

# Sockets and DNS

## CSE 333

**Instructor:** Alex Sanchez-Stern

**Teaching Assistants:**

Audrey Seo

Deeksha Vatswani

Derek de Leuw

Katie Gilchrist

# Administrivia

- ❖ HW3 due tomorrow night, 11pm
  - Plus late days if needed and you have them remaining
    - You can check Canvas to see how many late days you have left
  - Don't forget about comments on your helper functions
    - Many people miss this and lose points
  - Any last-minute questions? observations?
- ❖ Exercise 15 due Monday
  - Client-side TCP connection

# Lecture Outline

- ❖ **Sockets API**
  - Sockets Overview
  - Network Addresses
  - API Functions
- ❖ DNS

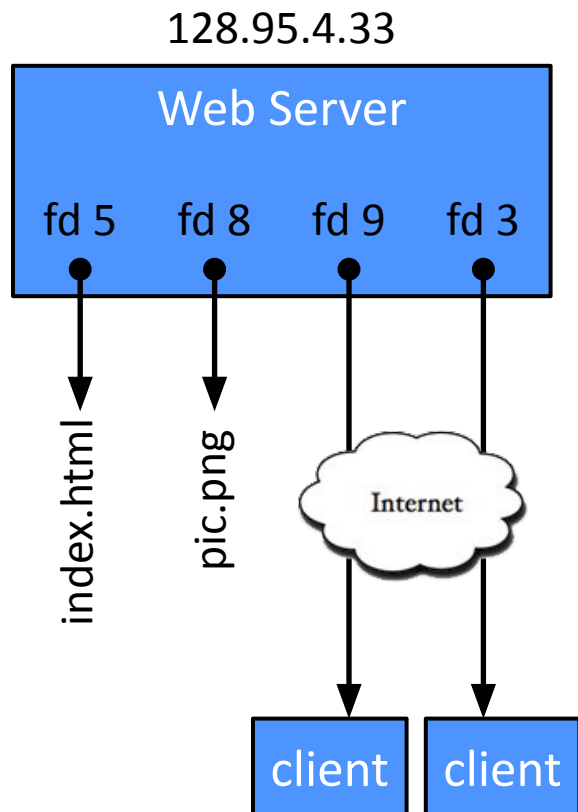
# Files and File Descriptors

- ❖ Remember `open()`, `read()`, `write()`, and `close()` ?
  - POSIX system calls for interacting with files
  - `open()` returns a `file descriptor`
    - An integer that represents an open file
    - This file descriptor is then passed to `read()`, `write()`, and `close()`
  - Inside the OS, the file descriptor is used to index into a table that keeps track of any OS-level state associated with the file, such as the file position

# Networks and Sockets

- ❖ UNIX likes to make *all* I/O look like file I/O
  - You use `read()` and `write()` to communicate with remote computers over the network!
  - A file descriptor used for network communications is called a `socket`
  - Just like with files:
    - Your program can have multiple network channels open at once
    - You need to pass a file descriptor to `read()` and `write()` to let the OS know which network channel to use

# File Descriptor Table



OS's File Descriptor Table for the Process

File Descriptor	Type	Connection
0	pipe	stdin (console)
1	pipe	stdout (console)
2	pipe	stderr (console)
3	TCP socket	local: 128.95.4.33:80 remote: 44.1.19.32:7113
5	file	index.html
8	file	pic.png
9	TCP socket	local: 128.95.4.33:80 remote: 102.12.3.4:5544

# Types of Sockets

## ❖ Stream sockets

- For connection-oriented, point-to-point, reliable byte streams
  - Using TCP, SCTP, or other stream transports

## ❖ Datagram sockets

- For connection-less, one-to-many, unreliable packets
  - Using UDP or other packet transports

## ❖ Raw sockets

- For layer-3 communication (raw IP packet manipulation)

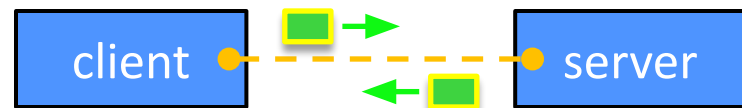
# Stream Sockets

- ❖ Typically used for client-server communications
  - **Client**: An application that establishes a connection to a server
  - **Server**: An application that receives connections from clients
  - Can also be used for other forms of communication like peer-to-peer

1) Establish connection:



2) Communicate:



3) Close connection:

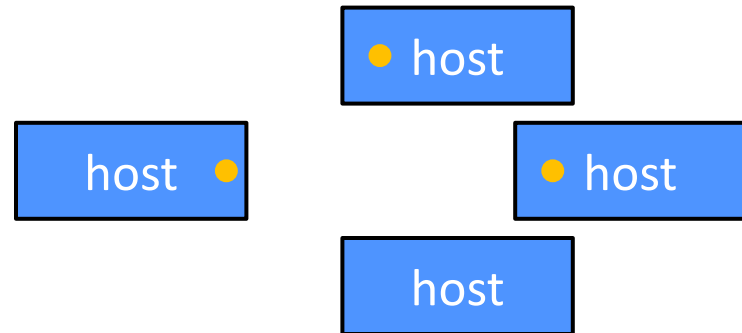




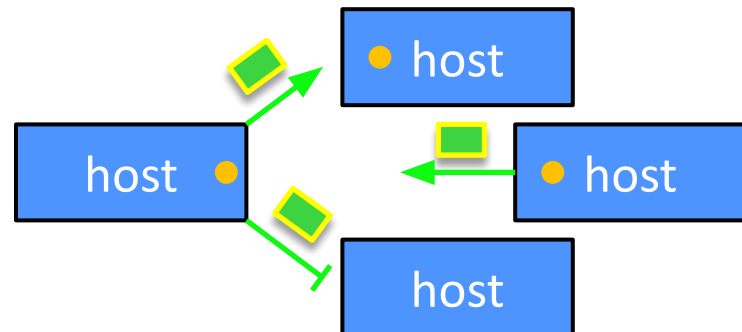
# Datagram Sockets

- ❖ Often used as a building block
  - No flow control, ordering, or reliability, so used less frequently

1) Create sockets:



2) Communicate:



# The Sockets API

- ❖ Berkeley sockets originated in 4.2BSD Unix (1983)
  - It is the standard API for network programming
    - Available on most OSs
  - Written in C
  
- ❖ POSIX Socket API
  - A slight update of the Berkeley sockets API
    - A few functions were deprecated or replaced
    - Better support for multi-threading was added

# Lecture Outline

- ❖ **Sockets API**
  - Sockets Overview
  - **Network Addresses**
  - API Functions
- ❖ DNS


# IPv4 Network Addresses

- ❖ An IPv4 address is a **4-byte** tuple
  - For humans, written in “dotted-decimal notation”
  - *e.g.* 128.95.4.1 (80 : 5f : 04 : 01 in hex)
  
- ❖ IPv4 address exhaustion
  - There are  $2^{32} \approx 4.3$  billion IPv4 addresses
  - There are  $\approx 8$  billion people in the world (July 2024)
  - There are  $\approx 30$  billion internet connected devices

# IPv6 Network Addresses

- ❖ An IPv6 address is a **16-byte** tuple

- *e.g.* 2d01:00b8:f188:0000:0000:0000:0000:1f33




Written as eight “hextets”  
(groups of four hex digits),  
separated by colons

# IPv6 Network Addresses

- ❖ An IPv6 address is a **16-byte** tuple

- *e.g.* 2d01:00b8:f188:0000:0000:0000:0000:1f33



2d01:b8:f188:0000:0000:0000:0000:1f33

Leading zeros within a group  
can be omitted

# IPv6 Network Addresses

- ❖ An IPv6 address is a **16-byte** tuple


- *e.g.* 2d01:00b8:f188:0000:0000:0000:0000:1f33

2d01:b8:f188:0000:0000:0000:0000:1f33

2d01:b8:f188::1f33

Consecutive groups of zeros can be turned into a double colon (can only be done for one group).

# IPv6 Network Addresses

- ❖ An IPv6 address is a **16-byte** tuple
  - *e.g.* 2d01:00b8:f188:0000:0000:0000:0000:1f33
  - Transition is still ongoing
    - When writing network code, we need to support both
    - This unfortunately makes network programming more of a headache
  - For  compatibility, IPv4 addresses can be mapped to IPv6 addresses

128.95.4.1

::ffff:128.95.4.1

::ffff:805f:401

Decimal	Hexadecimal
128	80
95	5f
4	04
1	01

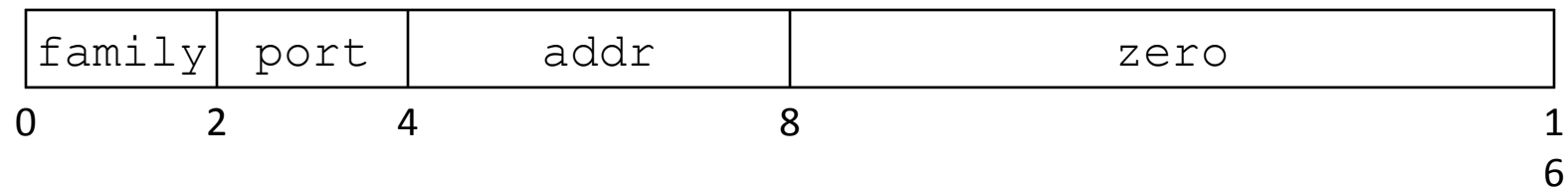


# IPv4 Address Structures

```
// IPv4 4-byte address
struct in_addr {
    uint32_t s_addr;           // Address in network byte order
};                             // (big endian)

// An IPv4-specific address structure
struct sockaddr_in {
    sa_family_t    sin_family; // Address family: AF_INET
    in_port_t      sin_port;   // Port in network byte order
    struct in_addr sin_addr;    // IPv4 address
    unsigned char  sin_zero[8]; // Pad out to 16 bytes
};
```

struct sockaddr\_in:



# Working with Socket Addresses

- ❖ How to handle both IPv4 and IPv6?
  - Use C structs for each, but make them somewhat similar
  - Use defined constants to differentiate when to use each: `AF_INET` for IPv4 and `AF_INET6` for IPv6

```
// IPv4 4-byte address
struct in_addr {
    uint32_t s_addr;
};

// An IPv4 address structure
struct sockaddr_in {
    sa_family_t    sin_family;
    in_port_t      sin_port;
    struct in_addr sin_addr;
    unsigned char  sin_zero[8];
};
```

```
// IPv6 16-byte address
struct in6_addr {
    uint8_t s6_addr[16]
};

// An 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;
    uint32_t       sin6_scope_id;
};
```

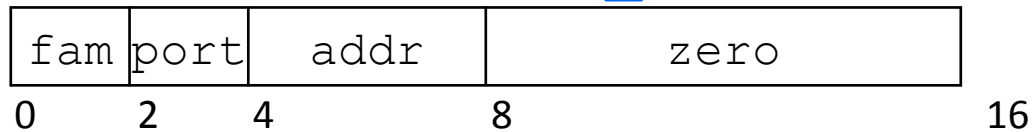
# Generic Address Structures

- ❖ One struct defined for pointers to both

```
// A mostly-protocol-independent address structure.  
// Pointer to this is parameter type for socket system calls.  
struct sockaddr {  
    sa_family_t sa_family;    // Address family (AF_* constants)  
    char        sa_data[14]; // Socket address (size varies  
                                // according to socket domain)  
};
```

# Generic Address Structures

`struct sockaddr_in:`



`struct sockaddr_in6:`



```
int get_port(struct sockaddr* addr) {  
    return ((struct sockaddr_in*) addr)->port;  
}
```

```
int process_addr(struct sockaddr* addr) {  
    if (addr->sa_family == AF_INET) {  
        ...  
    } else if (addr->sa_family == AF_INET6) {  
        ...  
    }  
}
```

# Generic Address Structures

- ❖ `struct sockaddr` isn't actually big enough to hold IPv6 addresses!
  - Only generic when used as pointer

```
// A mostly-protocol-independent address structure.  
// Pointer to this is parameter type for socket system calls.  
struct sockaddr {  
    sa_family_t sa_family;    // Address family (AF_* constants)  
    char        sa_data[14]; // Socket address (size varies  
                                // according to socket domain)  
};
```

IPv6 addresses  
are 16 bytes!

# Generic Address Structures

- ❖ For storing addresses generically, we have `struct sockaddr_storage`

```
// A mostly-protocol-independent address structure.
// Pointer to this is parameter type for socket system calls.
struct sockaddr {
    sa_family_t sa_family;    // Address family (AF_* constants)
    char        sa_data[14]; // Socket address (size varies
                             // according to socket domain)
};

// A structure big enough to hold either IPv4 or IPv6 structs
struct sockaddr_storage {
    sa_family_t ss_family;    // Address family

    // padding and alignment; don't worry about the details
    char __ss_pad1[_SS_PAD1SIZE];
    int64_t __ss_align;
    char __ss_pad2[_SS_PAD2SIZE];
};
```

- ❖ Commonly create `struct sockaddr_storage`, then pass pointer cast as `struct sockaddr*` to **`connect()`**

# Lecture Outline

## ❖ Sockets API

- Sockets Overview
- Network Addresses
- **API Functions**

## ❖ DNS

# Working with Socket Addresses

- ❖ Structures, constants, and helper functions available in `#include <arpa/inet.h>`
- ❖ This is a C API: no string objects, exceptions, or references
- ❖ Addresses stored in **network byte order** (big endian)
- ❖ Converting between host and network byte orders:
  - `uint32_t htonl(uint32_t hostlong);`
  - `uint32_t ntohl(uint32_t netlong);`
    - 'h' for host byte order and 'n' for network byte order
    - Also versions with 's' for short (`uint16_t` instead)



# Address Conversion

- ❖ `int inet_pton(int af, const char* src, void* dst);`
  - Converts human-readable c-string representation (“presentation”) to network byte ordered address
  - Returns 1 (success), 0 (bad `src`), or -1 (error)

genaddr.cc

```
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
    struct sockaddr_in sa;      // IPv4
    struct sockaddr_in6 sa6;    // IPv6

    // IPv4 string to sockaddr_in (192.0.2.1 = C0:00:02:01).
    inet_pton(AF_INET, "192.0.2.1", &(sa.sin_addr));

    // IPv6 string to sockaddr_in6.
    inet_pton(AF_INET6, "2001:db8:63b3:1::3490", &(sa6.sin6_addr));

    return EXIT_SUCCESS;
}
```

# Address Conversion

❖ `const char* inet_ntop(int af, const void* src, char* dst, socklen_t size);`

- Converts network addr in `src` into buffer `dst` of size `size`  
[genstring.cc](#)

```
#include <stdlib.h>
#include <arpa/inet.h>

int main(int argc, char **argv) {
    struct sockaddr_in6 sa6;           // IPv6
    char astring[INET6_ADDRSTRLEN];    // IPv6

    // IPv6 string to sockaddr_in6.
    inet_pton(AF_INET6, "2001:0db8:63b3:1::3490", &(sa6.sin6_addr));

    // sockaddr_in6 to IPv6 string.
    inet_ntop(AF_INET6, &(sa6.sin6_addr), astring, INET6_ADDRSTRLEN);
    std::cout << astring << std::endl;

    return EXIT_SUCCESS;
}
```

```
> ./genstring
2001:db8:63b3:1::3490
```

# Lecture Outline

- ❖ Sockets API
  - Sockets Overview
  - Network Addresses
  - API functions
- ❖ **DNS**

# Domain Name System

- ❖ People tend to use domain names like “www.google.com”, not IP addresses
  - The Sockets API lets you convert between the two
  - It’s a complicated process, though:
    - A given domain name can have many IP addresses
      - An IP address will reverse map into at most one domain name
    - A DNS lookup may require interacting with many DNS servers
- ❖ You can use the Linux program “dig” to explore DNS
  - `dig @server name type (+short)`
    - `server`: specific name server to query (optional)
    - `type`: A (IPv4), AAAA (IPv6), ANY (includes all types)

# Dig example

A few other sections  
not important to this  
class

```
> dig www.google.com A
```

```
...
```

```
;; ANSWER SECTION:
```

```
www.google.com. 146 IN A 142.250.217.68
```

```
;; Query time: 19 msec
```

```
;; SERVER: 8.8.8.8#53(8.8.8.8) (UDP)
```

```
...
```

```
> dig www.google.com AAAA
```

```
...
```

```
;; ANSWER SECTION:
```

```
www.google.com. 34 IN AAAA 2607:f8b0:400a:804::2004
```

```
;; Query time: 23 msec
```

```
;; SERVER: 8.8.8.8#53(8.8.8.8) (UDP)
```

```
...
```

# DNS Hierarchy

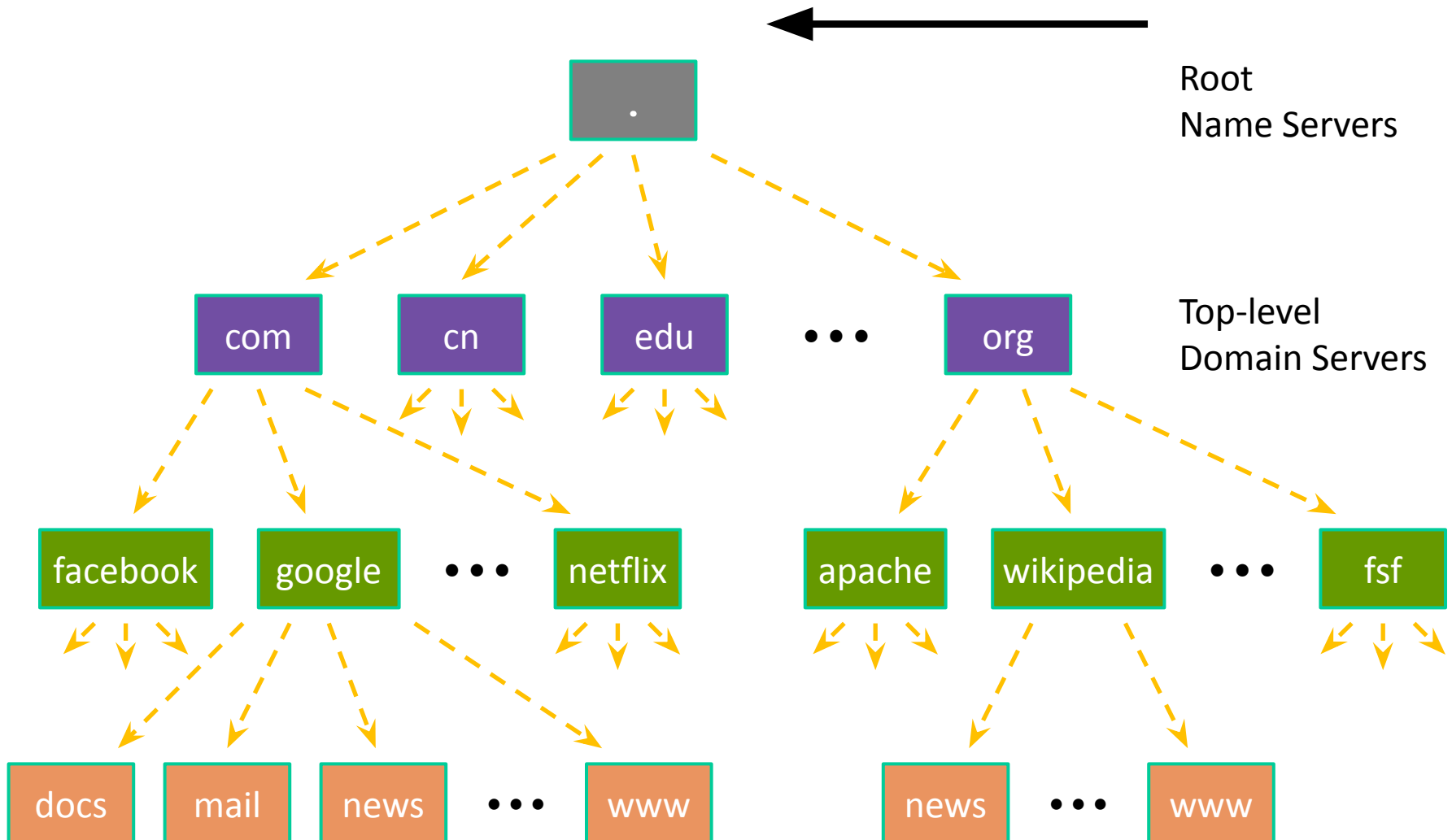
- ❖ The dots in a web address actually have a meaning!
  - Each web address component is a different “level” of DNS
  - Read from right to left

mail.google.com



# DNS Hierarchy

mail.google.com



# Resolving DNS Names

- ❖ The POSIX way is to use **getaddrinfo** ()
  - A complicated system call found in `#include <netdb.h>`
  - ```
int getaddrinfo(const char* hostname,
                const char* service,
                const struct addrinfo* hints,
                struct addrinfo** res);
```
  - Returns 0 on success; returns negative number on failure



# Resolving DNS Names

```
int getaddrinfo(const char* hostname,
               const char* service,
               const struct addrinfo* hints,
               struct addrinfo** res);
```

- ❖ **hostname**: String representation for host: DNS name or IP address
- ❖ **service**: String representation for port/service. Can be either:
  - The port number as a string
  - A “service name” which will be looked up in a special /etc/services file to get a port
  - nullptr to allow any port
- ❖ **hints**: a structure with constraints you want respected
- ❖ **res**: an output parameter for the list of results
  - Represented as an `struct addrinfo*`
  - Has a `next` pointer and acts as a linked list in the case of multiple results

# Resolving DNS Names

- ❖ If `getaddrinfo()` returns a negative value (error), pass the return value to `gai_strerror()` to get a c-string corresponding to the error.

- `const char *gai_strerror(int errcode);`

- ❖ Free the `struct addrinfo` list when you're done using it with `freeaddrinfo()`

- `void freeaddrinfo(struct addrinfo *res);`

# struct addrinfo

- ❖ The addrinfo struct can be a bit complicated.
  - No need to memorize it! You can always look it up, we won't test you on it, etc.

```
struct addrinfo {  
    int      ai_flags;           // additional flags  
    int      ai_family;         // AF_INET, AF_INET6, AF_UNSPEC  
    int      ai_socktype;       // SOCK_STREAM, SOCK_DGRAM, 0  
    int      ai_protocol;       // IPPROTO_TCP, IPPROTO_UDP, 0  
    size_t   ai_addrlen;        // length of socket addr in bytes  
    struct sockaddr* ai_addr;    // pointer to socket addr  
    char*     ai_canonname;      // canonical name  
    struct addrinfo* ai_next;    // can form a linked list  
};
```

# getaddrinfo

See:

`dnsresolve.cc`

<https://courses.cs.washington.edu/courses/cse333/25su/lecture/18-network-sockets+dns-example>

# Extra Exercise #1

- ❖ Write a program that:
  - Reads DNS names, one per line, from `stdin`
  - Translates each name to one or more IP addresses
  - Prints out each IP address to `stdout`, one per line