

Systems Programming

Sockets, DNS

Instructors:

Amber Hu

Justin Hsia

Teaching Assistants:

Ally Tribble

Blake Diaz

Connor Olson

Grace Zhou

Jackson Kent

Janani Raghavan

Jen Xu

Jessie Sun

Jonathan Nister

Mendel Carroll

Rose Maresh

Violet Monserate

Relevant Course Information

- ❖ Exercise 15 & 16 will be released on Friday
 - ex15 due Monday (3/2), ex16 due next Wednesday (3/4)
 - Primarily adapting existing network programming code
- ❖ Homework 3 is due tomorrow (2/26)
 - HW 2 grades have been released!
 - Usual reminder: **don't forget to tag, clone elsewhere, and recompile**
 - Will need to copy `libhw1.a` and `libhw2.a`
 - Copy from `solution_binaries` first, then test your HW 1&2 libs
- ❖ Homework 4 will be released on Friday (2/27)

Mid-Quarter Survey Feedback

❖ Lecture

- Want more polls
- Disconnect between lectures and exams
- Summary slide at the end would be nice

❖ Section

- Problems could be harder, closer to exam-style
- More live coding, code examples

❖ Exercises

- Ex 7 (POSIX directories) and Ex 9 (1st C++ Vector)
 - Many comments: some positive, some negative
- Exercise policies are harsh (no late, no drops)
 - Good news: You now get 2 exercise drops!

Lecture Outline

- ❖ **Network Programming**
 - Sockets API
 - Network Addresses
 - DNS Lookup

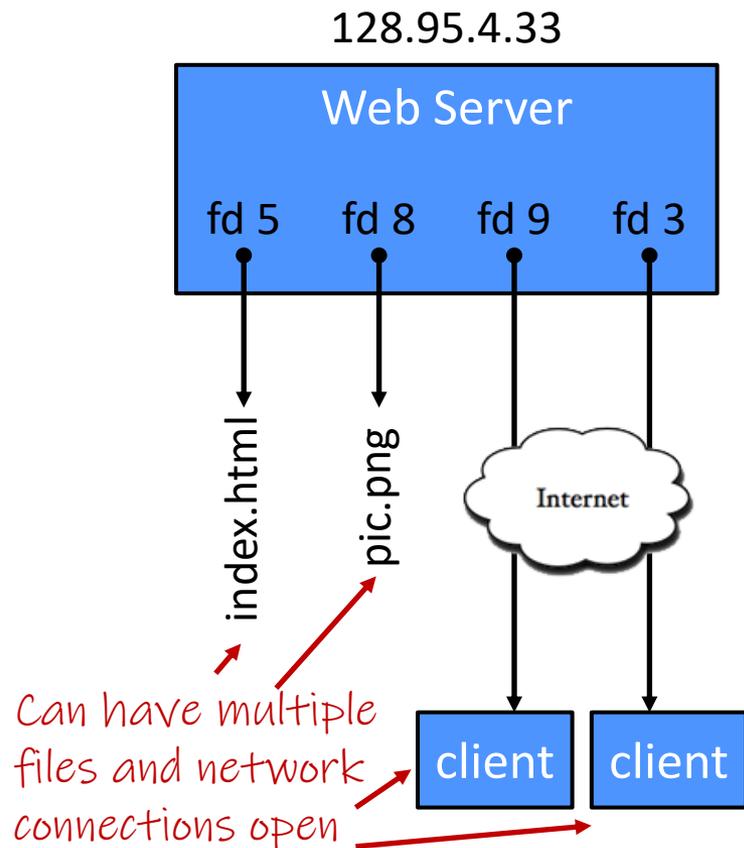
Files and File Descriptors

- ❖ 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 use 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)

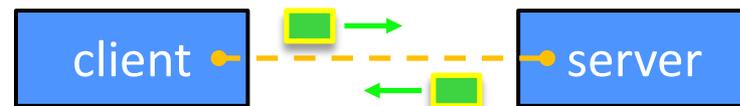
Stream Sockets

- ❖ 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:



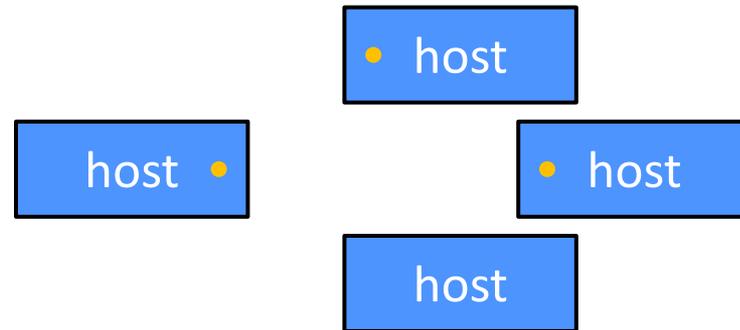
3) Close connection:



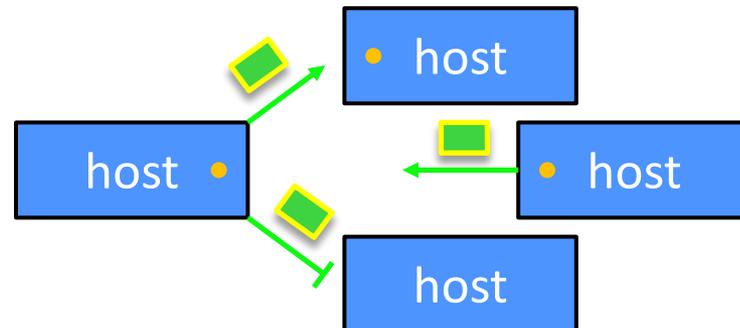
Datagram Sockets

- ❖ Often used as a building block
 - No flow control, ordering, or reliability, so used less frequently
 - *e.g.*, online gaming or DNS lookups

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

Socket API: Client TCP Connection

- ❖ We'll start by looking at the API from the point of view of a client connecting to a server over TCP

- ❖ There are five steps:
 - 1) Figure out the IP address and port to which to connect
 - 2) Create a socket
 - 3) Connect the socket to the remote server
 - 4) **read**() and **write**() data using the socket
 - 5) Close the socket

Step 1: Figure Out IP Address and Port

- ❖ Several parts:
 - Network addresses
 - Data structures for address info *C data structures* 😞
 - DNS (Domain Name System) – finding IP addresses

IPv4 Network Addresses

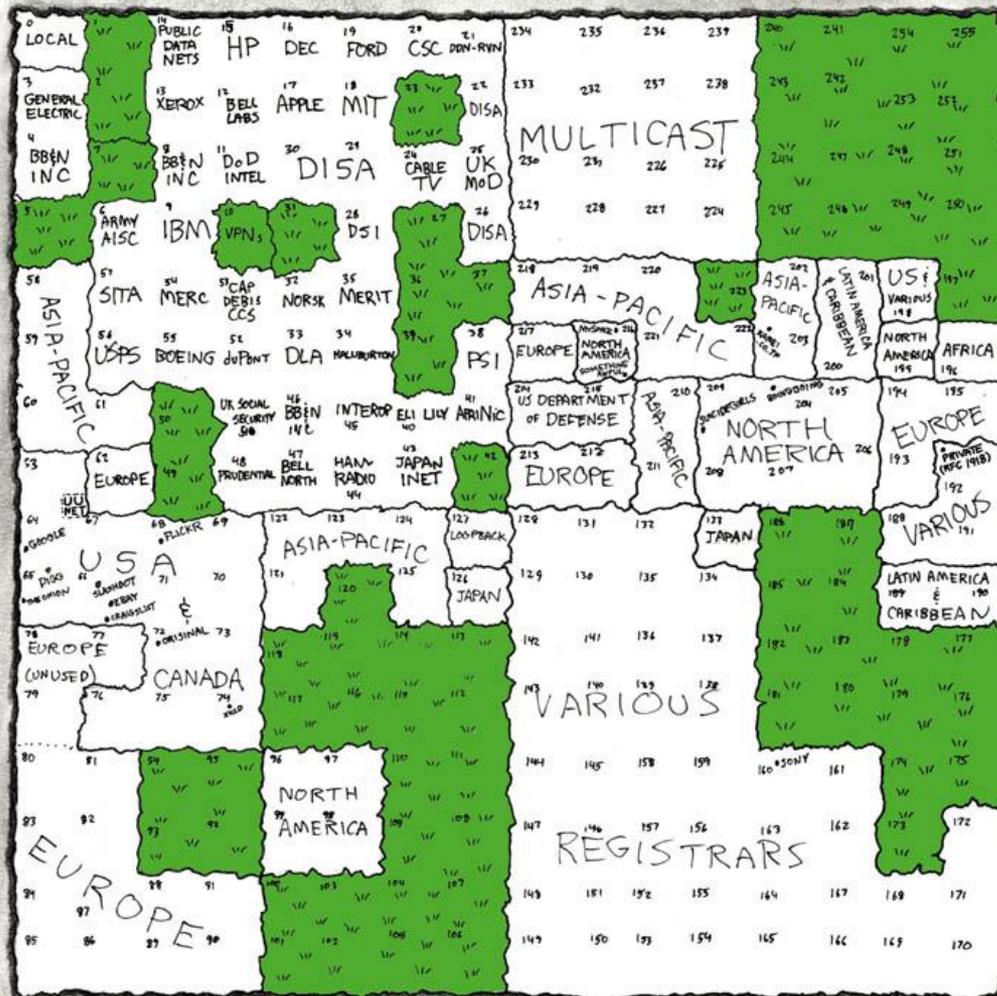
- ❖ An IPv4 address is a **4-byte** tuple
 - How many possible IPv4 addresses are there?
 - For humans, written in “dotted-decimal notation”
 - *e.g.*, 128.95.4.1 (80:5f:04:01 in hex)
- ❖ IPv4 address exhaustion
 - There are $2^{32} \approx 4.3$ billion IPv4 addresses
 - There are ≈ 8.29 billion people in the world (February 2026)

IPv6 Network Addresses

- ❖ An IPv6 address is a **16-byte** tuple
 - Typically written in “hextets” (groups of 4 hex digits)
 - Can omit leading zeros in hextets
 - Double-colon replaces consecutive sections of zeros
 - *e.g.*, 2d01:0db8:f188:0000:0000:0000:0000:1f33
 - Shorthand: 2d01:db8:f188::1f33
 - Which address does ::1 represent?
 - Transition is still ongoing
 - IPv4-mapped IPv6 addresses
 - 128.95.4.1 mapped to ::ffff:128.95.4.1 or ::ffff:805f:401
 - This unfortunately makes network programming more of a headache
☹

Aside: IP Address Allocation

MAP OF THE INTERNET
THE IPv4 SPACE, 2006



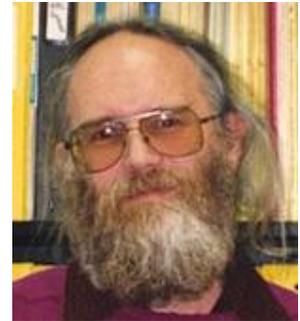
❖ This map is outdated (2006), as all IPv4 addresses have been allocated, but what interesting observations can you make?

- Geographic regions?
- Companies?

<https://xkcd.com/195/>

Aside: IP Address Allocation

- ❖ Global IP address allocation (among other things) is overseen by the **Internet Assigned Numbers Authority (IANA)**
 - “Currently it is a function of ICANN, a nonprofit private American corporation established in 1998 primarily for this purpose under a United States Department of Commerce contract. Before it, IANA was administered principally by Jon Postel at [USC], under a contract... with the United States Department of Defense.”
- ❖ Does this make sense? Is this fair?
 - Historically, it does (Internet “born” in the US)
 - Probably not entirely fair though – what values and priorities are encoded in this allocation?



Computing Standards and Protocols

- ❖ We've seen tons of these! Many more exist!
 - ASCII, IEEE 754 (Floating Point), POSIX, IP, TCP/UDP, HTTP, etc.
 - These have *profound* and *long-lasting* effects
- ❖ **Standards always encode the priorities of their creators into data**
 - *e.g.*, ASCII prioritizes English and memory efficiency
 - *e.g.*, IP addresses allocated with a very US-centric view, often granting larger-than-necessary swaths to the “big players” of the time
- ❖ Who was in the room when it happened? (*i.e.*, creation)
- ❖ Who has a seat at the table? (*i.e.*, maintenance)

Linux Socket Addresses

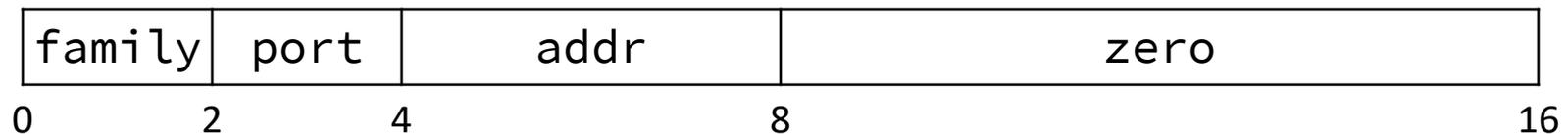
- ❖ Structures, constants, and helper functions available in `#include <arpa/inet.h>`
- ❖ Addresses stored in **network byte order** (big endian)
- ❖ Converting between host and network byte orders:
 - `uint32_t htonl(uint32_t hostlong);`
 - `uint32_t ntohl(uint32_t netlong);`
 - 'h' for host byte order and 'n' for network byte order
 - Also versions with 's' for short (`uint16_t` instead)
- ❖ 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 Address Structures

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

// 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`:





Poll Everywhere

pollev.com/cse333a

What will the first 4 bytes of the struct sockaddr_in be?

- ❖ Represents a socket connected to 198.35.26.96 (c6:23:1a:60) on port 80 (0x50) stored on a little-endian machine
 - AF_INET = 2

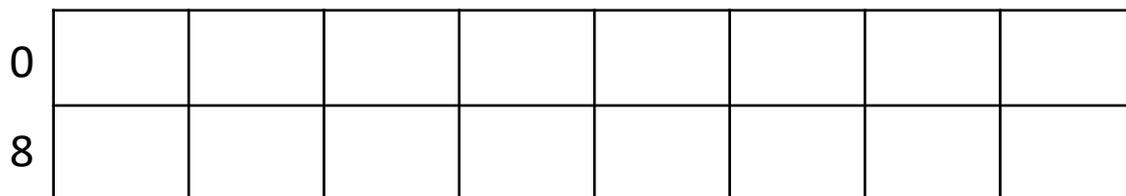
A. 0x 00 02 00 50

B. 0x 00 02 50 00

C. 0x 02 00 00 50

D. 0x 02 00 50 00

E. We're lost...

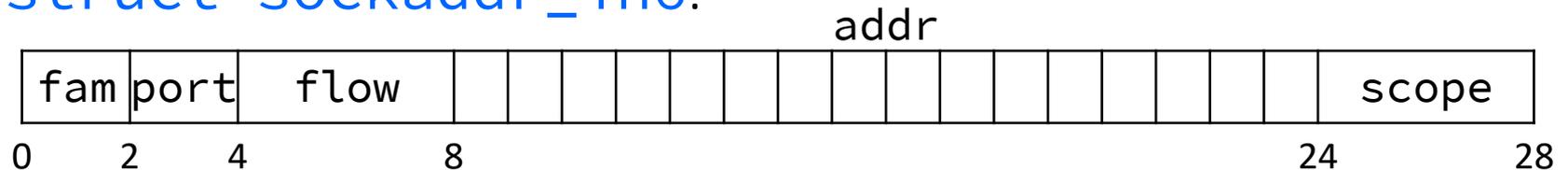


IPv6 Address Structures

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

// An IPv6-specific address structure
struct sockaddr_in6 {
    sa_family_t    sin6_family;    // Address family: AF_INET6
    in_port_t      sin6_port;      // Port number
    uint32_t       sin6_flowinfo;  // IPv6 flow information
    struct in6_addr sin6_addr;     // IPv6 address
    uint32_t       sin6_scope_id;  // Scope ID
};
```

`struct sockaddr_in6:`



Generic Address Structures

```
// 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()`

Address Conversion

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

```
#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;
}
```

genaddr.cc

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`
 - Returns `dst` on success; `NULL` on error

```
#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;
}
```

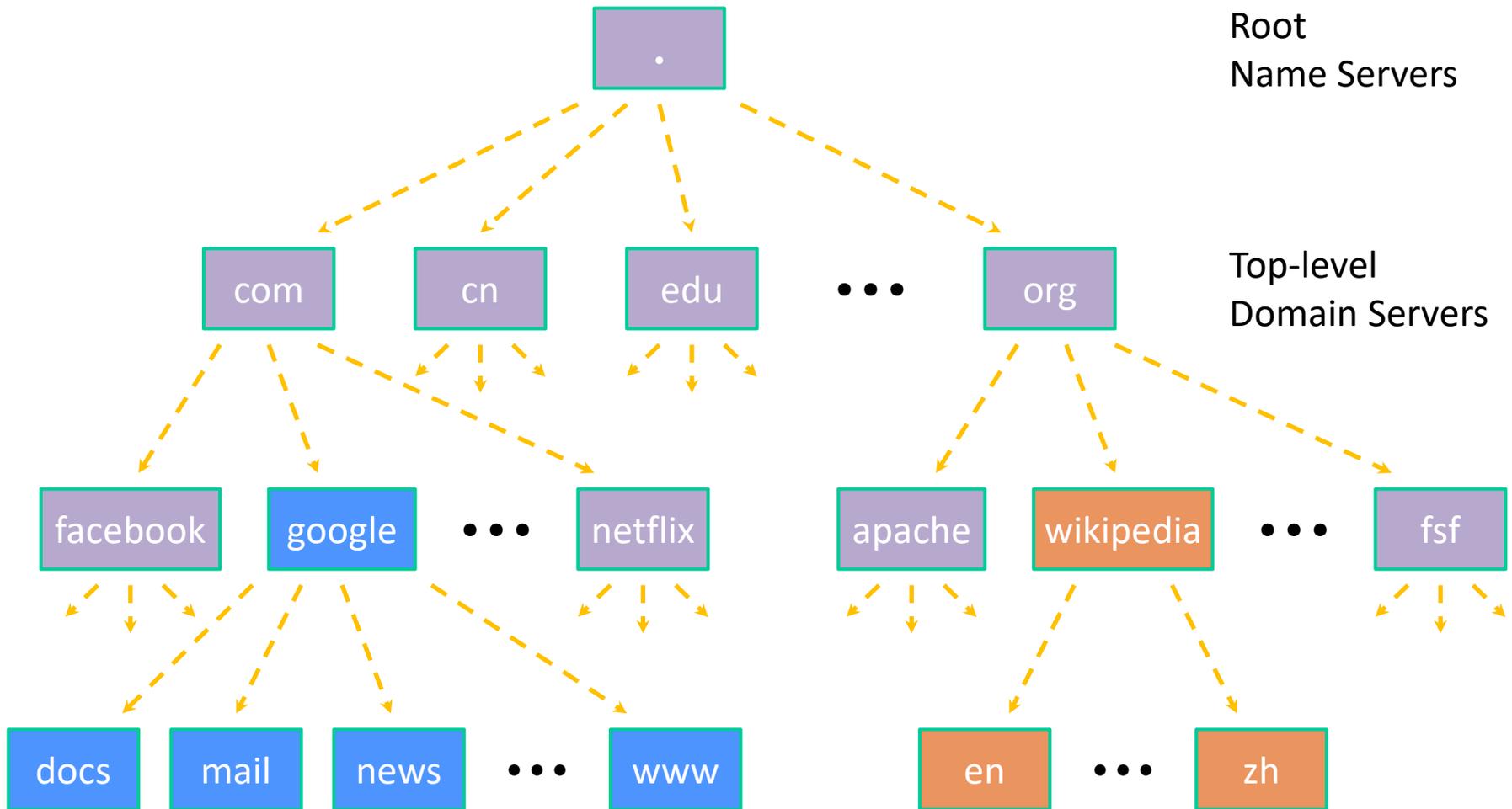
genstring.cc

Domain Name System

- ❖ People tend to use DNS names, not IP addresses
 - The Sockets API lets you convert between the two
 - It's a complicated process, though:
 - A given DNS name can have many IP addresses
 - Many different IP addresses can map to the same DNS name
 - An IP address will reverse map into at most one DNS name
 - A DNS lookup may require interacting with many DNS servers

- ❖ You can use the Linux program “dig” to explore DNS
 - `dig name type (+short)`
 - type: A (IPv4), AAAA (IPv6), ANY (includes all types)
 - Can specify a nameserver to query with `@server`

DNS Hierarchy



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);
```

- Tell **getaddrinfo()** which host and port you want resolved
 - String representation for host: DNS name or IP address
- Set up a “hints” structure with constraints you want respected
 - Filters out results you’re not looking for
- **getaddrinfo()** gives you a list of results packed into an “addrinfo” structure/linked list
 - Returns 0 on success; returns *negative number* on failure
- Free the `struct addrinfo` later using **freeaddrinfo()**

getaddrinfo

❖ **getaddrinfo** () arguments:

- hostname – domain name or IP address string
- service – port # (e.g., "80") or service name (e.g., "http") or `NULL/nullptr`
- hints – filter results

```
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  
};
```

DNS Lookup Procedure

```
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
};
```

- 1) Create a `struct addrinfo` hints
- 2) Zero out hints for “defaults”
- 3) Set specific fields of hints as desired
- 4) Call `getaddrinfo()` using `&hints`
- 5) Resulting linked list `*res` will have all fields appropriately set

❖ See [dnsresolve.cc](#)