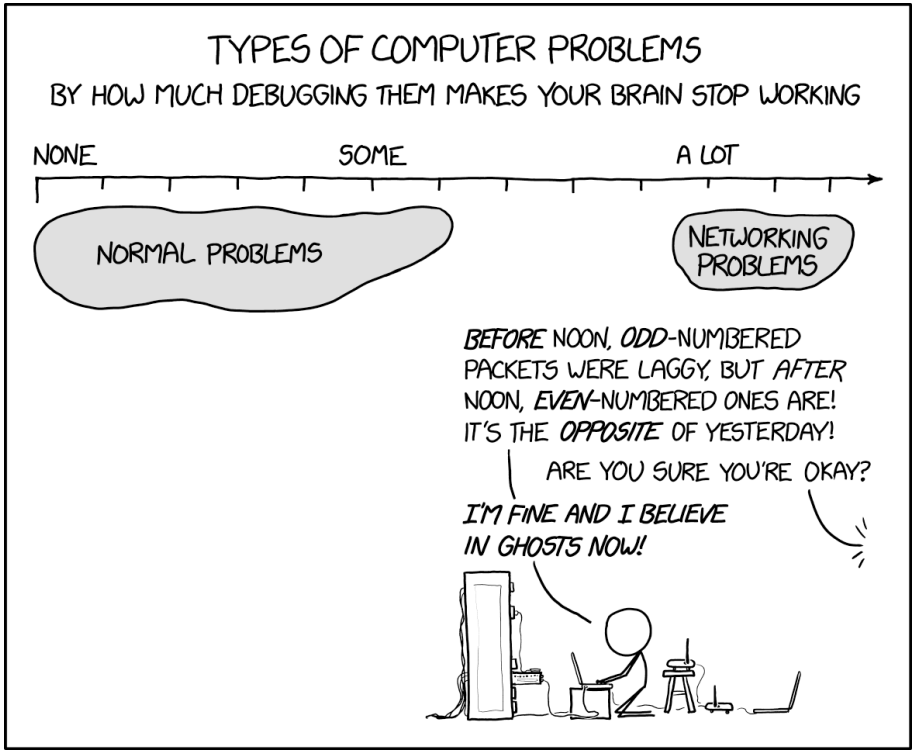
CS107, Lecture 17: Sockets Programming



Title Text: Look, the latency falls every time you clap your hands and say you believe.

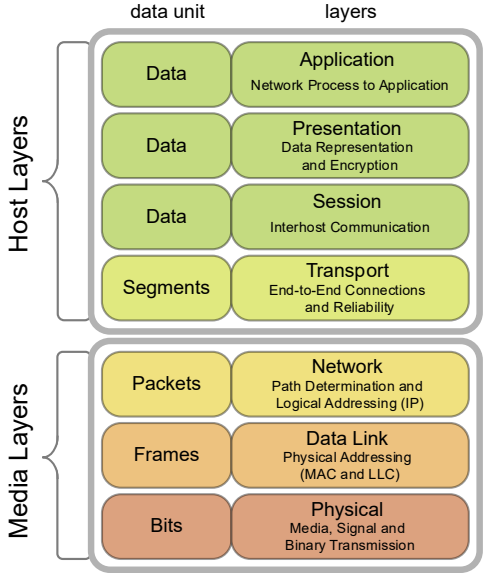
Slides by Adam Keppler and Daniel Rebelsky, modeled in part off of slides from Nick Troccoli and Jerry Cain, and content in part from Beej's Guide to Network Programming Using Internet Sockets

Announcements

- Wednesday Lecture is a Review Session (8/5)
  - Please comment on the Ed Announcement with preferred topics
- No lab next week (Week 7 has lab, no lab Week 8)
- Friday Lecture is a Social (Optional, board games and snacks)
- No Lecture (8/10 and 8/12) study for the Final!
- Office Hours will be held at the usual time Monday 8/10 and Wednesday 8/12

Quick Overview

- Networking is built as a stack of layers
- Each layer typically only talks to the ones directly above and below it
- Our programs live at the very top (the application layer)
- Sockets are the doorway down: the OS handles transport (TCP/UDP), network (IP), and everything below



Bits: The Physical Layer

- At the very bottom, everything is bits: electrons on copper, pulses of light in fiber, radio waves in the air
- The physical layer's only job is moving raw 1s and 0s from one end of a link to the other
- No structure or meaning, just raw signals
- Everything above exists to impose structure on this stream

… 0 1 0 1 1 0 0 1 1 1 0 1 0 0 1 0 1 1 0 1 0 0 0 1 1 0 1 0 …

(Binary Transmissions)

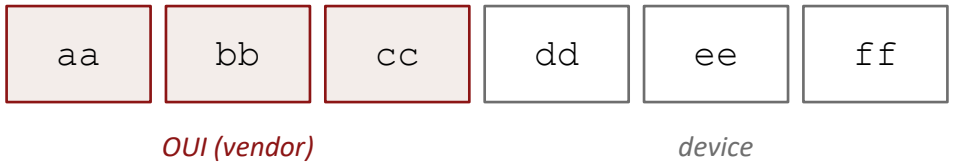
MAC Addresses

- A MAC (Media Access Control) address is the 48-bit identifier of a network interface (aka network interface card or NIC) assigned by the manufacturer, it is unique per NIC

- Written as six hex octets: the first three are the OUI (Organizationally Unique Identifier, i.e., the vendor), the last three identify the device

- The namespace is flat — no hierarchy, no relation to where the device is — which is exactly why MACs can't route across networks

- Modern OSes randomize Wi-Fi MACs per network to prevent tracking — a hardware ID that follows you is a privacy problem (a device is not forced into sending its real MAC)



Frames: The Data Link Layer

- The data link layer groups bits into frames
- Frames carry MAC addresses: 48-bit hardware addresses (e.g., aa:bb:cc:dd:ee:ff)
- MAC addresses are good for local delivery only — each hop has to unwrap the frame and re-wrap it for the next link as each device has its own NIC
- One device to one device communication

an Ethernet frame

Packets: The Network Layer

- The network layer (IP) takes over for delivery across multiple hops / devices (A Network)
- IP rides inside the frame, as part of the payload, typically one packet per frame
- Messages are often large and have to be sub-divided into smaller units called a packet
- The receiver must recombine the packets to see the message
- Packets have their own from/to-address, at this layer the addresses are Internet Protocol Addresses (IP Addresses), which survive end-to-end (unlike MAC addresses which are 1-hop only)

IPv4 Addresses

- An IPv4 address is 32 bits, typically written in dotted quad format, e.g. 127.0.0.1
- 127.0.0.1 is special it always refers to the current device
- 32 bits only makes for about 4 billion addresses … which we've fully allocated
    - The internet grew just a little faster than anyone expected in 1981
- Workarounds (like Network Address Translation (NAT)) stretched the supply, but the real fix was to allow for a bigger address space

IPv6 Addresses

- IPv6 addresses are 128 bits, enough to never run out? Or Famous last words? (roughly 3.4×10^{38} addresses)

- Written as eight groups of 16 bits in hex, separated by colons

- Notation shortcut: one run of consecutive zero groups can collapse to "::"

```
2607:f6d0:0:0:0:0:0:0     // full form
2607:f6d0::               // same address, zeros collapsed
```

DNS: Names Instead of Numbers

- The Domain Name System (DNS) allows us to use domain names / hostnames into human readable names that map to IP addresses

- So you can remember `web.stanford.edu` instead of `171.67.215.200`

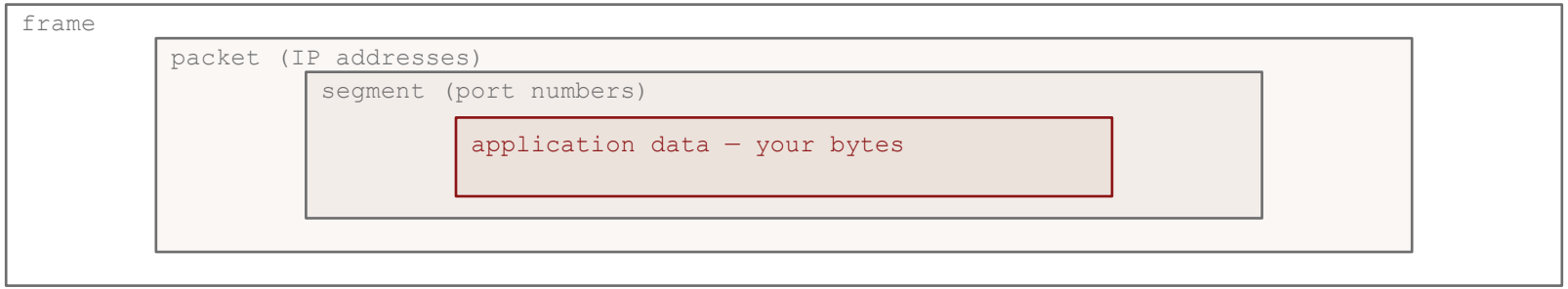
- DNS is a giant distributed contact list: you ask for a name, it answers with one or more IP addresses

- A single name can map to several addresses (IPv6 and IPv4, multiple servers)

```
web.stanford.edu    →    171.67.215.200
                         // resolved via DNS at connection time
```

Segments: The Transport Layer

- IP gets a packet to the right machine — but a machine runs many programs. Which one gets the data?
- The transport layer (TCP or UDP) chops application data into segments that ride inside packets, and adds port numbers to address the right application
- IP address = which machine; port = which program on it
- This is the layer sockets expose to us — and it comes in two flavors, TCP and UDP



encapsulation: each layer wraps the one above

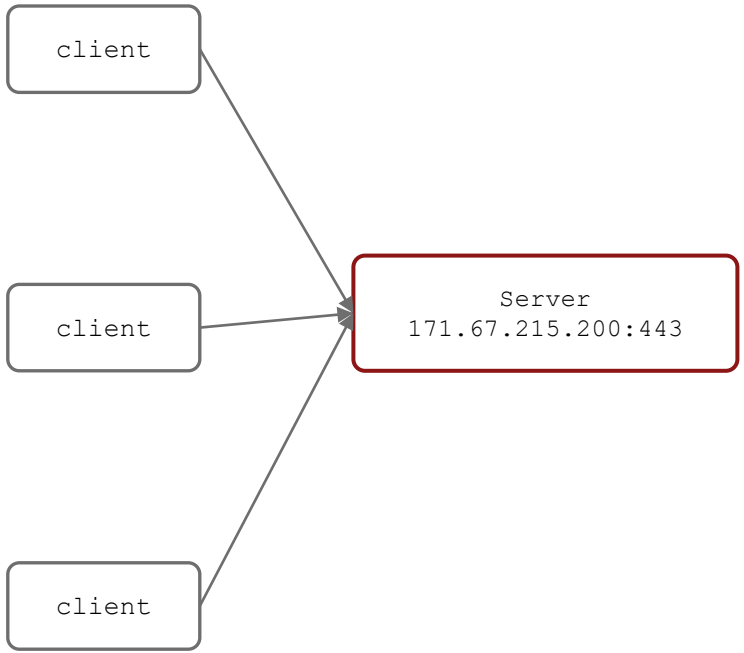
Ports

- A port number is 16 bits and identifies which application on a machine should get the data
- Official port usage is assigned by IANA; see also /etc/services on the myth machines
- Ports under 1024 are typically reserved — on the myths, you need special permission to bind to them (use high ports like 8080 for your own servers)

Port	Service
22	SSH
53	DNS
80	HTTP
443	HTTPS

The Client-Server Model

- For the rest of the lecture, we'll implicitly use this model: a server (imagine Google) serves data to one or more clients (people Googling)
- The server waits at a known address and port; clients initiate contact
- To reach a server, a client therefore needs both an IP address and a port number



TCP vs UDP

Both run on top of IP; both are identified by a 16-bit port — they differ in the promises they make.

Both put multi-byte header fields (like ports) in network byte order — big-endian

TCP — SOCK_STREAM

- "Reliable bytestream" abstraction — except in exceptional cases, data arrives correctly and in order
- Connection-oriented: handshake first, then talk
- Use for non-time-critical applications: web servers (HTTP prior to HTTP/3), ssh, email, …

UDP — SOCK_DGRAM

- Unreliable datagram abstraction — effectively a userspace wrapper around IP (hence "User Datagram Protocol")
- No connection, no delivery or ordering guarantees
- Use for time-critical applications or ones that tolerate loss: video conferencing, online gaming, DNS, …

htons() and Friends

- Four conversion functions, named by a simple scheme:

  h = host, n = network, s = short (16-bit), l = long (32-bit)

- You will need htons() every time you fill in a port by hand — forgetting it is the #1 “why is my server on port 36895 when I said 8080?” bug (since that’s 8080 with its bytes swapped)

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

addr.sin_port = htons(8080);          // right
addr.sin_port = 8080;                 // wrong: port 36895 on x86!
```

Detour: File Descriptors

- When working with Unix/Linux, there is a common joke that “everything is a file”, this is due to how the operating system exposes many interfaces in a similar way
- A file descriptor is an integer the OS hands to our process as an identifier for a channel that supports read/write syscalls and a set of modification options typically through `fcntl`
- A FILE * is a convenient wrapper around a file descriptor
- We always have these by default: 0 (stdin), 1 (stdout), 2 (stderr)
- We use file descriptors for both real files and for sockets (among other things), which is why working with socket can feel familiar

Detour: Error Handling

- Many system calls (and wrapping C functions) can fail
- In C, failure is often a negative return value, with errno (a global) set appropriately (see man errno; perror prints the corresponding message)

- We've mostly been able to ignore this so far, but we can't ignore it with sockets:
  - Explicitly check every return value that might fail, and write out the failure condition
  - Wrap functions in safe forms (e.g., the textbook creates Write from write)
  - Use macros or helper functions to help simplify and avoid repeating boilerplate code

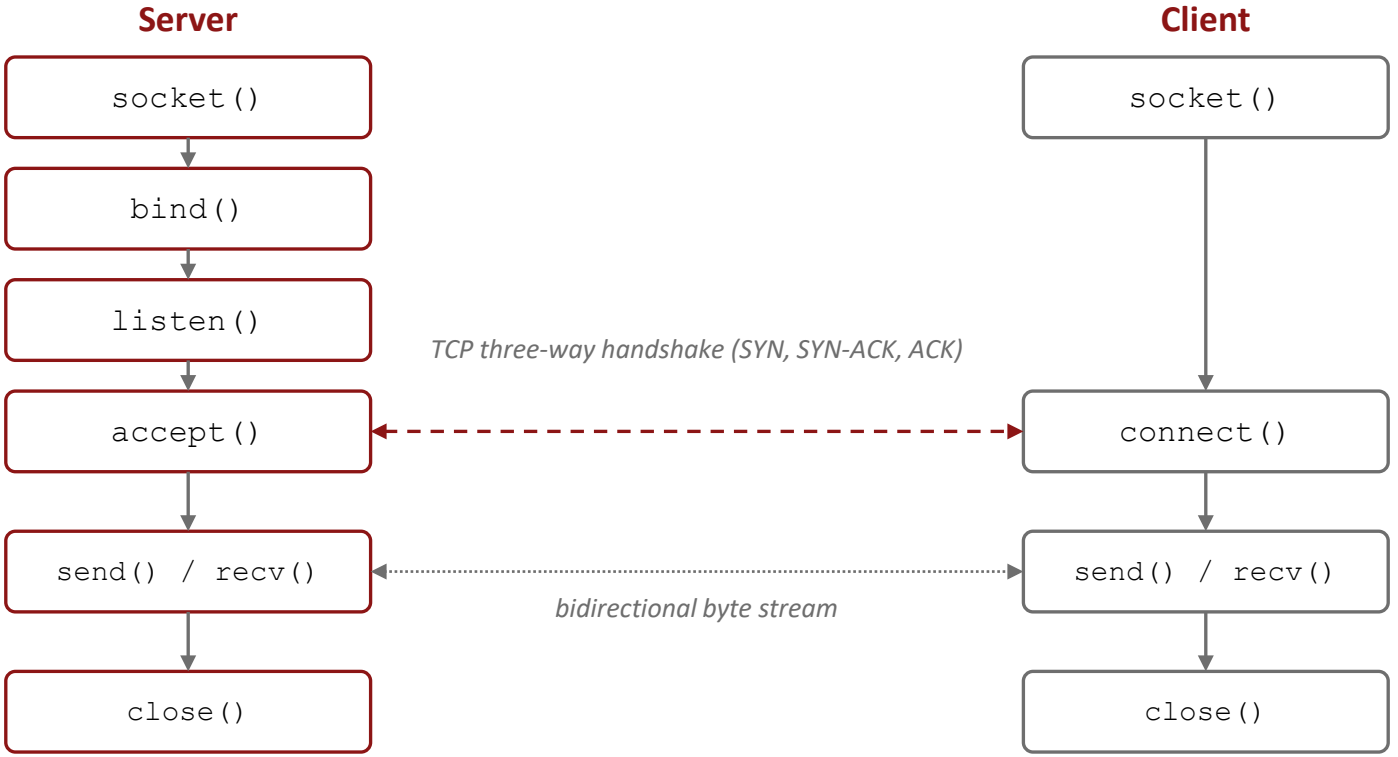
Detour: Man Pages and Beej's Guide

- Beej's Guide to Network Programming can be a friendlier starting point than the real man pages https://beej.us/guide/bgnet/

- Beej's guide is also the source for several code patterns you'll see today, it's a common reference for C sockets

Socket Programming Basics

The Life of a TCP Connection



This diagram is the map for the rest of the lecture — the footer on each slide shows where we are.

socket()

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

- The domain specifies what type of socket we want — for this lecture, one of PF_INET (IPv4) or PF_INET6 (IPv6)
    - PF => Protocol Family, AF => Address Family, PF and AF map 1-to-1 so they became the same
- The type for this lecture will always be SOCK_STREAM (meaning TCP; it could also be SOCK_DGRAM for UDP)
- The protocol is the protocol number (e.g., IPPROTO_TCP or IPPROTO_UDP) — but we can pass 0, since SOCK_STREAM implies TCP and the function will figure it out
- Returns a file descriptor on success and <0 on error (setting errno as appropriate)
    - The fd isn't connected to anything yet — it's just a handle; the next calls give it a job

struct sockaddr and struct sockaddr_in

- struct sockaddr is the generic type for a socket address — in practice we fill in a struct sockaddr_in (IPv4) or sockaddr_in6 (IPv6) and cast

```
struct sockaddr {                   // the generic "base type"
    unsigned short sa_family;       // address family, AF_xxx
    char sa_data[14];               // 14 bytes of protocol address
};

struct sockaddr_in {                // IPv4 — what we actually fill in
    short int sin_family;           // address family, AF_INET
    unsigned short int sin_port;    // port number (network byte order!)
    struct in_addr sin_addr;        // internet address
    unsigned char sin_zero[8];      // padding to sizeof(struct sockaddr)
};

struct in_addr {
    uint32_t s_addr;                // 32-bit IPv4 address
};
```

struct sockaddr_in6

- The IPv6 counterpart — same idea, bigger address, two extra fields you can ignore today

```
struct sockaddr_in6 {
    u_int16_t sin6_family;        // address family, AF_INET6
    u_int16_t sin6_port;          // port number (network byte order!)
    u_int32_t sin6_flowinfo;      // IPv6 flow information
    struct in6_addr sin6_addr;    // IPv6 address
    u_int32_t sin6_scope_id;      // scope ID
};

struct in6_addr {
    unsigned char s6_addr[16];    // 128-bit IPv6 address
};
```

- Good news: getaddrinfo() fills these in for you, either family
- You will rarely have to build one by hand

inet_pton(), inet_addr(), and inet_aton()

- Converting address strings (like “171.67.215.200”) into the binary form the structs need:

```
int         inet_aton(const char *cp, struct in_addr *inp); //ASCII to Network
in_addr_t inet_addr(const char *cp);
                    // cp is a dotted quad string, IPv4 ONLY

int inet_pton(int af, const char *src, void *dst);
                    // presentation to network
                    // af = AF_INET or AF_INET6 — handles both
```

- aton and addr only work for IPv4 addresses — prefer inet_pton for new code

- In practice, prefer getaddrinfo() over all of these — next slide

getaddrinfo()

```
int getaddrinfo(const char *node,     // e.g. "www.example.com" or IP
                const char *service,  // e.g. "http" or port number
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- One call that does it all: DNS lookup, service-name lookup, and struct filling — gives us a linked list of struct addrinfos

```
struct addrinfo {
    int ai_flags;              // AI_PASSIVE, AI_CANONNAME, etc.
    int ai_family;             // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype;           // SOCK_STREAM, SOCK_DGRAM
    int ai_protocol;           // use 0 for "any"
    size_t ai_addrlen;         // size of ai_addr in bytes
    struct sockaddr *ai_addr; // struct sockaddr_in or _in6
    char *ai_canonname;        // full canonical hostname
    struct addrinfo *ai_next; // linked list, next node
};
```

getaddrinfo() in practice

```
struct addrinfo hints;
struct addrinfo * res;
memset(&hints, 0, sizeof(hints));
hints.ai_family   = AF_UNSPEC;      // IPv4 or IPv6, don't care
hints.ai_socktype = SOCK_STREAM;    // TCP

int err = getaddrinfo("web.stanford.edu", "443", &hints, &res);
if (err != 0) { fprintf(stderr, "%s\n", gai_strerror(err)); /*...*/ }

for (struct addrinfo *p = res; p != NULL; p = p->ai_next) {
    // try socket() (+ connect()/bind()) with p's fields;
    // stop at the first one that works
}
freeaddrinfo(res);                  // don't leak the list!
```

- A name may resolve to several addresses (IPv6 + IPv4) — try each in order until one works
- getaddrinfo() does not set errno — use gai_strerror() on its return value; and freeaddrinfo() is a real heap free (Valgrind will notice)
- socklen_t is just an integer type for address lengths — pass sizeof(struct sockaddr_in) (or ai_addrlen from getaddrinfo)

bind()

```
int bind(int sockfd, const struct sockaddr *my_addr, socklen_t addrlen);
```

- “bind”s a socket to a particular address/port combo — this is how a server claims its well-known port
- We tend to only use bind as a server — as a client, we tend not to care what our local port is (the OS picks one)
- We will often use INADDR_ANY to indicate that we want to accept a connection on any of the machine’s IPv4 addresses
  - slightly different for IPv6 — see “Jumping from IPv4 to IPv6” in Beej’s guide
- With getaddrinfo(): pass AI_PASSIVE in hints and NULL for node, and ai_addr comes back ready to bind

"Address already in use": SO_REUSEADDR

- You will likely hit this often: kill your server, restart it, and bind() fails with EADDRINUSE
- Why: after close(), TCP holds the port in TIME_WAIT (typically ~60s) to catch stray packets from the old connection
- The Fix: Adding SO_REUSEADDR tells the kernel "let me bind anyway" — standard practice for servers during development
- must be set after socket() but before bind() — it has no effect afterwards

```
int yes = 1;
if (setsockopt(fd, SOL_SOCKET, SO_REUSEADDR,
               &yes, sizeof(yes)) < 0) { perror("setsockopt"); /*...*/ }

bind(fd, ...);   // now succeeds even during TIME_WAIT
```

listen()

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

- Starts our socket “listening” (what a server would do), the OS will now queue incoming connection attempts on our behalf
- backlog is how many outstanding requests can be queued until we accept them
  - a small number like 10 is fine for the assignment
- listen() returns immediately — it doesn’t wait for anyone to connect (that’s accept’s job, next)

accept()

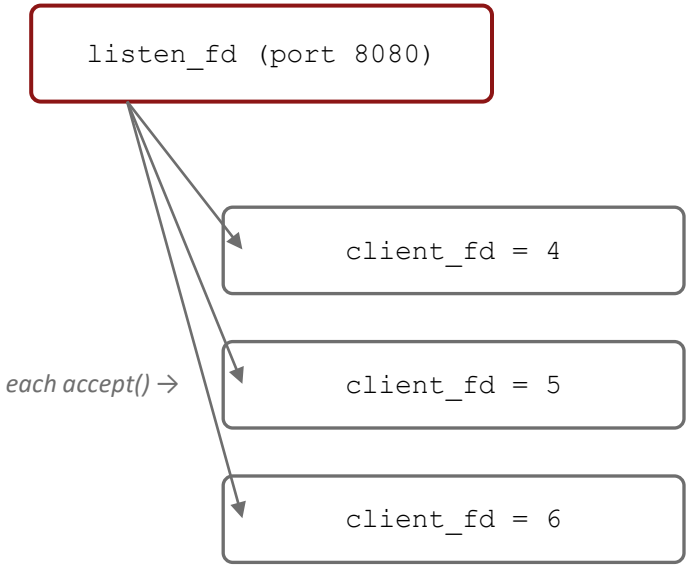
```
int accept(int sockfd, const struct sockaddr *addr, socklen_t *addrlen);
```

- Blocks until a client connects, then returns a file descriptor for that remote connection
- addr is filled in with the client's address — we'll use a struct sockaddr_storage, which is guaranteed large enough to hold any address type (or pass NULL if we don't care who connected)

```
struct sockaddr_storage {
    sa_family_t ss_family;    // address family
    // all this is padding, implementation specific, ignore it:
    char    __ss_pad1[_SS_PAD1SIZE];
    int64_t __ss_align;
    char    __ss_pad2[_SS_PAD2SIZE];
};
```

accept() returns a *new* fd

- The listening socket is a factory, not a phone line — accept() hands you a brand-new fd for each client connection
- The listening socket stays open and keeps listening; call accept() on it again for the next client
- All send()/recv() traffic for a client goes through that client's connection fd — never the listening fd
- This is how one server talks to many clients: one listening socket, N connection sockets
- common bug: recv() on the listening socket — it fails with ENOTCONN

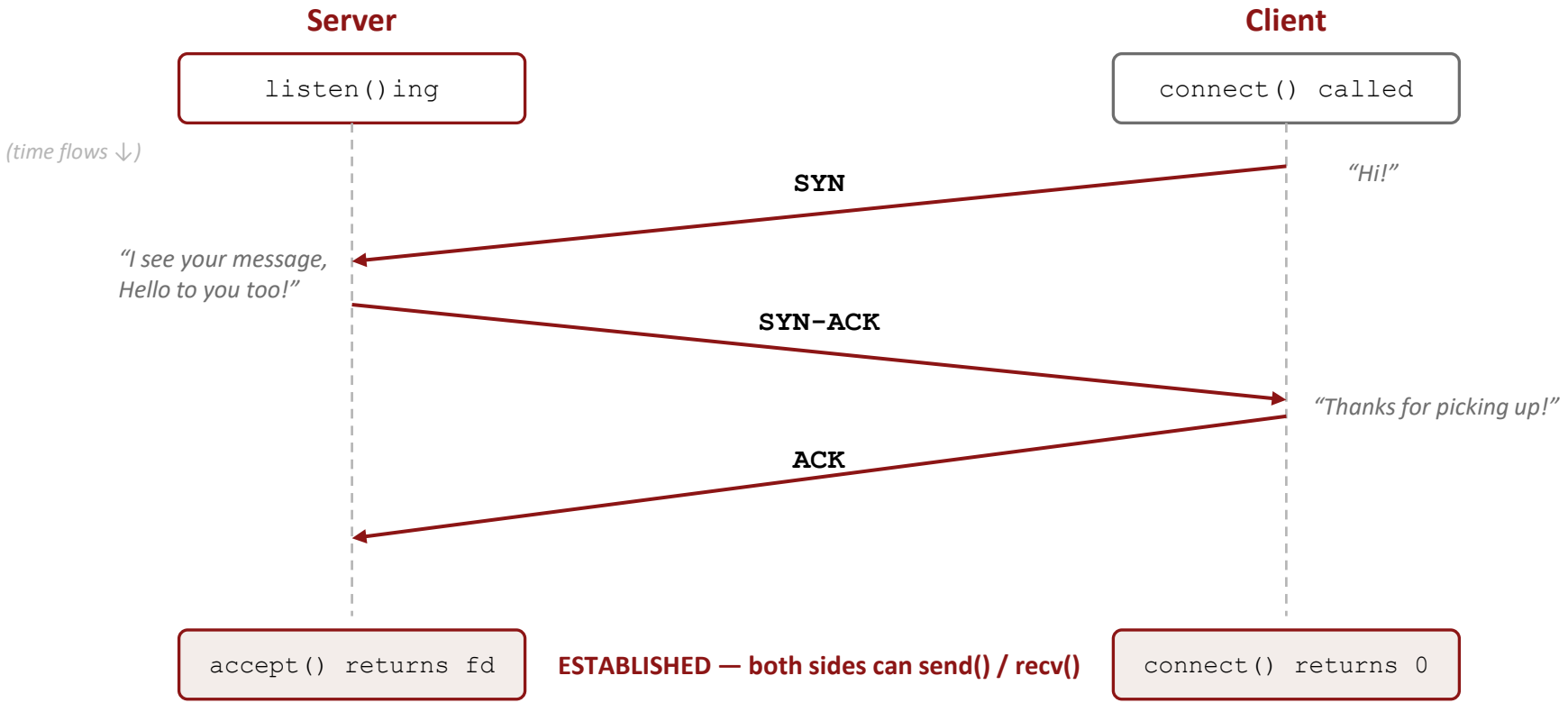


connect()

```
int connect(int sockfd, const struct sockaddr *serv_addr, socklen_t addrlen);
```

- The client's side of the story: connects our local socket to the remote address
- This is the call that performs the TCP three-way handshake with the server's listening socket
- No bind() needed, the OS automatically picks a local port for us
- On success, the fd is a live connection: send() and recv() away

The TCP Three-Way Handshake



socket() → bind() → listen() → **accept()** ⇄ **connect()** → send()/recv() → close()

send()

```
ssize_t send(int sockfd, const void *msg, int len, int flags);
```

- Returns how many bytes were actually sent — which may be less than we requested (which we'll have to handle — next slide)
- flags can be 0 by default
- While we could use write, we tend to use send instead since it lets us do more specific socket things (see the man page for flags)

Partial sends: the send_all() loop

- Under load or with large buffers, send() can accept fewer bytes than you asked — this is not an error
- You must loop, advancing your pointer by whatever was accepted, until everything is sent — write it once and reuse it

```
// Returns 0 on success, -1 on error.
int send_all(int fd, const void *buf, size_t len) {
    const char *position = buf;
    while (len > 0) {
        ssize_t sent = send(fd, position, len, 0);
        if (sent < 0) return -1;   // check errno; EINTR -> retry
        position += sent;
        len      -= sent;
    }
    return 0;
}
```

recv()

```
ssize_t recv(int sockfd, void *buf, int len, int flags);
```

- Returns how many bytes were received (no more than len)
- Returns <0 on error, 0 when the remote side has closed — checking for that 0 is how you know the conversation is over
- recv() hands you raw bytes, not a C string — no '\0' arrives over the wire; terminate manually before printf("%s") or any call expecting a C string
- How many bytes you get per call is up to the network — next slide

TCP is a byte stream, not messages

- If the client calls send() three times, the server will not necessarily see three recv() results — TCP preserves bytes and order, nothing else
- So a protocol needs framing: a rule for finding message boundaries — a delimiter (HTTP uses “\r\n\r\n”) or a length prefix (byte count first, then payload)
- loop on recv() until you have a complete message by your framing rule — never assume one recv() == one message

close() vs shutdown(): different jobs

close(fd)

- Releases the file descriptor — this is the mandatory cleanup step; nothing else frees the fd
- After it, this process can't touch the socket at all — both directions gone
- Begins orderly connection teardown (sends FIN); any unsent data is still delivered in the background
- Every fd you accept() or socket() needs exactly one

shutdown(fd, how)

- Acts on the connection, not the descriptor — the fd stays allocated (you still close() later)
- Can cut one direction only: SHUT_RD (0), SHUT_WR (1), SHUT_RDWR (2)
- SHUT_WR is the useful one: sends FIN now — "I'm done sending" — while you keep reading replies (a half-close)
- SHUT_RD is local bookkeeping only — the peer isn't told

They're complements, not alternatives: shutdown() is optional protocol signaling; close() is mandatory resource cleanup. Most programs only ever need close().

The half-close pattern: who actually uses shutdown()?

The classic use is the **client** in an EOF-delimited request/response protocol: half-close to say “request complete,” then read the full reply. The server usually needs only plain close().