CS 3700
Networks and Distributed Systems

Lecture 1: Logistics, Networking Programming, Overview
2 Outline

- Course Logistics
- Networking Overview
- Intro to Network Programming
Hello!

- Welcome to CS 3700
  - Are you in the right classroom?
  - Okay, good.

- Who am I?
  - Professor Alan Mislove
  - amislove@ccs.neu.edu
  - West Village H 250
  - Office Hours: 4:30-5:30pm Mondays
Why Take This Course?

- How many of you have checked your e-mail, FB, texts...
Why Take This Course?

- How many of you have checked your e-mail, FB, texts... Today?
Why Take This Course?

- How many of you have checked your e-mail, FB, texts…
  - Today?
  - In the past hour?
Why Take This Course?

- How many of you have checked your e-mail, FB, texts...
  - Today?
  - In the past hour?
  - Since I started talking?
Networks and dist. systems are ubiquitous

- Touch every part of our daily life
  - Web search
  - Social networking
  - Watching movies
  - Ordering merchandise
  - Wasting time
Networks and dist. systems are ubiquitous

- Touch every part of our daily life
  - Web search
  - Social networking
  - Watching movies
  - Ordering merchandise
  - Wasting time

![Google Logo](image)
Networks and dist. systems are ubiquitous

- Touch every part of our daily life
  - Web search
  - Social networking
  - Watching movies
  - Ordering merchandise
  - Wasting time
Networks and dist. systems are ubiquitous

- Touch every part of our daily life
  - Web search
  - Social networking
  - Watching movies
  - Ordering merchandise
  - Wasting time
Networks and dist. systems are ubiquitous

- Touch every part of our daily life
  - Web search
  - Social networking
  - Watching movies
  - Ordering merchandise
  - Wasting time
Networks and dist. systems are ubiquitous

- Touch every part of our daily life
  - Web search
  - Social networking
  - Watching movies
  - Ordering merchandise
  - Wasting time
Networks and dist. systems are ubiquitous

- Networking is one of the most critical topics in CS
  - There would be no...
    - Big Data
    - Cloud
    - Apps or Mobile Computing
  - ... without networks
Goals

- Fundamental understanding of networking and systems
  - All the way from bits on a wire…
  - … across the ever-evolving Internet…
  - … to a complex distributed application

- Focus on software and protocols
  - Not hardware
  - Minimal theory

- Project-centric, hands on experience
  - Real projects, protocols, etc
Online Resources

- http://www.ccs.neu.edu/~amislove/cs3700/spring15

- Class forum is on Piazza
  - Sign up today!
  - Install their iPhone/Android app

- When in doubt, post to Piazza
  - Piazza is preferable to email
  - Use #hashtags (#homework1, #lecture2, #project3, etc.)
Teaching Style

- I am a networking and systems researcher
  - Things make sense to me that may not make sense to you
  - I talk fast if nobody stops me

- Solution: ask questions!
  - Seriously, ask questions
  - Standing up here in silence is very awkward
  - I will stand here until you answer my questions

- Help me learn your names
  - Say your name before each question
Textbook

- Two books, both optional
  - Computer Networks: A Systems Approach
  - Distributed Systems: Concepts and Design
# Workload

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projects (5)</td>
<td>1%, 12%, 15%, 12%, 15%</td>
</tr>
<tr>
<td>Homeworks (10)</td>
<td>1.5% each</td>
</tr>
<tr>
<td>Midterm</td>
<td>12.5%</td>
</tr>
<tr>
<td>Final</td>
<td>12.5%</td>
</tr>
<tr>
<td>Participation</td>
<td>5%</td>
</tr>
</tbody>
</table>
Projects

- This course is project-centric
  - Designed to give you real networking experience
  - Start early!
  - Seriously, start early!

- 5 projects
  - Due at 11:59:59pm on specified date
  - Use turn-in scripts to submit your code, documentation, etc.
  - Working code is paramount
Project Logistics

- **Languages**
  - You may choose the language for (most of) the projects
    - Code must compile on the CCIS Linux machines

- Project 0 is released now, due next week

- Project questions?
  - Post them on Piazza!
Project Groups

- Projects will be completed in groups of two
  - Unless we have odd numbers…

- Partner selection
  - Pick whoever you want
  - You may switch partners between projects
  - Do not complain to me about your lazy partner
    - Hey, you picked them

- Can’t find a partner?
  - Post a message on Piazza!
Late Policy

- Each student is given 4 slip days that they can use at any time to extend a deadline
  - You don’t need to ask me, just turn-in stuff late
  - All group members must have unused slip days
    - i.e. if one member has zero slip days left, the whole group is late

- Assignments are due at 11:59:59, **no exceptions**
  - 1 second late = 1 hour late = 1 day late
  - 20% off per day late
Exams

- Midterm and Final
  - 1-2 hours, in class
  - Midterm on networking, final on distributed systems
  - The final will not be cumulative

- All exams are:
  - Closed book, closed notes, leave the laptop at home
  - You may have a 1-page double-sided “cheat sheet”
  - And use a calculator
Cheating

- Do not do it
  - Seriously, don’t make me say it again
- Cheating is an automatic zero
  - Will be referred to the university for discipline and possible expulsion
- Project code must be original
  - You and your groupmates only
    - Unless we give you starter code, obviously
  - StackOverflow/Quora are not your friends
  - If you have questions about an online resource, ask me
Questions?
Outline

- Course Logistics
- Networking Overview
- Intro to Network Programming
What is a Comm. Network?
What is a Comm. Network?

A communications network is a network of links and nodes arranged so that messages may be passed from one part of the network to another.
A communications network is a network of **links** and **nodes** arranged so that **messages** may be passed from one part of the network to another.

- **What are nodes and links?**
  - People and roads
  - Telephones and switches
  - Computers and routers
What is a Comm. Network?

A communications network is a network of **links** and **nodes** arranged so that **messages** may be passed from one part of the network to another.

- **What are nodes and links?**
  - People and roads
  - Telephones and switches
  - Computers and routers

- **What is a message?**
  - **Information**
What is a Comm. Network?

A communications network is a network of **links** and **nodes** arranged so that **messages** may be passed from one part of the network to another.

- **What are nodes and links?**
  - People and roads
  - Telephones and switches
  - Computers and routers

- **What is a message?**
  - **Information**

**Networks are key for:**
- **Speed**
- **Distance**
Networks are Fundamental
Networks are Fundamental

Smoke Signals!
Networks are Old

- 2400 BC: courier networks in Egypt
- 550 BC: postal service invented in Persia
Networks are Old

- 2400 BC: courier networks in Egypt
- 550 BC: postal service invented in Persia

Problems:
- Speed
- Reliability
- Security
Towards Electric Communication

1837: Telegraph invented by Samuel Morse
- Distance: 10 miles
- Speed: 10 words per minute
- In use until 1985!
Towards Electric Communication

- 1837: Telegraph invented by Samuel Morse
  - Distance: 10 miles
  - Speed: 10 words per minute
  - In use until 1985!

- Key challenge: how to encode information?
  - Originally used unary encoding
    - A •
    - B ••
    - C •••
    - D ••••
    - E •••••
1837: Telegraph invented by Samuel Morse
- Distance: 10 miles
- Speed: 10 words per minute
- In use until 1985!

Key challenge: how to encode information?
- Originally used unary encoding:
  - A •
  - B ••
  - C •••
  - D ••••
  - E •••••
- Next generation: binary encoding:
  - A •–
  - B –•••
  - C –•–•
  - D –••
  - E •••••
1837: Telegraph invented by Samuel Morse
- Distance: 10 miles
- Speed: 10 words per minute
- In use until 1985!

Key challenge: how to encode information?
- Originally used unary encoding
  - A • B •• C ••• D •••• E •••••
- Next generation: binary encoding
  - A •– B –••• C –•–• D –•• E •
Telephony

- 1876 – Alexander Graham Bell invents the telephone
1876 – Alexander Graham Bell invents the telephone

Key challenge: how to scale the network?

Originally, all phones were directly connected

- $O(n^2)$ complexity; $n*(n–1)/2$
Telephony

- 1876 – Alexander Graham Bell invents the telephone
- Key challenge: how to scale the network?
  - Originally, all phones were directly connected
    - $O(n^2)$ complexity; $n*(n-1)/2$
  - 1878: Switching
Telephony

- 1876 – Alexander Graham Bell invents the telephone
- Key challenge: how to scale the network?
  - Originally, all phones were directly connected
    - $O(n^2)$ complexity; $n*(n-1)/2$
  - 1878: Switching
Telephony

- 1876: Alexander Graham Bell invents the telephone
- Key challenge: how to scale the network?

Originally, all phones were directly connected with $O(n^2)$ complexity; $n*(n–1)/2$ connections.

1878: Switching systems were introduced to manage the network more effectively.
1876 – Alexander Graham Bell invents the telephone

Key challenge: how to scale the network?

- Originally, all phones were directly connected
  - $O(n^2)$ complexity; $n^*(n-1)/2$
- 1878: Switching
- 1937: Trunk lines + multiplexing
1876: Alexander Graham Bell invents the telephone.

**Advantages**
- Easy to use
- Switching mitigates complexity
- Makes cable management tractable

**Problems**
- Manual switching
- 1918: Cross country call took 15 minutes to set up
Growth of the Telephone Network

- 1881: Twisted pair for local loops
- 1885: AT&T formed
- 1892: Automatic telephone switches
- 1903: 3 million telephones in the US
- 1915: First transcontinental cable
- 1927: First transatlantic cable
- 1937: First round-the-world call
- 1946: National numbering plan
Crazy idea: Packet switching

- Telephone networks are circuit switched
  - Each call reserves resources end-to-end
  - Provides excellent quality of service
- Problems
Crazy idea: Packet switching

- Telephone networks are circuit switched
  - Each call reserves resources end-to-end
  - Provides excellent quality of service
- Problems
  - Resource intense (what if the circuit is idle?)
  - Complex network components (per circuit state, security)
Crazy idea: Packet switching

- Telephone networks are circuit switched
  - Each call reserves resources end-to-end
  - Provides excellent quality of service
- Problems
  - Resource intense (what if the circuit is idle?)
  - Complex network components (per circuit state, security)
- Packet switching
  - No connection state, network is store-and-forward
  - Minimal network assumptions
  - Statistical multiplexing gives high overall utilization
The World’s Most Successful Computer Science Research Project

The ARPA Network

DEC 1969

4 Nodes

Figure 6.2 Drawing of 4 Node Network (Courtesy of Alex McKenzie)
History of the Internet

- 1961: Kleinrock @ MIT: packet-switched network
- 1962: Licklider’s vision of Galactic Network
- 1965: Roberts connects computers over phone line
- 1967: Roberts publishes vision of ARPANET
- 1969: BBN installs first InterfaceMsgProcessor at UCLA
- 1970: Network Control Protocol (NCP)
- 1972: Public demonstration of ARPANET
- 1972: Kahn @ DARPA advocates Open Architecture
- 1972: Vint Cerf @ Stanford writes TCP
The 1960s
The 1960s
1973

ARPA NETWORK, LOGICAL MAP, SEPTEMBER 1973

[Diagram showing the network connections as of September 1973]
Growing Pains

- Problem: early networks used incompatible protocols
Kahn’s Ground Rules

1. Each network is independent, cannot be forced to change
2. Best-effort communication (i.e. no guarantees)
3. Routers connect networks
4. No global control
Kahn’s Ground Rules

1. Each network is independent, cannot be forced to change
2. Best-effort communication (i.e. no guarantees)
3. Routers connect networks
4. No global control

- Principles behind the development of IP
- Led to the Internet as we know it
- Internet is still structured as independent networks
The Birth of Routing
The Birth of Routing

Trivia

• Kahn believed that there would only be \(~20\) networks.
• He was way off.
• Why?
More Internet History

- 1974: Cerf and Kahn paper on TCP (IP kept separate)
- 1980: TCP/IP adopted as defense standard
- 1983: ARPANET and MILNET split
- 1983: Global NCP to TCP/IP flag day
- 198x: Internet melts down due to congestion
- 1986: Van Jacobson saves the Internet (BSD TCP)
- 1987: NSFNET merges with other networks
- 1988: Deering and Cheriton propose multicast
- 199x: QoS rises and falls, ATM rises and falls
- 1994: NSF backbone dismantled, private backbone
- 1999-present: The Internet boom and bust ... and boom
- 2007: Release of iPhone, rise of Mobile Internet
More Internet History

- 1974: Cerf and Kahn paper on TCP (IP kept separate)
- 1980: TCP/IP adopted as defense standard
- 1983: ARPANET and MILNET split
- 1983: Global NCP to TCP/IP flag day
- 198x: Internet melts down due to congestion
- 1986: Van Jacobson saves the Internet (BSD TCP)
- 1987: NSFNET merges with other networks
- 1988: Deering and Cheriton propose multicast
- 199x: QoS rises and falls, ATM rises and falls
- 1994: NSF backbone dismantled, private backbone
- 1999-present: The Internet boom and bust … and boom
- 2007: Release of iPhone, rise of Mobile Internet

What is next?
Internet Applications Over Time

- 1972: Email
- 1973: Telnet – remote access to computing
- 1982: DNS – “phonebook” of the Internet
- 1985: FTP – remote file access
- 1989: NFS – remote file systems
- 1991: The World Wide Web (WWW) goes public
- 1995: SSH – secure remote shell access
- 1995-1997: Instant messaging (ICQ, AIM)
- 1998: Google
- 1999: Napster, birth of P2P
- 2001: Bittorrent
- 2004: Facebook
- 2005: YouTube
- 2007: The iPhone
Internet Applications Over Time

- 1972: Email
- 1973: Telnet – remote access to computing
- 1982: DNS – “phonebook” of the Internet
- 1985: FTP – remote file access
- 1989: NFS – remote file systems
- 1991: The World Wide Web (WWW) goes public
- 1995: SSH – secure remote shell access
- 1995-1997: Instant messaging (ICQ, AIM)
- 1998: Google
- 1999: Napster, birth of P2P
- 2001: Bittorrent
- 2004: Facebook
- 2005: YouTube
- 2007: The iPhone

Invented by Shawn Fanning at NEU
Internet Applications Over Time

- 1972: Email
- 1973: Telnet – remote access to computing
- 1982: DNS – “phonebook” of the Internet
- 1985: FTP – remote file access
- 1989: NFS – remote file systems
- 1991: The World Wide Web (WWW) goes public
- 1995: SSH – secure remote shell access
- 1995-1997: Instant messaging (ICQ, AIM)
- 1998: Google
- 1999: Napster, birth of P2P
- 2001: Bittorrent
- 2004: Facebook
- 2005: YouTube
- 2007: The iPhone
Internet Applications Over Time

- 1972: Email
- 1973: Telnet – remote access to computing
- 1982: DNS – “phonebook” of the Internet
- 1985: FTP – remote file access
- 1989: NFS – remote file systems
- 1995: SSH – secure remote shell access
- 1995-1997: Instant messaging (ICQ, AIM)
- 1998: Google
- 1999: Napster, birth of P2P
- 2001: Bittorrent
- 2004: Facebook
- 2005: YouTube
- 2007: The iPhone

What is next?
Takeaways

- Communication is fundamental to human nature
Takeaways

- Communication is fundamental to human nature
- Key concepts have existed for a long time
  - Speed/bandwidth
  - Latency
  - Switching
  - Packets vs. circuits
  - Encoding
  - Cable management
  - Multiplexing
  - Routing
Takeaways

- Communication is fundamental to human nature
- Key concepts have existed for a long time
  - Speed/bandwidth
  - Latency
  - Switching
  - Packets vs. circuits
- The Internet has changed the world
  - Promise of free ($) and free (freedom) communication
  - Shrank the world
Takeaways

- Communication is fundamental to human nature
- Key concepts have existed for a long time
  - Speed/bandwidth
  - Latency
  - Switching
  - Packets vs. circuits
- The Internet has changed the world
  - Promise of free ($) and free (freedom) communication
  - Shrank the world
- What made the Internet so successful? Stay tuned!

- Encoding
- Cable management
- Multiplexing
- Routing
Course Logistics
Networking Overview
Intro to Network Programming
Socket Programming

- Goal: familiarize yourself with socket programming
  - Why am I presenting C sockets?
  - Because C sockets are the de-facto standard for networking APIs
Socket Programming

- Goal: familiarize yourself with socket programming
  - Why am I presenting C sockets?
  - Because C sockets are the de-facto standard for networking APIs

- Project 0: Implement a semi-trivial protocol
  - We will have a server set up for you
  - There may be chances for extra credit ;}

...
C Sockets

- Socket API since 1983
  - Berkeley Sockets
  - BSD Sockets (debuted with BSD 4.2)
  - Unix Sockets (originally included with AT&T Unix)
  - Posix Sockets (slight modifications)

- Original interface of TCP/IP
  - All other socket APIs based on C sockets
Clients and Servers

- A fundamental problem: rendezvous
  - One or more parties want to provide a service
  - One or more parties want to use the service
  - How do you get them together?
Clients and Servers

- A fundamental problem: rendezvous
  - One or more parties want to provide a service
  - One or more parties want to use the service
  - How do you get them together?

- Solution: client-server architecture
  - Client: initiator of communication
  - Server: responder
  - At least one side has to wait for the other
    - Service provider (server) sits and waits
    - Clients locates servers, initiates contact
    - Use well-known semantic names for location (DNS)
# Key Differences

<table>
<thead>
<tr>
<th>Clients</th>
<th>Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute on-demand</td>
<td>Always-on</td>
</tr>
<tr>
<td>Unprivileged</td>
<td>Privileged</td>
</tr>
<tr>
<td>Simple</td>
<td>Complex</td>
</tr>
<tr>
<td>(Usually) sequential</td>
<td>(Massively) concurrent</td>
</tr>
<tr>
<td>Not performance sensitive</td>
<td>High performance</td>
</tr>
<tr>
<td></td>
<td>Scalable</td>
</tr>
</tbody>
</table>
Similarities

- Share common protocols
  - Application layer
  - Transport layer
  - Network layer
- Both rely on APIs for network access
Sockets

- Basic network abstraction: the socket
Sockets

- Basic network abstraction: the socket

- Socket: an object that allows reading/writing from a network interface

- In Unix, sockets are just file descriptors
  - `read()` and `write()` both work on sockets
  - Caution: socket calls are blocking
C Socket API Overview

**Clients**
1. `gethostbyname()`
2. `socket()`
3. `connect()`
4. `write()` / `send()`
5. `read()` / `recv()`
6. `close()`

**Servers**
1. `socket()`
2. `bind()`
3. `listen()`
4. `while (whatever) {`
5. `accept()`
6. `read()` / `recv()`
7. `write()` / `send()`
8. `close()`
9. `}`
10. `close()`
C Socket API Overview

### Clients
1. `gethostbyname()`
2. `socket()`
3. `connect()`
4. `write()` / `send()`
5. `read()` / `recv()`
6. `close()`

### Servers
1. `socket()`
2. `bind()`
3. `listen()`
4. `while (whatever) {
   5.   `accept()`
   6.   `read()` / `recv()`
   7.   `write()` / `send()`
   8.   `close()`
   9. }
10. `close()`
Most basic call, used by clients and servers
Get a new socket

Parameters
- int domain: a constant, usually PF_INET
- int type: a constant, usually SOCK_STREAM or SOCK_DGRAM
  - SOCK_STREAM means TCP
  - SOCK_DGRAM means UDP
- int protocol: usually 0 (zero)

Return: new file descriptor, -1 on error

Many other constants are available
Why so many options?
int socket(int, int, int)

- Most basic call, used by clients and servers
- Get a new socket
- Parameters
  - int domain: a constant, usually PF_INET
  - int type: a constant, usually SOCK_STREAM or SOCK_DGRAM
  - int protocol: usually 0 (zero)

- Return: new file descriptor, -1 on error
- Many other constants are available
  - Why so many options?

**The C socket API is extensible.**
- The Internet isn’t the only network domain
- TCP/UDP aren’t the only transport protocols
- In theory, transport protocols may have different dialects
int bind(int, struct sockaddr *, int)

- Used by servers to associate a socket to a network interface and a port
  - Why is this necessary?

- Parameters:
  - int sockfd: an unbound socket
  - struct sockaddr * my_addr: the desired IP address and port
  - int addrlen: sizeof(struct sockaddr)

- Return: 0 on success, -1 on failure
  - Why might bind() fail?
int bind(int, struct sockaddr *, int)

- Used by servers to associate a socket to a network interface and a port
  - Why is this necessary?
- Parameters:
  - int sockfd: an unbound socket
  - struct sockaddr * my_addr: the desired IP address and port
  - int addrlen: sizeof(struct sockaddr)

Return: 0 on success, -1 on failure

Why might `bind()` fail?
- Each machine may have multiple network interfaces
  - Example: Wifi and Ethernet in your laptop
  - Example: Cellular and Bluetooth in your phone
- Each network interface has its own IP address
- We’ll talk about ports next…
int bind(int, struct sockaddr *, int)

- Used by servers to associate a socket to a network interface and a port
  - Why is this necessary?
- Parameters:
  - int sockfd: an unbound socket
  - struct sockaddr * my_addr: the desired IP address and port
  - int addrlen: sizeof(struct sockaddr)
- Return: 0 on success, -1 on failure
  - Why might bind() fail?
Port Numbers
Port Numbers

- Basic mechanism for multiplexing applications per host
  - 65,535 ports available
  - Why?
Port Numbers

- Basic mechanism for multiplexing applications per host
  - 65,535 ports available
  - Why?

TCP/UDP port field is 16-bits wide
Port Numbers

- Basic mechanism for **multiplexing** applications per host
  - 65,535 ports available
  - Why?
- Ports <1024 are **reserved**
  - Only privileged processes (e.g. superuser) may access
  - Why?
  - Does this cause security issues?
Port Numbers

- Basic mechanism for **multiplexing** applications per host
  - 65,535 ports available
  - Why?
- Ports <1024 are **reserved**
  - Only privileged processes (e.g. superuser) may access
  - Why?
  - Does this cause security issues?

  - In olden times, all important apps used low port numbers
  - Examples: IMAP, POP, HTTP, SSH, FTP
  - This rule is no longer useful
Port Numbers

- Basic mechanism for **multiplexing** applications per host
  - 65,535 ports available
  - Why?

- Ports <1024 are **reserved**
  - Only privileged processes (e.g. superuser) may access
  - Why?
  - Does this cause security issues?

- “I tried to open a port and got an error”
  - Port collision: only one app per port per host
  - Dangling sockets...
Dangling Sockets

- Common error: bind fails with “already in use” error
- OS kernel keeps sockets alive in memory after close()
  - Usually a one minute timeout
  - Why?
Dangling Sockets

- Common error: bind fails with “already in use” error
- OS kernel keeps sockets alive in memory after close()
  - Usually a one minute timeout
  - Why?

- Closing a TCP socket is a multi-step process
- Involves contacting the remote machine
- “Hey, this connection is closing”
- Remote machine must acknowledge the closing
- All this book keeping takes time
Dangling Sockets

- Common error: bind fails with “already in use” error
- OS kernel keeps sockets alive in memory after close()
  - Usually a one minute timeout
  - Why?
Dangling Sockets

- Common error: bind fails with "already in use" error
- OS kernel keeps sockets alive in memory after close()
  - Usually a one minute timeout
  - Why?

- Allowing socket reuse

```c
int yes=1;
if (setsockopt(listener, SOL_SOCKET, SO_REUSEADDR, &yes, sizeof(int))
   == -1) { perror("setsockopt"); exit(1); }
```
- Structure for storing naming information
  - But, different networks have different naming conventions
  - Example: IPv4 (32-bit addresses) vs. IPv6 (64-bit addresses)
**struct sockaddr**

- Structure for storing naming information
  - But, different networks have different naming conventions
  - Example: IPv4 (32-bit addresses) vs. IPv6 (64-bit addresses)
- In practice, use more specific structure implementation
  1. `struct sockaddr_in my_addr;`
  2. `memset(&my_addr, 0, sizeof(sockaddr_in));`
  3. `my_addr.sin_family = htons(AF_INET);`
  4. `my_addr.sin_port = htons(MyAwesomePort);`
  5. `my_addr.sin_addr.s_addr = inet_addr("10.12.110.57");`
Little Endian vs. Big Endian

- Not a big deal as long as data stays local
- What about when hosts communicate over networks?
Little Endian vs. Big Endian
- Not a big deal as long as data stays local
- What about when hosts communicate over networks?

Network byte order
- Standardized to Big Endian
- Be careful: x86 is Little Endian

Functions for converting host order to network order
- h to n s – host to network short (16 bits)
- h to n l – host to network long (32 bits)
- n to h * – the opposite
If you don’t care about the port

```
my_addr.sin_port = htons(0);
```

Chooses a free port at random

This is rarely the behavior you want
Binding Shortcuts

- If you don’t care about the port
  - `my_addr.sin_port = htons(0);`
  - Chooses a free port at random
  - This is rarely the behavior you want

- If you don’t care about the IP address
  - `my_addr.sin_addr.s_addr = htonl(INADDR_ANY);`
  - `INADDR_ANY == 0`
  - Meaning: don’t bind to a specific IP
  - Traffic on any interface will reach the server
    - Assuming its on the right port
  - This is usually the behavior you want
int listen(int, int)

- Put a socket into listen mode
  - Used on the server side
  - Wait around for a client to connect()

- Parameters
  - int sockfd: the socket
  - int backlog: length of the pending connection queue
    - New connections wait around until you accept() them
    - Just set this to a semi-large number, e.g. 1000

- Return: 0 on success, -1 on error
int accept(int sockfd, void *addr, int *addrlen)

- Accept an incoming connection on a socket

- **Parameters**
  - int *sockfd*: the `listen()`ing socket
  - void *addr*: pointer to an empty `struct sockaddr`
    - Clients IP address and port number go here
    - In practice, use a `struct sockaddr_in`
  - int *addrlen*: length of the data in addr
    - In practice, `addrlen == sizeof(struct sockaddr_in)`

- **Return**: a new socket for the client, or -1 on error

- **Why?**
int accept(int, void *, int *)

- Accept an incoming connection on a socket

- Parameters
  - int sockfd: the listen()ing socket
  - void * addr: pointer to an empty struct sockaddr

  - Clients IP address and port number go here
  - In practice, use a struct sockaddr_in
  - int * addrlen: length of the data in addr
  - In practice, addrlen == sizeof(struct sockaddr_in)

- Return: a new socket for the client, or -1 on error

- Why?
  - You don’t want to consume your listen() socket
  - Otherwise, how would you serve more clients?
  - Closing a client connection shouldn’t close the server
close(int sockfd)

- Close a socket
  - No more sending or receiving

- `shutdown(int sockfd, int how)`
  - Partially close a socket
    - `how = 0;` // no more receiving
    - `how = 1;` // no more sending
    - `how = 2;` // just like `close()`
  - Note: `shutdown()` does **not** free the file descriptor
  - Still need to `close()` to free the file descriptor
C Socket API Overview

Clients
1. `gethostbyname()`
2. `socket()`
3. `connect()`
4. `write() / send()`
5. `read() / recv()`
6. `close()`

Servers
1. `socket()`
2. `bind()`
3. `listen()`
4. `while (whatever) {
   5. `accept()
   6. `read() / recv()
   7. `write() / send()
   8. `close()
   9. }
10. `close()`
struct * gethostbyname(char *)

- Returns information about a given host
- Parameters
  - const char * name: the domain name or IP address of a host
  - Examples: “www.google.com”, “10.137.4.61”
- Return: pointer to a hostent structure, 0 on failure
  - Various fields, most of which aren’t important
  1. struct hostent * h = gethostname(“www.google.com”);
  2. struct sockaddr_in my_addr;
  3. memcpy(&my_addr.sin_addr.s_addr, h->h_addr, h->h_length);
struct * gethostbyname(char *)

- Returns information about a given host
- Parameters
  - const char * name: the domain name or IP address of a host
  - Examples: “www.google.com”, “10.1.37.4.61”
- Return: pointer to a hostent structure, 0 on failure
  - Various fields, most of which aren’t important
  1. struct hostent * h = gethostbyname(“www.google.com”);
  2. struct sockaddr_in my_addr;
  3. memcpy(&my_addr.sin_addr.s_addr, h->h_addr, h->h_length);
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen)

- Connect a client socket to a `listen()`ing server socket

- **Parameters**
  - `int sockfd`: the client socket
  - `struct sockaddr * serv_addr`: address and port of the server
  - `int addrlen`: length of the sockaddr structure

- **Return**: 0 on success, -1 on failure

- Notice that we don’t `bind()` the client socket
  - Why?
write() and send()

- `ssize_t write(int fd, const void *buf, size_t count);`
  - `fd`: file descriptor (i.e. your socket)
  - `buf`: the buffer of data to send
  - `count`: number of bytes in `buf`
  - Return: number of bytes actually written

- `int send(int sockfd, const void *msg, int len, int flags);`
  - First three, same as above
  - `flags`: additional options, usually 0
  - Return: number of bytes actually written

- Do not assume that `count / len ==` the return value!
  - Why might this happen?
read() and recv()

- `ssize_t read(int fd, void *buf, size_t count);`
  - Fairly obvious what this does

- `int recv(int sockfd, void *buf, int len, unsigned int flags);`
  - Seeing a pattern yet?

Return values:
- `-1: there was an error reading from the socket`
  - Usually unrecoverable. `close()` the socket and move on
- `>0: number of bytes received`
  - May be less than `count / len`
- `0: the sender has closed the socket`
More Resources

- Beej’s famous socket tutorial