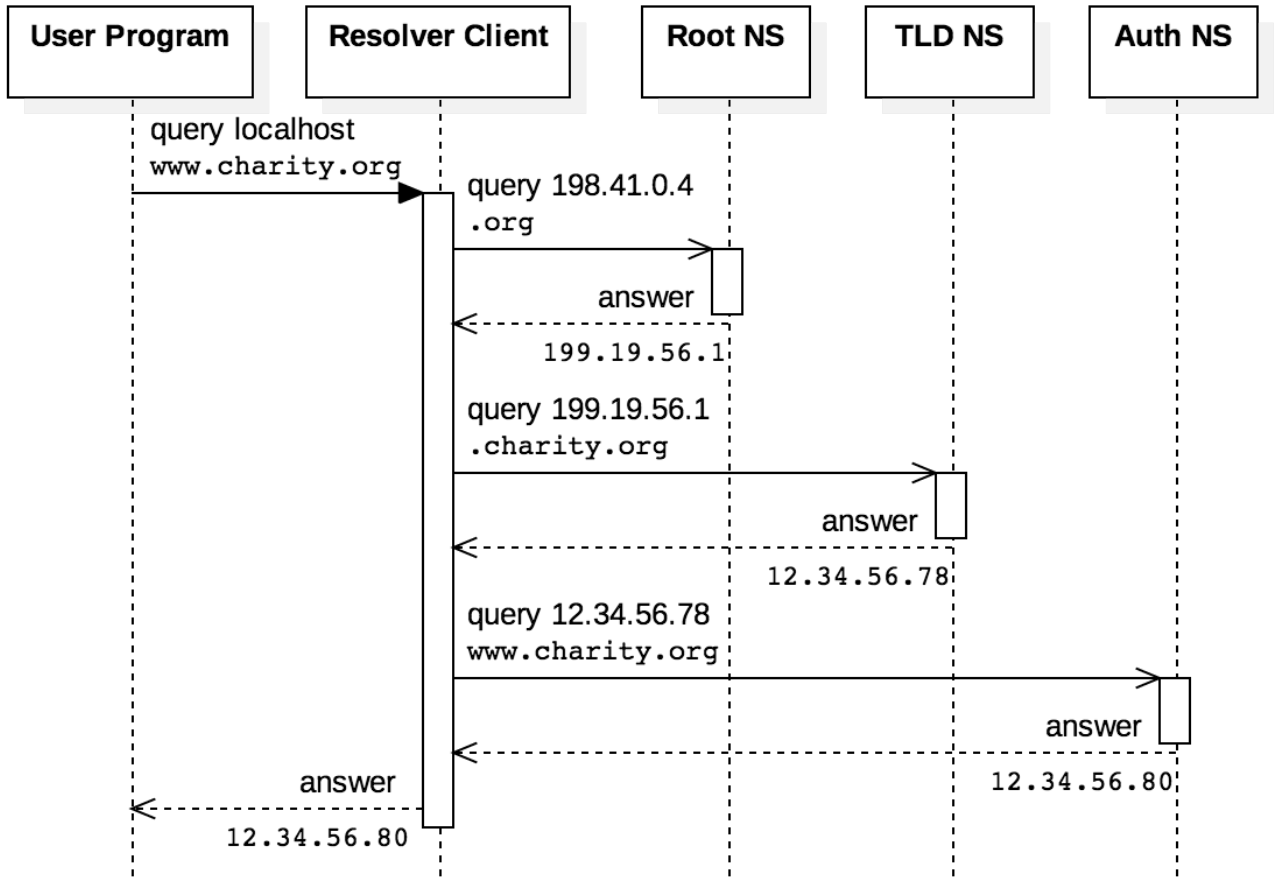
DNS

- ICANN maintains the root structure of DNS
- Below the root are top level domains (TLDs) like **`com`**, **`edu`**, **`org`**, **`net`** and a lot of weird newish ones like **`engineering`** and **`pink`**
- Different companies manage each TLD
- Domains can be looked up from the TLD that houses it
  - **`edu`** knows where **`otterbein.edu`** can be found
  - Dots separate each entity
- It's a kind of little endian ordering where the leftmost entity is the most specific, growing more general to the right
- DNS is case insensitive



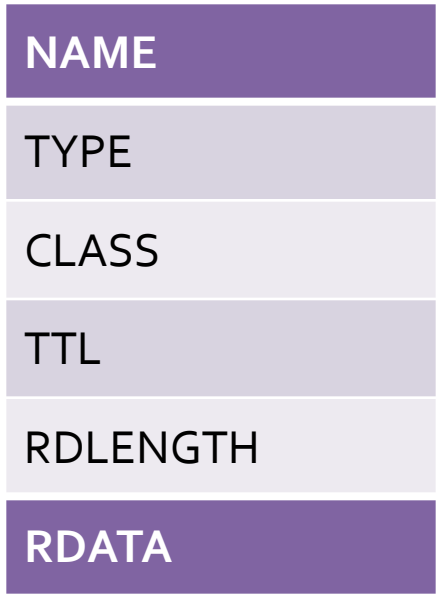
DNS queries

- Queries can be iterative:
  - Ask the root, get a response for the TLD
  - Ask the TLD for the domain you want
  - Get a response closer to what you're looking for and repeat
  - Shown on the right
- Queries can also be recursive:
  - Ask a name server, it handles everything
- To make the system efficient, servers cache domains that have been asked for recently
- There's a time-to-live value that says how long a cached domain should be kept



DNS resource record structure

- DNS information is sent in resource records, which have the following form:
  - NAME is the human-readable domain name
  - TYPE is gives the kind of record
    - A is an IP address
    - CNAME is a canonical name
    - NS is an authoritative name server
  - CLASS is what protocol, often IN for Internet
  - TTL is time-to-live in a cache
  - RDLENGTH is the length of the data in the record
  - RDATA is the data
- NAME and RDATA are variable length, and all other fields are 16 bits



DNS requests

- Like HTTP, DNS is a request-response protocol
- Unlike HTTP, DNS uses UDP and messages aren't as human readable
- DNS messages contain five fields: header, question, answer, authority, and additional
  - Headers start with a random ID to keep messages straight
- Example request to resolve **`example.com`**:

Field	Data in Hex	Meaning
Header	`1234`	`XID=0x1234`
	`0100`	`OPCODE=SQUERY`
	`0001 0000 0000 0000`	1 question field
Question	`0765 7861 6d70 6c65 0363 6f6d 00`	`QNAME=EXAMPLE.COM`
	`0001 0001`	`QCLASS=IN, QTYPE=A`
Answer		
Authority		
Additional		

Note:

Instead of dots, **`QNAME`** gives the number of characters for each name part

Character	7	e	x	a	m	p	l	e	3	c	o	m	0
Hex	07	65	78	61	6d	70	6c	65	03	63	6f	6d	00

DNS responses

- Here's a reasonable response to the request from the previous slide
- Don't worry about the OPCODE, it's a set of bits laid out according to DNS rules
- QNAME uses a special code to indicate that the name is 12 bytes into this response (to avoid repetition)

Field	Data in Hex	Meaning
Header	1234	XID=0x1234
	8180	OPCODE=SQUERY, RESPONSE, RA
	0001 0001 0000 0000	1 question and 1 answer
Question	0765 7861 6d70 6c65 0363 6f6d 00	QNAME=EXAMPLE.COM
	0001 0001	QCLASS=IN, QTYPE=A
Answer	c00c	QNAME=EXAMPLE.COM [compressed]
	0001	QTYPE=A
	0001	QCLASS=IN
	0000 e949	TTL = 0xe949 = 59721
	04	RDLENGTH = 4
	0x5db8d822 [93.184.216.34]	RDATA
Authority		
Additional		

Broadcasting

Static and dynamic IP addresses

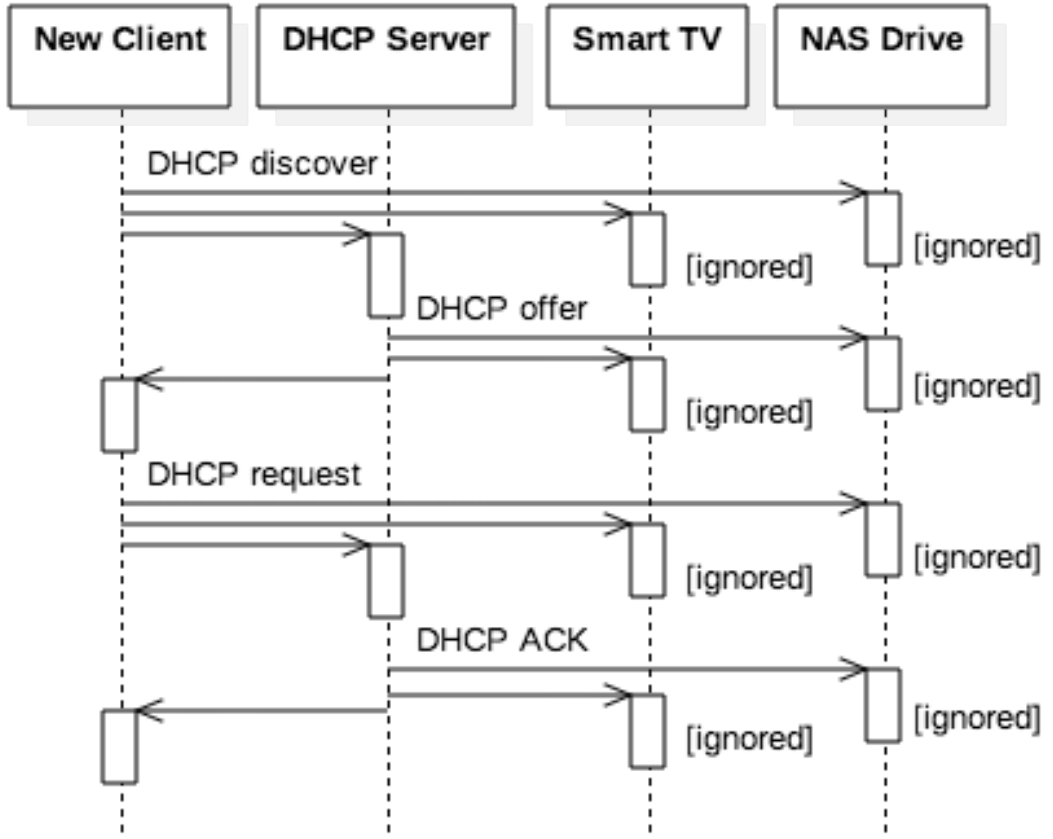
- Networked devices are configured to have either a static or dynamic IP address
  - **Static IP addresses** are set in a configuration file and rarely change
  - **Dynamic IP addresses** are assigned when a device connects to a network
- Most of the devices you own have dynamic IP addresses
  - Your laptop or phone is dynamically assigned an IP address by your router when you connect, either on WiFi or Ethernet
  - Even your router is dynamically assigned an IP address by your ISP
- Servers usually have static IP addresses so that DNS records don't need constant updates

DHCP

- But if you've just connected your device to the network, how does it get its IP address?
- **Dynamic Host Configuration Protocol** (**DHCP**) is a protocol for getting a dynamic IP address
- The socket programming we've been talking about requires an IP address for a client
  - That's where messages are sent back to
- So how do you receive a message if you don't have an IP address…because you're trying to get an IP address?
- **Broadcasting!**

Broadcasting

- Instead of sending a message point to point, DHCP messages are broadcast via UDP on port 67
  - The destination is the special IP address 255.255.255.255 reserved for broadcasting
  - The source is the special IP address 0.0.0.0 indicating no valid IP
- A DHCP server receives messages and responds with an IP address
- Other devices ignore the messages



DHCP steps

- DHCP is more complex than HTTP or DNS since it's got multiple steps
  1. The new device broadcasts a DHCP discover message asking for an IP
  2. The DHCP server broadcasts a DHCP offer message offering an IP
  3. The new device broadcasts a DHCP request message asking to take the IP it was offered
  4. The DHCP server broadcasts a DHCP ACK message acknowledging that the device has been assigned the address in question
- Like DNS, the device uses a random **`xid`** to keep different requests straight
- When the device requests the IP it's been offered, it increments the **`xid`** by `1`

DHCP example

- The table shows an example of the addresses and messages broadcast to request and assign an IP address
  - **yiaddr** is the new IP address
  - **siaddr** is the server IP address
  - The lease time is how long the IP address is valid for, in seconds: 86,400 = 24 hours
- When the lease expires, the device can ask for the IP again

Message type	UDP addressing	DHCP contents
DHCP discover	SRC: 0.0.0.0:68 DEST: 255.255.255.255:67	op: BOOTREQUEST xid: 42 yiaddr: 0.0.0.0
DHCP offer	SRC: 192.168.1.1:67 DEST: 255.255.255.255:68	op: BOOTREPLY xid: 42 yiaddr: 192.168.1.7 siaddr: 192.168.1.1 lease time: 86400
DHCP request	SRC: 0.0.0.0:68 DEST: 255.255.255.255:67	op: BOOTREQUEST xid: 43 yiaddr: 192.168.1.7 siaddr: 192.168.1.1
DHCP ACK	SRC: 192.168.1.1:67 DEST: 255.255.255.255:68	op: BOOTREPLY xid: 43 yiaddr: 192.168.1.7 siaddr: 192.168.1.1 lease time: 86400

Application Layer

Peer-to-peer applications

- We have already given examples of client-server applications
  - HTTP
  - DNS
- Although some client-server interactions are much more complicated, many of the same principles will apply
- **Peer-to-peer applications** (**P2P**) are the other major, application-layer approach
  - Every host is potentially a client and a server
  - Communicating with peers is more complex because there isn't a single server to keep track of
- In many situations, P2P applications can provide better performance than client-server when the number of hosts is large

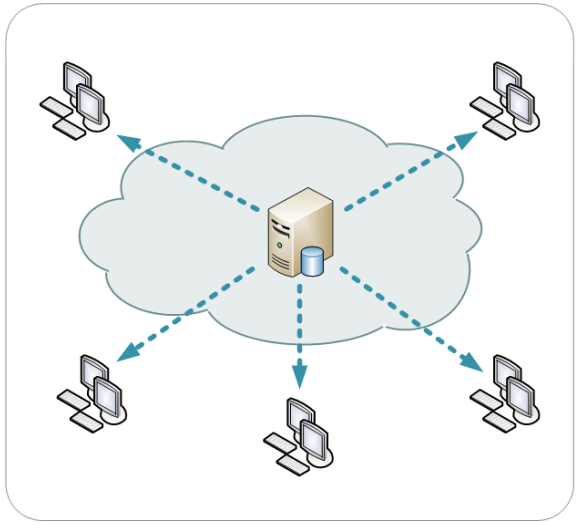
P2P examples

- P2P has a historic association with illegal file sharing, but P2P architectures are used for many different kinds of applications

Application	Service Description	Examples
Content Distribution	Scalable approaches to sharing data across the Internet	▪ **File storage and sharing:** Gnutella, BitTorrent, InterPlanetary File System (IPFS) ▪ **Content delivery networks (CDNs):** Akamai, Limelight ▪ **Streaming media:** Spotify (originally), Sonos ▪ **Software update distribution:** Linux, World of Warcraft
Distributed Computing	Delegating work for an application across many computers	▪ **Privacy and censorship resistance:** Tor, Freenet ▪ **Cryptocurrency:** Bitcoin ▪ **Botnets and malware:** Storm
Collaboration	Providing real-time human communication	▪ **Voice Over IP (VOIP):** Skype (originally) ▪ **Instant Messaging:** Tox
Platforms	Building applications	▪ **Java:** JXTA (obsolete)

Content delivery networks

- Normally, there's a single server for a webpage
- What if that webpage has content that millions of people from all over the country want to view?
  - The load on the server will be huge
  - Getting the webpage will be slow, or the server could crash
- Content delivery networks (CDNs) provide caches of webpages
- People trying to view a webpage will be redirected to a physically close mirror
- Big companies like Google, Amazon, and Netflix have their own CDN services
- Less well-known companies like Akamai provide CDN services to others



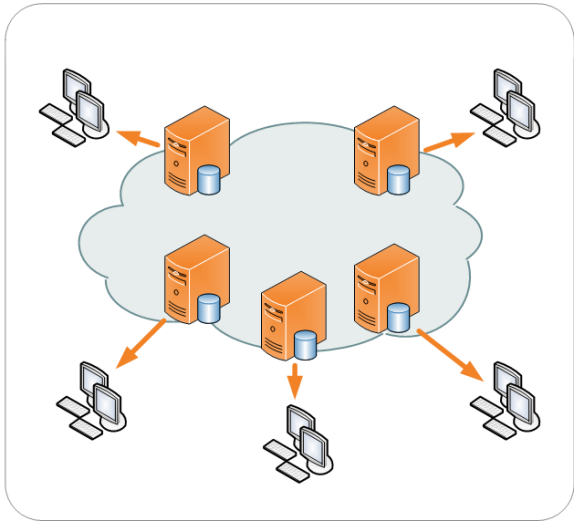
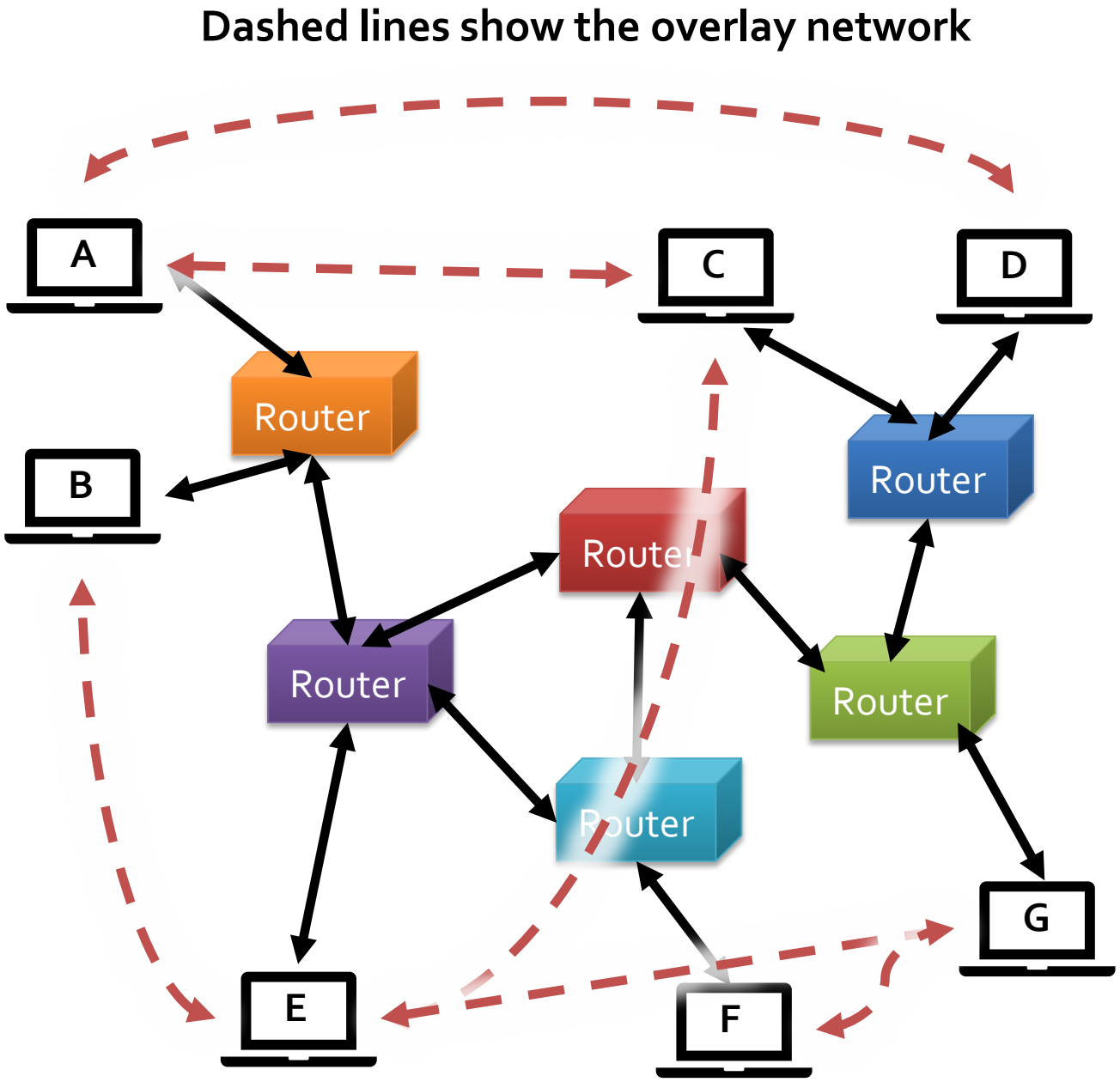


Image from Wikipedia

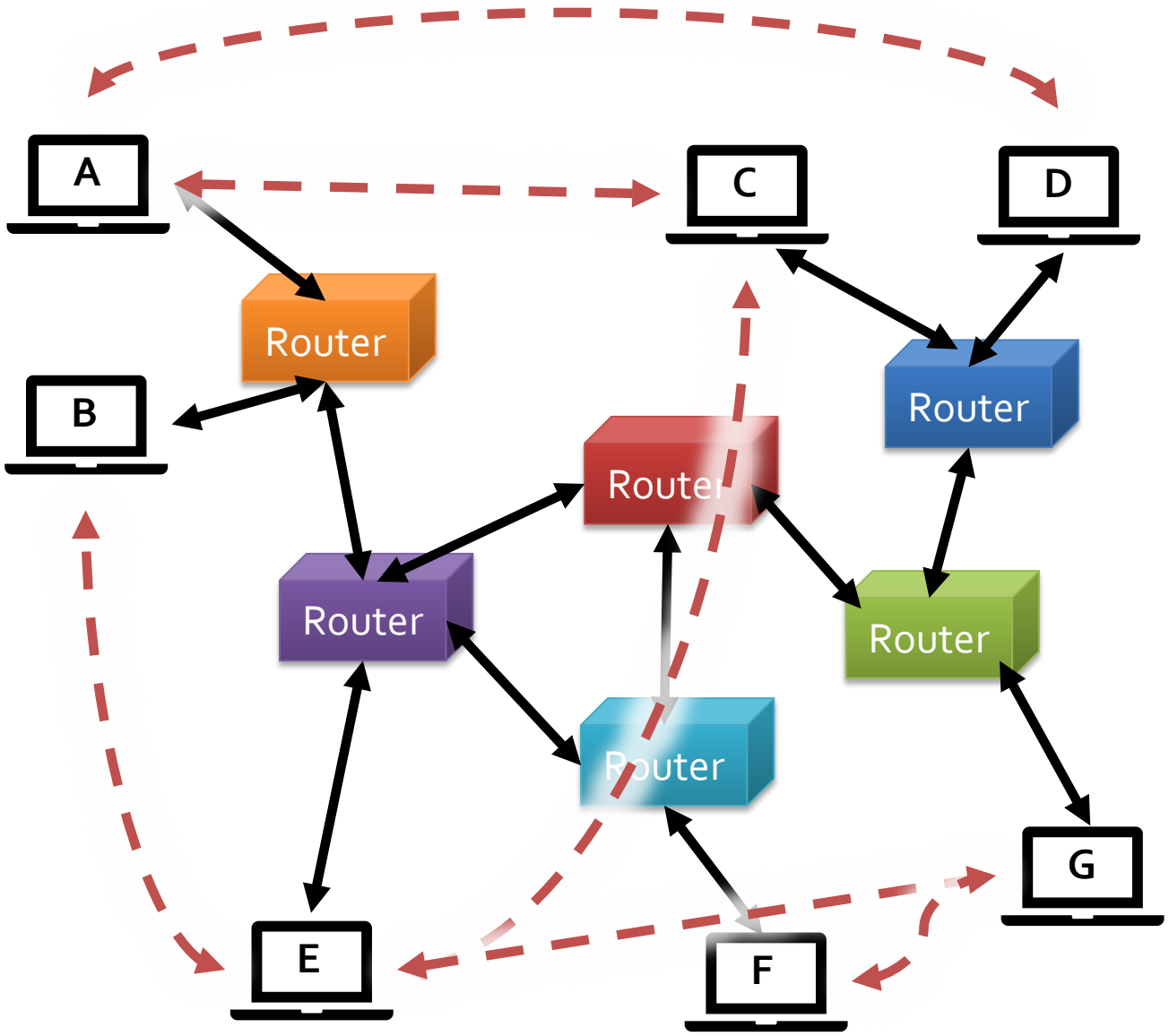
Overlay networks

- P2P hosts are fundamentally connected by the Internet
- However, they view their connections inside the P2P network as an **overlay network**, connections to other P2P hosts
- Thus, socket connections are made to P2P neighbors who forward messages on to other neighbors



Overlay networks

- For A to send a message to B, it has to send it to C, which sends it to E which sends it to B
  - Even though A and B are on the same network!
- While this arrangement seems inefficient, it can be used in applications like Tor, where the goal is to hide the true addresses of the hosts



Characteristics of P2P networks

- **Design decision:** Should a P2P network be structured or unstructured?
- Early P2P networks were often unstructured
  - This approach made sense for illegal file sharing
  - Unstructured networks have a lot of **churn**, hosts arriving and leaving frequently
- Many P2P networks are now structured with a logical framework
  - Maybe the overlay network arranges nodes in a circle, with each node knowing about the node before it and after it
  - A CDN might have an organization based on physical proximity

More characteristics of P2P networks

- **Design decision:** How are objects like files identified in the network?
- Unstructured networks often use **query flooding**
  - Ask all your neighbors if they have a file
  - If they don't have it, they'll ask their neighbors, and so on
- Structured networks have more options
  - Indexing objects based on location
  - Local indexes only know about neighbors
  - Depending on the structure, algorithms can be used to search the network
  - Centralized indexes are simple but put strain on a central server
  - Other approaches distribute the index across several servers

Transport Layer

Transport layer

- The transport layer provides a logical structure for end-to-end communication between two different (networked) processes
- Although there are other protocols for the transport layer, the most common ones are flavors of UDP and TCP
- As we have pointed out in the past:
  - TCP provides reliable transport that tries to fix failures
  - UDP is faster but unreliable

UDP

- The **User Datagram Protocol** (**UDP**) provides a bare-bones approach to sending messages
- Information included in a UDP segment is:
  - Source port
  - Destination port
  - Length of the segment
  - Checksum
  - Payload (actual data)
- Each header field is 16 bits, making a header of 8 bytes for each UDP segment in addition to the data

Checksum

- UDP uses a checksum to make sure that the segment isn't corrupted during transmission
- It is possible (but unlikely) that a message with some bits flipped will have the same checksum as the original
- Pseudo-code:
  - Add up all the 16-bit quantities in the message into a 32-bit sum
  - While adding, if the most significant bit of the sum is 1, change the sum to be the sum of its lower and upper halves
  - If there was an odd number of bytes, add the last byte padded with zeroes
  - After the sum is made, add the lower and upper halves of it to get a 16-bit value
  - Return the bitwise negation of the result

Example UDP segments

- The following are UDP segments for a DNS request and response

Header	1388 0035 0025 f693	source port = 5000 **(0x1388)** destination port = 53 **(0x0035)** length = 37 **(0x0025)** checksum
Payload	1234 0100 0001 0000 0000 0000 0765 7861 6d70 6d64 0363 6f6d 0000 0100 01	DNS request for **example.com**

Header	0035 1388 0035 af04	source port = 53 **(0x0035)** destination port = 5000 **(0x1388)** length = 53 **(0x0035)** checksum
Payload	1234 8180 0001 0001 0000 0000 0765 7861 6d70 6d64 0363 6f6d 0000 0100 01c0 0c00 0100 0100 00e9 4900 045d b8d8 22	DNS response for **example.com**

Unreliability

- Once the UDP segment arrives, the receiver can compute a checksum for the segment to see if it matches the one provided
- UDP itself doesn't do anything with this checksum value, but the applications built on UDP can decide to request the data again or ignore the bad data
- This lack of reliability seems like a problem, but it can be useful for streaming movies or audio
- It's also useful for DNS and DHCP, which are not usually visible to the user

TCP

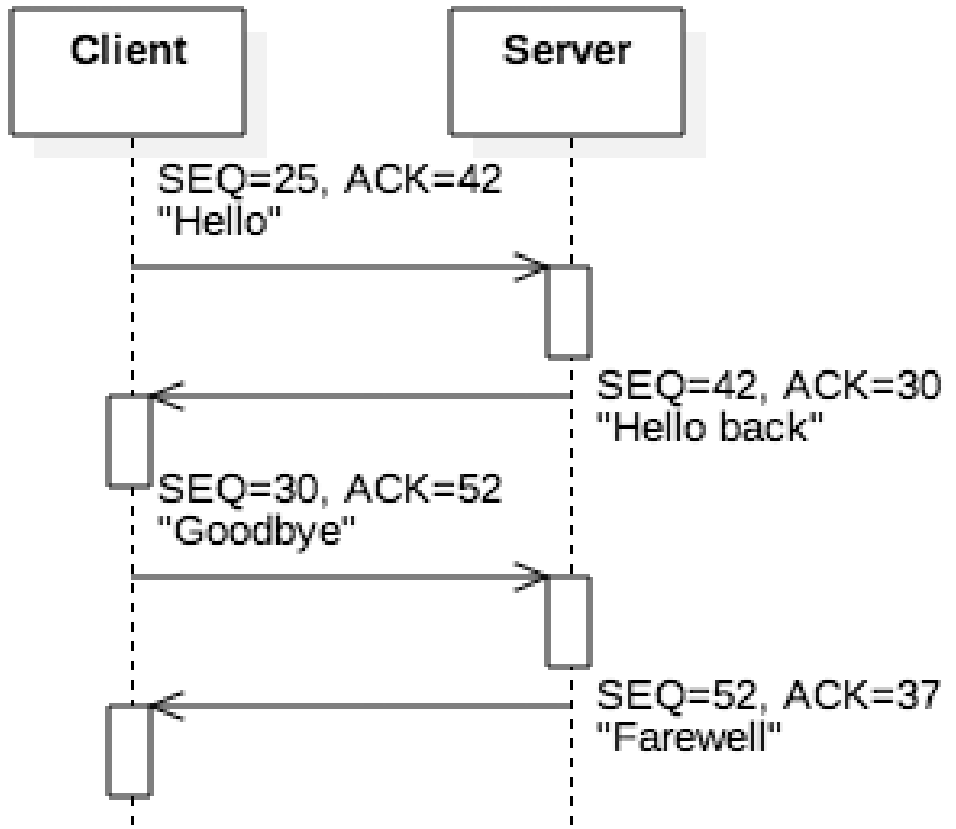
- Reliable transport is often desirable, so **Transmission Control Protocol** (**TCP**) is usually used for that purpose
- Unlike UDP, TCP creates a **session** with multiple messages sent back and forth between the two hosts
- Messages are numbered
- TCP also uses flow control, allowing hosts to avoid sending more data at once than their receivers can handle

TCP segments

- Because they have to do more, TCP segments contain more information:
  - Source port
  - Destination port
  - Sequence number (SEQ)
  - Acknowledgement number (ACK)
  - Flags
  - Receive window
  - Checksum
  - Urgent data pointer
  - Optional fields
  - Payload (actual data)
- Like UDP, most of these fields are 16 bits
  - SEQ and ACK are 32 bits
  - Optional fields vary
  - Payload is however long it needs to be

Numbering

- So that segments aren't lost, hosts send a sequence number (SEQ) with each segment
- The initial value is a random number ***k***
- After sending ***n*** bytes, the next SEQ will be ***n*** + ***k***
- So that the A knows how much B has gotten, B's next response to A contains an acknowledgement number (ACK) which is the last SEQ from A plus the size of that message
- In this way, both sides know how much the other side is sending, what's lost, and what's received
- If nothing is lost and messages are going back and forth, each SEQ will be the last ACK received



Flow control

- Buffers are always finite
- A TCP connection has a buffer that's reading information as it arrives from the other host
- Data is removed from this buffer as the process reads it from the socket
- If too much data is arriving, the buffer fills up, and data will be lost
- Each time a process sends a TCP segment, it also sends a **receive window** value, giving the number of bytes available in the buffer for that connection
- If there's not enough space for the next message, the sender will break its message into parts so that the part it sends will fit into the receive window

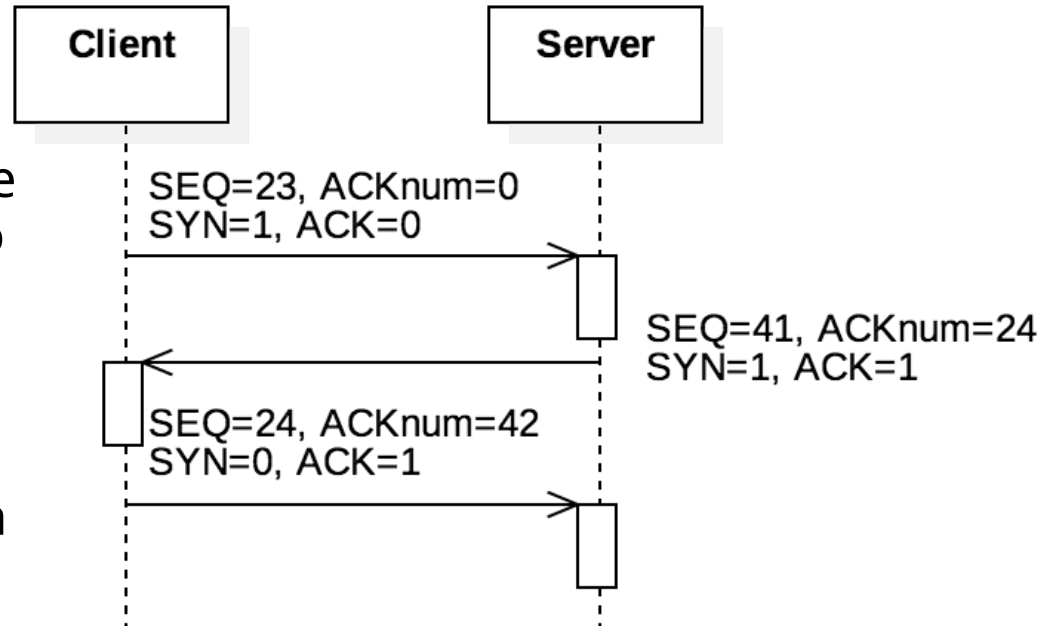
Example TCP segment

- The following is a TCP segment for an HTTP GET request

Header	1388 0050 0000 0017 0000 002a 5010 1000 cf33 0000	source port = 5000 (0x1388) destination port = 80 (0x0050) sequence number = 23 (0x17) acknowledgement number = 42 (0x2a) flags receive window = 4096 (0x1000) checksum urgent data ptr
Payload	4745 5420 2f20 4854 5450 2f31 2e31 0d0a 486f 7374 3a20 6578 616d 706c 652e 636f 6d0d 0a43 6f6e 6e65 6374 696f 6e3a 2063 6c6f 7365 0d0a 0d0a	GET / HTTP/1 .1\r\nHost: ex ample.com\r\nC onnection: c lose\r\n\r\n

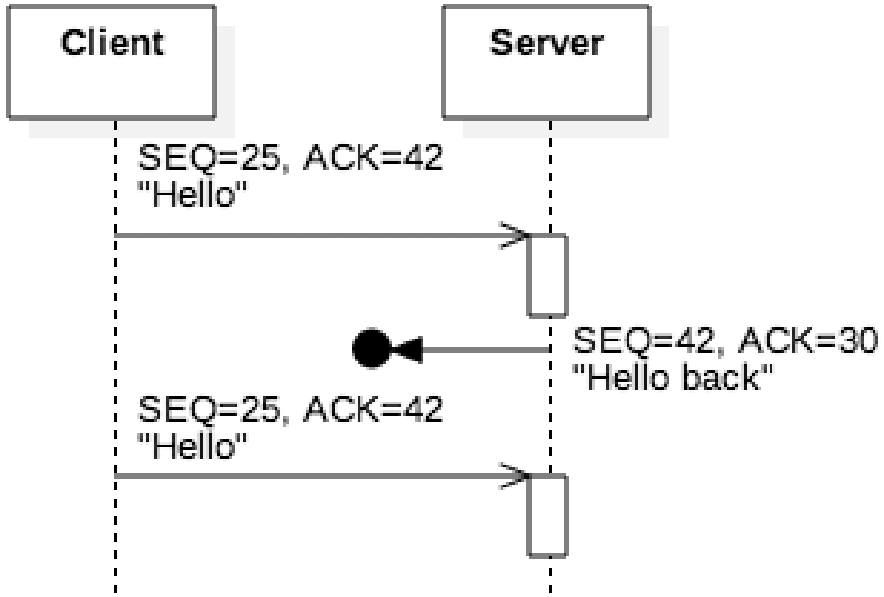
TCP handshake

- When a TCP connection is being established by a client calling **`connect()`**, three segments are sent:
  - SYN (from the client)
  - SYN-ACK (from the server)
  - ACK (from the client)
- These segments are called the three-way handshake
- They are normal segments except that they have no data
- The SYN bit is set on the SYN and SYN-ACK segments, and the ACK bit is set on the ACK segment
- ACK is set on any segment intended to show that an earlier segment is being acknowledged



Packet loss

- Using SEQ and ACK numbers with the checksum allows for error detection
- It's hard to be sure what went wrong, but some conclusions can be drawn:
  - Incorrect ACK: If the ACK is too small, the sender of the ACK missed one or more messages
  - Incorrect SEQ: If the SEQ is larger than expected, the receiver of the SEQ missed one or more messages
  - Incorrect checksum: The segment is corrupted or part is missing
- In all three cases, sending the last segment based on acknowledged data is a request for the other side to resend

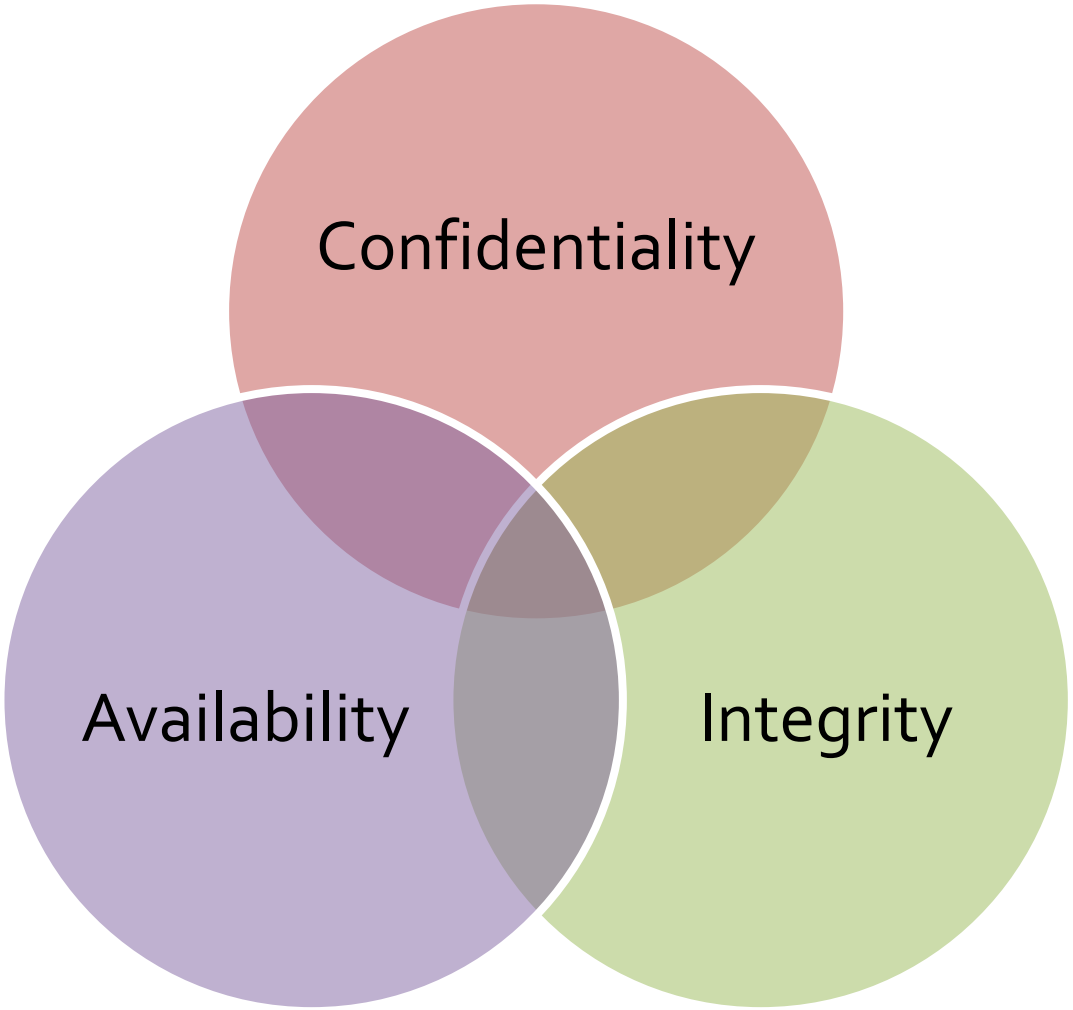


Timeouts

- To make robust guarantees about message delivery, TCP also keeps track of the time it takes for segments to make a trip
- If a segment is missing for long enough, TCP can request it again
- How long should it wait?
- Because the Internet is a large and heterogeneous place, it wouldn't make sense to wait for any particular fixed time
- Instead, the retransmission timeout (RTO) is computed based on previous transmission times and how much they fluctuate

Network Security

CIA



- Network security is built on principles from general computer security:
  - Confidentiality
  - Integrity
  - Availability

Confidentiality

- **You don't want other people to be able to read your stuff**
  - Some of your stuff, anyway
- Cryptography, the art of encoding information so that it is only readable by those knowing a secret (key or password), is a principle tool used here
- Confidentiality is also called **secrecy** or **privacy**

Integrity

- **You don't want people to change your stuff**
- You want to know:
  - That your important data cannot be easily changed
  - That outside data you consider trustworthy cannot be easily changed either
- There are many different ways that data can be messed up, and every application has different priorities

Availability

- **You want to be able to use your stuff**
- Many attacks are based on **denial of service**, simply stopping a system from functioning correctly
  - A SYN flood where attackers try constantly to create TCP connections from spoofed IP addresses is a classic DoS attack
- Availability can mean any of the following:
  - The service is present in usable form
  - There is enough capacity for authorized users
  - The service is making reasonable progress
  - The service completes in an acceptable period of time

Cryptography

- "Secret writing"
- The art of encoding a message so that its meaning is hidden
- **Cryptanalysis** is breaking those codes
- Cryptography is a powerful tool for *confidentiality* because modern encryption methods make it almost impossible to read an encrypted message
- Cryptographic hash functions provide a tool for *integrity* because they can make it obvious when a message has been changed

Cryptography and availability

- Although cryptography provides tools for confidentiality and integrity, there's no clear cryptographic tool for availability
- In fact, cryptography often makes availability **worse** because encryption puts more strain on a system
  - Making it more susceptible to various DoS attacks
- There's **always** tension between confidentiality, integrity, and availability
- Increasing confidentiality and integrity usually decreases availability

Encryption and decryption

- **Encryption** is the process of taking a message and encoding it
- **Decryption** is the process of decoding the code back into a message
- A **plaintext** is a message before encryption
- A **ciphertext** is the message in encrypted form
- A **key** is an extra piece of information used in the encryption process

Symmetric key cryptography

- Symmetric encryption is what you probably think of as encryption
- Two parties have a **key** which they use for both encrypting and decrypting messages
  - The key is also known as a **shared secret**
- We have excellent symmetric encryption algorithms, of which AES is the most used
- But how do we distribute keys between parties who want to communicate secretly?

AES

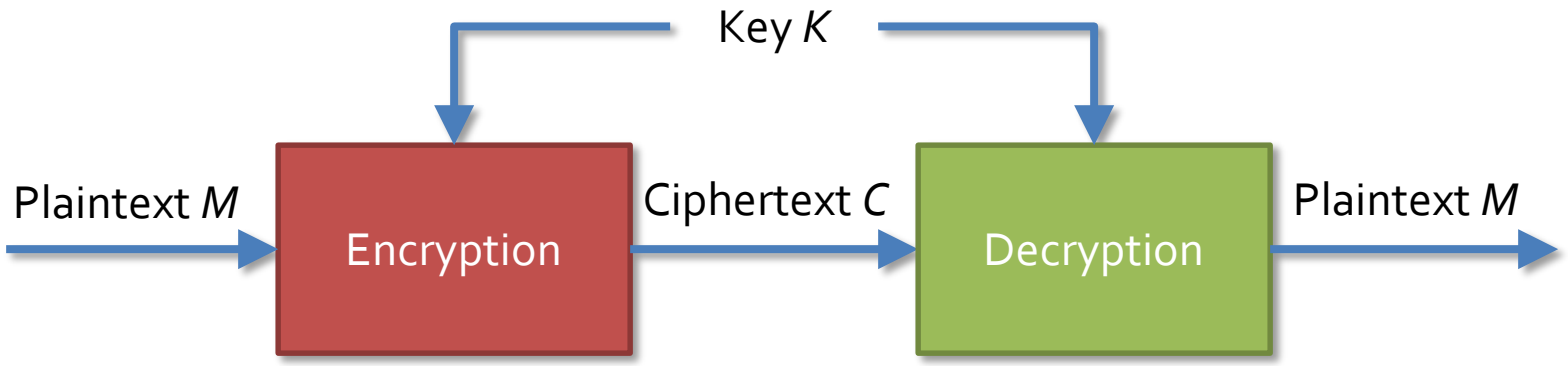
- **A**dvanced **E**ncryption **S**tandard
- Symmetric block cipher designed to replace DES
- Block size of 128-bits
- Key sizes of 128, 192, and 256 bits
- Like the older (and deprecated) DES, has a number of rounds (10, 12, or 14 depending on key size)
- Originally called Rijndael, after its Belgian inventors
- Competed with 14 other algorithms over a 5 year period before being selected by NIST
- No known attacks exist against good implementations of AES
  - It should take more than a billion billion years to break an AES encryption
  - Even quantum computers shouldn't change that much

Public key cryptography

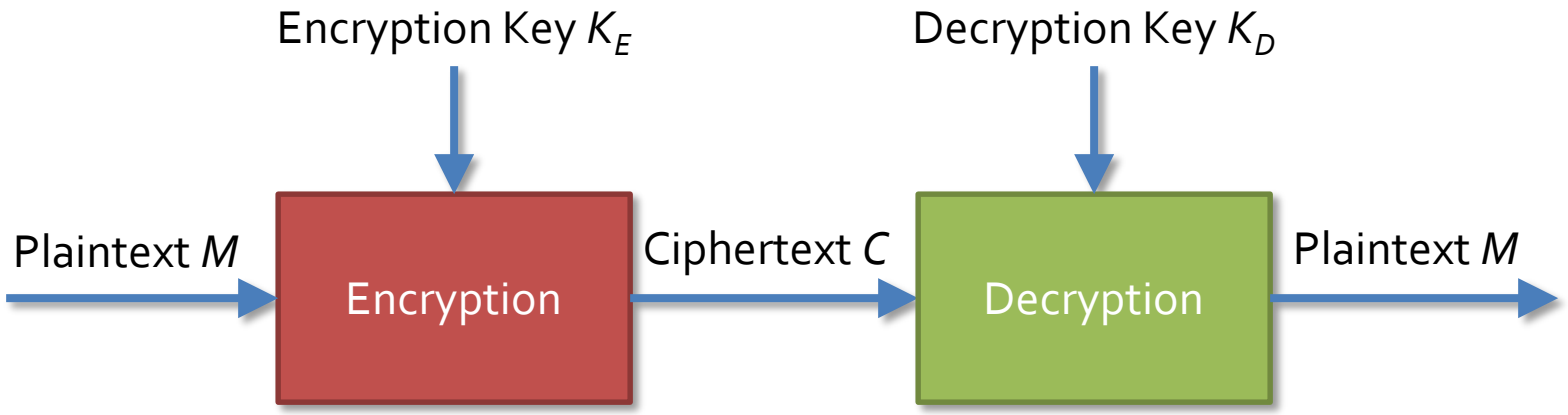
- Sometimes, we need something different
- We want a **public key** that anyone can use to encrypt a message to Alice
- Alice has a **private key** that can decrypt such a message
- The **public key** can only encrypt messages; it **cannot** be used to decrypt messages
- Public key cryptography is enormously useful, since companies can publish their public key far and wide
  - Anyone who wants to send them a secret message can do so
  - No secret needs to be shared ahead of time

Symmetric vs. public key

Symmetric Key Cryptography



Public Key Cryptography



RSA Algorithm

- RSA is the most commonly used public key cryptosystem
- Named for **R**ivest, **S**hamir, and **A**dleman
- Take a plaintext ***M*** converted to an integer

- Create an ciphertext ***C*** as follows:
  $C = M^e \bmod n$

- Decrypt ***C*** back into ***M*** as follows:
  $M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$

Why it's safe

- Crazy number theory
- For RSA, the modulus $n = p \cdot q$ where p and q are two large (hundreds of digits) primes
- It's easy to compute d, the decryption exponent, if you know p and q
- No one knows an efficient way to factor a large composite number
- However, quantum computers *could* make RSA much less safe

Catch-22

- Your computer needs to be able read the password file to check passwords
- But, even an administrator shouldn't be able to read everyone's passwords
- Hash functions to the rescue!

Definition

- A **cryptographic** (or one-way) hash function (also called a cryptographic checksum) takes a variable sized message ***M*** and produces a fixed-size hash code **H**(***M***)
- **Not the same as hash functions from data structures**
- The hash code produced is also called a **digest**
- It can be used to provide authentication of both the integrity and the sender of a message
- It allows us to store some information about a message that an attacker cannot use to recover the message

Collisions

- When two messages hash to the same value, this is called a **collision**
- Because of the pigeonhole principle, collisions are unavoidable
- The key feature we want from our hash functions is that collisions are difficult to predict

Crucial properties

Preimage Resistance

- Given a digest, should be hard to find a message that would produce it
- One-way property

Second Preimage Resistance

- Given a message m, it should be hard to find a different message that has the same digest

Collision Resistance

- Should be hard to find any two messages that hash to the same digest (collision)

Additional properties

Avalanching	• A small change in input should correspond to a large change in output
Applicability	• Hash function should work on a block of data of any size
Uniformity	• Output should be a fixed length
Speed	• It should be fast to compute a digest in software and hardware • No longer than retrieval from secondary storage

Password dilemma resolved

- Instead of storing the actual passwords, Windows and Unix machines store the hash of the passwords
- When someone logs on, the operating system hashes the password and compares it to the stored version
- No one gets to see your original password!
- Hash functions are also used for digital signatures

SHA family

- **Secure Hash Algorithm**
- Created by NIST
- SHA-0 was published in 1993, but it was replaced in 1995 by SHA-1
- The difference between the two is only a single bitwise rotation, but the NSA said it was important
- SHA-1 security
  - Digest size: 160 bits
  - Considered unsafe
  - Theoretical attacks can run in 2^{63} SHA-1 evaluations
- SHA-2 is a successor family of hash functions
  - 224, 256, 384, 512 bit digests
  - Now the preferred hashing function
  - Designed by the NSA

SHA-3

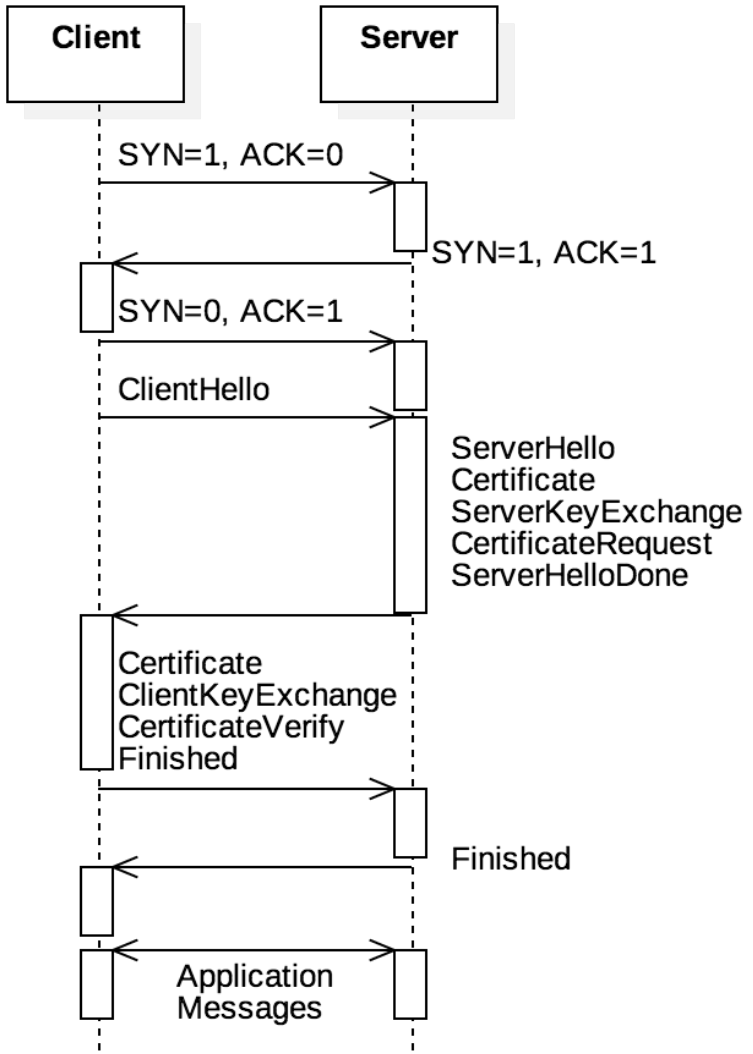
- SHA-3 (Keccak) uses a completely different form of hashing than SHA-0, SHA-1, and SHA-2
- Although the attacks on SHA-1 are expensive and there are no real attacks on SHA-2, the attacks on SHA-0 made people nervous about hash functions following the same design
- SHA-3 also allows for variable size digests, for added security
  - 224, 256, 384, and 512 are standard
- Either SHA-2 or SHA-3 is considered secure (for now)

Transport-Layer Security

- **Transport-Layer Security** (**TLS**) adds end-to-end security to TCP
  - Secure versions of protocols often add an "s" to their names: HTTPS, SFTP, and IMAPS
  - These protocols use TLS
- With TLS, the TCP data is encrypted
- However, the TCP headers are *not* encrypted
  - If they were, the OS wouldn't know which port to deliver them to
  - Because network traffic needs to know where to go, it's usually possible to do traffic analysis, even when the data is encrypted

TLS handshake

- With TLS, the connection first performs a TCP three-way handshake
- Then, the client and the server perform a TLS handshake that uses public key cryptography to agree on a session key
- The session key is used to communicate securely using symmetric key encryption (probably AES) during the TCP session



Confidentiality and integrity

- Because the data in the TCP segments is encrypted with AES, the information's confidentiality is maintained
- To protect integrity, a message authentication code (MAC) of the TCP headers is attached as an optional TCP field
  - The MAC is a cryptographic hash digest, probably using SHA-2
- These are the broad strokes, but there are many details
- Details change with each version of TLS
  - We're up to TLS 1.3 now

Internet Layer

Internet layer

- TCP and UDP provide a framework for end-to-end communication between *processes*
- But they ignore the fact that different *hosts* are communicating
- The Internet layer provides a system for getting messages from host to host
  - The **data plane** gives the structure of the network, using Internet Protocol (IP) addresses
  - The **control plane** controls how messages are routed through the network

IP addresses and subnets

- IPv4 addresses only allow for about 4 billion addresses
- One approach for dealing with this limitation is **subnets**
  - Private subnets have IP addresses that can't be reached from outside
  - Subnets sometimes use the notation of an IP address followed by **`/n`** where **`/n`** is some number of bits like **`/16`** or **`/24`**, meaning that addresses inside the subnet must match the first 16 or 24 bits
  - Example: **`192.168.0.0/16`** means everything on the subnet must start with **`192.168`**
  - The matching bits can also be specified as a subnet mask: **`255.255.0.0`** is the one matching the first 16 bits

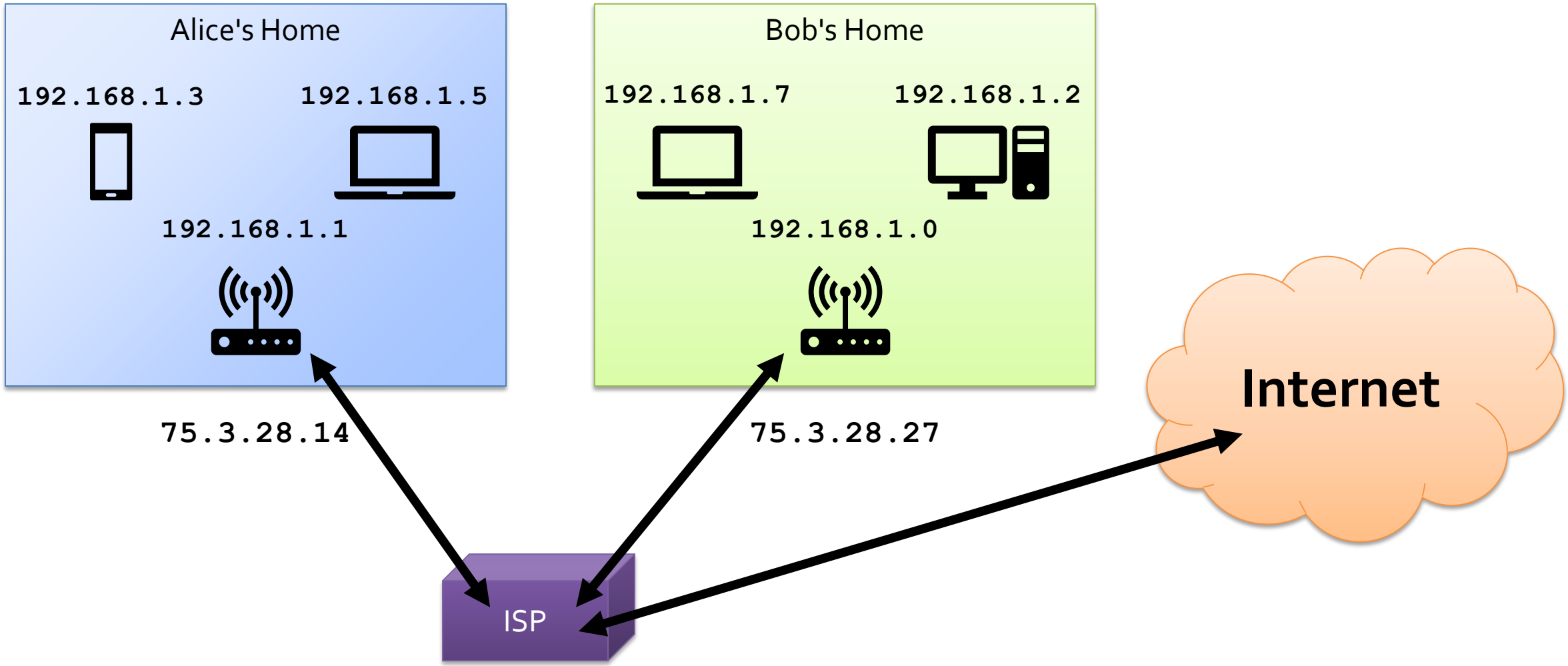
Special subnets

- Three ranges of IP addresses are reserved in IPv4 for private subnets:
  - **`192.168.0.0/16`** (2^{16} = 65,536 possible devices)
  - **`172.16.0.0/12`** (2^{20} = 1,048,576 possible devices)
  - **`10.0.0.0/8`** (2^{24} = 16,777,216 possible devices)
- The first range is probably familiar to you because it's used for most home networks
- IPv6 has its own range, **`fd00::/8`**, that allows for up to 2^{64} devices

NAT

- So that different subnets can communicate, a router connects the private subnet to the Internet
  - The router has a private IP address, used to communicate with the subnet, and a public IP address, used to communicate with the rest of the world
- Routers do **network address translation** (**NAT**), a kind of **IP masquerading**
  - The outside world only sees the router's IP
  - When the router gets a message, it sends it to the appropriate device in the private subnet
  - The router observes traffic and changes port numbers on incoming and outgoing packets so that multiple devices behind the router can communicate with a single server

Visualization of subnets



IPv4 packet format

- **`version`** distinguishes between IPv4 and IPv6
- **`protocol`** is TCP or UDP
- **`checksum`** is just for the header and does no checking for the payload
- **`TTL`** gives the number of times the packet can be forwarded (keeps packets from hopping around forever)
- Like UDP, IP makes no guarantees about reliability
- The purple **`options`** fields are variable length

<table>
<tr><th>0-3</th><th>4-7</th><th>8-11</th><th>12-15</th><th>16-19</th><th>20-23</th><th>24-27</th><th>28-31</th></tr>
<tr><td>version</td><td>length</td><td colspan="2">type of service</td><td colspan="4">total length</td></tr>
<tr><td colspan="4">identification</td><td>flags</td><td colspan="3">fragment offset</td></tr>
<tr><td colspan="2">TTL</td><td colspan="2">protocol</td><td colspan="4">checksum</td></tr>
<tr><td colspan="8">source address</td></tr>
<tr><td colspan="8">destination address</td></tr>
<tr><td colspan="8">options</td></tr>
<tr><td colspan="8">payload (transport-layer segment)</td></tr>
</table>

IP packet example

- Here's an example of the values (in hex) that might be stored in an IPv4 packet
- Note that IPv6 packets are similar but simpler, because they don't have optional fields

Header	4500	IPv4, length = 20 bytes (5 words)
	0060	total length = 96 bytes
	0000 0000	ID, flags, offset (not used here)
	08	TTL
	06	protocol (TCP)
	6862	checksum
	867e 8ddd	source address 134.126.141.221
	5dd8 d822	destination address 93.184.216.34
Payload	…	

Network routing protocols

- The Internet is a network of networks
  - Each independent network controlled by a single entity is called an **autonomous system** (**AS**)
- Each AS connects to other ASes at gateway routers
  - BGP is a protocol that describes how these routers communicate to each other the paths through them to other networks
- Within an AS, OSPF, RIP, and other protocols determine the fastest route through the network
  - OSPF uses Dijkstra's shortest path algorithm based on time delays, broadcasting information to other routers
  - Alternatively, when a router using RIP discovers a new shortest path, it forwards the information only to its neighbors

Link Layer

Link layer

- The Internet layer focuses on routing packets through networks
- The **link layer** focuses on forwarding packets from point to point
- This forwarding all happens within a single kind of technology
- Things can go wrong at this fundamental level of networking:
  - **Processing delay** because checksums and other information have to be computed
  - **Queuing delay** because other packets are waiting to be sent
  - **Transmission delay** because converting to the physical layer takes work
  - **Propagation delay** because the physical layer can't send data instantly
- All these delays can add up

Ethernet

- Ethernet is one of the best known examples of link level protocols
- Ethernet is a collection of standards for communicating over copper or fiber optics
- Like higher level protocols, Ethernet also wraps its data with a header (and a footer too)
  - Typically, link layer packets are called **frames**
- For historical reasons, Ethernet frames are described in **octets** (always 8 bits) rather than bytes (which used to be variable in size)

Ethernet frames

- An Ethernet frame uses:
  - 8 octets for a preamble that's always the same, to mark the start of a message
  - 6 octets for destination address
  - 6 octets for source address
  - 2 octets for type of Ethernet
  - A payload of variable size
  - 4 octets for a **cyclic redundancy check** (**CRC**), an error checking value computed from the whole frame that is stronger than a checksum
- Source and destination addresses are **media access control** (**MAC**) addresses that are usually the same for a device's entire life
- **Address Resolution Protocol** (**ARP**) is used to ask devices on the network for their MAC based on their IP

Size	8 octets	6 octets	6 octets	2 octets	varies	4 octets
Purpose	Preamble	Destination	Source	Type	Payload	CRC
Example	`aaaaaaaaaaaaaaab`	`f0def12cc22b`	`f45c89bd332d`	`0800`	`...`	`64713722`

Stacks on stacks on stacks

- Here's an example of all the layers together:

Ethernet header	aaaa	aaaa	aaaa	aaab	f0de	f12c	c22b	f45c	89bd	332d	0800		
IPv4 header	4500	0060	0000	0000	0806	6862	867e	8ddd	5dd8	d822			
TCP header	1388	0050	0000	0017	0000	002a	5010	1000	cf33	0000			
HTTP header	4745	5420	2f20	4854	5450	2f31	2e31	0d0a	486f	7374			
	3a20	6578	616d	706c	652e	636f	6d0d	0a43	6f6e	6e65			
	6374	696f	6e3a	2063	6c6f	7365	0d0a	0d0a					
Ethernet FCS	6471	3722											

TCP Payload

IPv4 Payload

Ethernet Payload

Wireless

- Wireless communication differs from wired at the link and physical layers and sometimes above
- There are a few important wireless network technologies:
  - **Wi-Fi** is a set of standards designed to replace normal wired networking connections
  - **Bluetooth** is designed for short-range mobile ad hoc networks (MANETS)
    - Uses a star topology where many peripherals connect to a central devices
  - **Zigbee** uses a wireless mesh network for communicating between many low powered devices
    - Popular for Internet of Things (IoT) applications

Upcoming

Next time…

- Exam 2!
  - During class time

Reminders

- Finish Assignment 5
  - **Due tonight by midnight!**
- **Exam 2 in class on Monday!**
- Review Chapters 4-5