Lecture 18: Sockets Programming

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

Google Forms

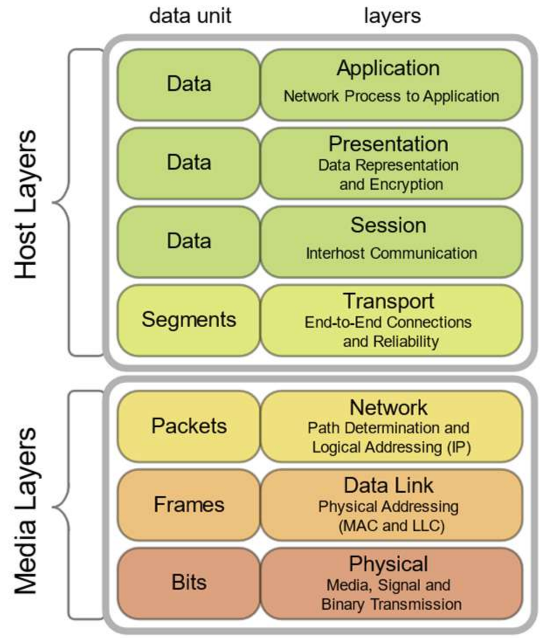
IntelliCopilot Survey:

https://forms.gle/vcyecuCTkPErHkL39

Attendance:

https://forms.gle/zg4kjyzfEsfpzp519

Quick Overview



TCP and UDP

- Both run on top of IP
- Both have a port number (16 bits)
  - Official port usage is assigned by IANA
  - Ports under 1024 are typically reserved (i.e., on the `myth` machines, you need special permission to bind to them)
  - Common ports include: 22 (SSH), 53 (DNS), 80 (HTTP), 443 (HTTPS)—see also https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml or `/etc/services`

TCP and UDP (continued)

- So, to connect to a remote server, we need both an IP address and a port number
  - Quick aside on IP addresses: IPv4 addresses are only 32 bits long, which only makes for about 4 billion total IPs, which we've fully allocated; IPv6 addresses, by contrast, are 128 bits long
    - IPv4 addresses are often written in dotted quad format of `192.168.1.1`
    - IPv6 addresses are a little more complicated, but can be written as `2607:f6d0:0:0:0:0:0:0` (which can also be written as `2607:f6d0::`)
    - Either way, IP addresses can be a little cumbersome to write, so we use DNS (domain name service) to map from domain names (e.g., `web.stanford.edu`) to IP addresses (e.g., `171.67.215.200`)

TCP and UDP (continued)

- Quick aside on client server model: for the rest of the lecture, we'll be implicitly referencing this model—the rough idea is that we have a server (imagine, e.g., Google) which serves data to one or more clients (imagine, e.g., people Googling)
- TCP and UDP both allow us to send arbitrary bytes over the network
- It is important that we send bytes in a way that both the client and server will understand
  - A protocol specifies how the bytes will be interpreted
  - IP and TCP/UDP level details specify that the network byte order should be big-endian (`myth` machines are little-endian)

TCP and UDP (continued)

- TCP provides a "reliable bytestream" abstraction (except in exceptional cases, the data will arrive correctly on the other side)
  - Useful for non-time critical applications (e.g., web servers (`HTTP` prior to `HTTP/3` runs over TCP), `ssh`, etc…)
- UDP provides an unreliable datagram abstraction (it's effectively just a userspace wrapper around IP, hence "User Datagram Protocol")
  - Useful for time critical applications, or applications that can deal with some data loss (e.g., video conferencing, online gaming, etc…)

SOCKET PROGRAMMING BASICS

`socket()`

- `int socket(int domain, int type, int protocol);`
- The domain specifies what type of socket we want—for this lecture, it will be one of `PF_INET` or `PF_INET6`
- 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., one of `IPPROTO_TCP` or `IPPROTO_UDP`, but we can use `0` since `SOCK_STREAM` means TCP, and it will figure it out)
- Returns a "file descriptor" on success and `<0` on error (setting errno as appropriate)

Detour: file descriptors

- You may encounter the phrase "everything is a file" when working in a Unix/Linux context
- File descriptors are one incarnation of this—a `FILE *` is a convenient wrapper around a file descriptor
  - A file descriptor is an integer that the OS hands to our process that we can use syscalls on to read/write data (e.g., `read`, `write`) or otherwise modify (e.g., `fcntl`)
  - We'll have the following file descriptors always by default: `0` (`stdin`), `1` (`stdout`), `2` (`stderr`)
- Note that we use file descriptors for both real files and for sockets (among other things)

Detour: error handling

- Many system calls (and wrapping C functions) can fail
- In C, we'll often see failure represented as a negative value, with `errno` (see `man errno`) set appropriately (`perror` will print the corresponding error message)
  - Basically every function today can fail in this manner
- In 107, we've mostly ignored this up until this point, but there are a few ways to handle this in C
  - Explicitly check every return value that might fail, write out the failure condition
  - Wrap functions in safe forms (e.g., the textbook creates `Write` from `write`)
  - Use macros to help simplify
  - `goto`s are often used for clean up, but given their potential for misuse, we won't cover them to closely here
  - On the (optional) sockets assignment, we'll provide a few options for error handling (which you should be doing)

Detour: `man` pages

- While, in general, we like to tell you to read the `man`page for the functions, the `man` pages for sockets programming tend to be comparatively more difficult to actually find and understand
- I would recommend using the fake `man` pages from https://beej.us/guide/bgnet/ and then consulting the real `man` pages later, as appropriate (and if necessary)

`bind()`

- “bind”s a socket to a particular address/port combo
- `int bind(int sockfd, struct sockaddr *my_addr, int addrlen);`
- Note, we tend to only use `bind` as a server (as a client, we tend not to actually care what our port is)

struct sockaddr

- `struct sockaddr` is the generic type for a socket address, but we'll use `struct sockaddr_in` or `struct sockaddr_in6` and cast to a `struct sockaddr`

```
struct sockaddr {
    unsigned short sa_family; // address family, AF_xxx
    char sa_data[14]; // 14 bytes of protocol address
};
struct sockaddr_in {
    short int sin_family; // Address family, AF_INET
    unsigned short int sin_port; // Port number
    struct in_addr sin_addr; // Internet address
    unsigned char sin_zero[8]; // Same size as struct sockaddr
};
struct in_addr {
    uint32_t s_addr; // that's a 32-bit int (4 bytes)
};
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]; // IPv6 address
};
```

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

- `aton` and `addr` only work for IPv4 addresses
- `int inet_aton(const char *cp, struct in_addr *inp);`
- `in_addr_t inet_addr(const char *cp);`
- `cp` is a string of a dotted quad IP address
- `int inet_pton(int af, const char *src, void *dst);`

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);
```

- 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
};
```

bind()

- `int bind(int sockfd, struct sockaddr *my_addr, int addrlen);`
- Binds our socket to the address and port specified by `my_addr`
- We will often use `INADDR_ANY` to indicate that we want to accept any IPv4 connection (slightly different for IPv6, see “Jumping from IPv4 to IPv6” on Beej’s guide)

`listen()`

- `int listen(int sockfd, int backlog);`
- Starts our socket "listening" (what a server would do)
- `backlog` is how many outstanding requests can be queued until we `accept` them

accept()

- `int accept(int sockfd, struct sockaddr *addr, socklen_t *addrlen);`
- Returns a file descriptor for a remote connection
- We'll use a `struct sockaddr_storage` (guaranteed large enough to store any address) for the address

```
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];
};
```

`connect()`

- `int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);`
- Useful for the client, connects our local socket to the remote address

`send()`

- `int send(int sockfd, const void *msg, int len, int flags);`
- Returns how many bytes were actually sent (may be less than we requested, which we'll have to handle)
- `flags` can be 0 by default
- Note that while we could use `write`, we tend to use `send` instead since it lets us to more specific socket things (see the `man` page for `flags`)

`recv()`

- `int 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 remote side has closed

`close()`

- `close(sockfd);`
- Prevents any further reads or writes to the socket, the remote peer will receive an error on trying to read or write
- Also, marks the fd as usable again (no longer counts toward our per-process limit)

`shutdown()`

- `int shutdown(int sockfd, int how);`
- Note that you will still have to `close` eventually

how	Effect
0	Further receives are disallowed
1	Further sends are disallowed
2	Further sends and receives are disallowed (like close())

CODE DEMO

Handling multiple clients

- We may not get to this in lecture, but you should investigate using `select()` and/or `poll()` (or `epoll` if you want to get really fancy) for the assignment