ATTRIBUTION

• These slides are released under an Attribution-NonCommercial-ShareAlike 3.0 Unported (CC BY-NC-SA 3.0) Creative Commons license
• These slides incorporate material from:
  • Computer Networks: A Systems Approach, 5e, by Peterson and Davie
ANNOUNCEMENTS

Homework 1 deadline pushed to Wednesday at 5pm
• Some updates on the FAQ, posted to the class website
Reading due: Donahoo and Calvert, Chapter 2
Start working on Project 1!
Outline

1. Homework 1 strategy
2. Internetworking overview
3. IP and IP addresses
4. DNS and naming
5. DNS API
CHALLENGE: READ IN A SIMPLE CALCULATOR

INSTRUCTION SET

- Instructions separated by <CR> <LF> two-byte delimiter
- Groups of instructions separated by a pair of <CR><LF> delimiters
- Goal:
  - Working with delimiters in isolation, without also needing to work with reading data from the network
- Today:
  - Approach most similar to what will work for your web server
BASIC INPUT/OUTPUT IN C/C++

- **ptr**: the buffer to read data into (fread); the data you want to write out (fwrite)
- **size**: how big of a data item you want to read/write
- **nmemb**: how many data items to read/write
- **stream**: the I/O stream you want to read or write to/from

**How to put these together to read in a stream?**
READING DATA INTO A BUFFER WITH FREAD()

- Create a buffer of size BUF_SIZE:
  - `uint8_t[BUF_SIZE] mybuf;`
- Recall fread():
  - `fread(pointer, size_of_item, num_items, fp);`
  - `fread(mybuf, BUF_SIZE, 1, fp);`
- How big to make BUF_SIZE?
  - For homework 1, can make it 8192
  - But what about your web server?
• Reading entire file into memory not efficient

• 3MB JPEG? 20MB JPEG?

• Not good to have memory usage of your server == size of files served

• Videos can be gigabytes in size

  • So what can we do?
WRITING IN FIXED-SIZED CHunks

• Writing out a big image

```c
uint8_t buffer[512];
uint64_t offset = 0;
do {
    read up to 512 bytes from file at offset into buffer
    write out buffer’s contents to the web client
    offset += how much data was read into the buffer;
} while (offset < file size);
```

• But what about reading (and parsing?)
  • Case 1: We don’t know the maximum size of a request
  • Case 2: We do know the maximum size of a request
CASE 1: DYNAMIC ARRAYS REVIEW

- Growable array of bytes
  - Underpins classes like Vector, ArrayList, etc...
- State variables:
  - uint64_t capacity;
  - uint8_t buffer[capacity]
  - uint64_t size
- Appending item ‘myItem’:
  - size = size + 1
  - buffer[size] = myItem;
- Accessing item ‘i’:
  - return buffer[i];
- But what if array is full?
  - I.e., (size == capacity)?
- Doubling algorithm:
  - allocate newbuffer[2 * capacity];
  - copy buffer into newbuffer
  - Replace pointer in the data structure with a pointer to newbuffer
  - Free the original buffer
- Amortized cost to insert?
  - O(1)
READING REQUEST OF UNKNOWN SIZE INTO A DYNAMIC BUFFER

dynamic_buffer requestBuffer;

uint8_t readBuffer[512];

while(...) {
    read up to 512 bytes from web client into readBuffer
    requestBuffer.append(readBuffer);
    Does requestBuffer contain a full request?
    If yes,(1) parse it, then (2) remove from requestBuffer
    Otherwise, keep reading from the client

HOW TO TELL IF BUFFER CONTAINS A COMPLETE REQUEST?

- This is the framing problem
- For length-based framing:
  - Keep reading until we have 12 bytes of request data
- For delimiter-based framing:
  - OK to simply scan for delimiters using e.g., a for() loop
    ```java
    for(int i = 0; i < requestBuffer.size(); i++)
    if (requestBuffer[i] == '\r') ...
    ```
CASE 2: REQUEST OF KNOWN SIZE (HW1, PROJ1, ...)

```c
uint8_t requestBuffer[8192];
uint8_t readBuffer[512];
uint64_t offset = 0;
while(...) {
    read up to 512 bytes from client into readBuffer
    copy bytes from readBuffer into requestBuffer at offset
    Does requestBuffer contain a full request?
        If yes: (1) parse it, then (2) move remainder of requestBuffer to the beginning, (3) keep reading
        If no: keep reading
}
```
HOMEWORK 1 SUMMARY

• Reading the file:
  • Buffer of type uint8_t[BUFSIZE]
  • fread() data from the stream into the buffer
  • You can assume fixed input size of 8,192 bytes

• Parsing a sequence of instructions:
  • Fine iterate through array looking for <CR> or <LF>
    • E.g., for() loop
  • But reading 1 byte at a time not efficient
    • 1 system call/context switch per byte!
Outline

1. Homework 1 strategy
2. Internetworking overview
3. IP and IP addresses
4. DNS and naming
5. DNS API
• What is an internetwork?

• An arbitrary collection of networks interconnected to provide some sort of host-host to packet delivery service
INTERNETWORKING PROTOCOL

- Each host has a “local” address on specific sub-network
  - Ethernet, WiFi, UCSD, Comcast, AT&T, Verizon Wireless, ...
- Yet each host has a single, global “IP” address
Outline

1. Homework 1 strategy
2. Internetworking overview
3. IP and IP addresses
4. DNS and naming
5. DNS API
IP SERVICE MODEL

• Packet Delivery Model
  • Connectionless model for data delivery
  • Best-effort delivery (unreliable service)
    • packets are lost
    • packets are delivered out of order
    • duplicate copies of a packet are delivered
    • packets can be delayed for a long time

• Global Addressing Scheme
  • Provides a way to identify all hosts in the network
**IP PACKET FORMAT**

- **Version (4):** currently 4
- **Hlen (4):** number of 32-bit words in header
- **TOS (8):** type of service (not widely used)
- **Length (16):** number of bytes in this datagram
- **Ident (16):** used by fragmentation
- **Flags/Offset (16):** used by fragmentation
- **TTL (8):** number of hops this datagram has traveled
- **Protocol (8):** demux key (TCP=6, UDP=17)
- **Checksum (16):** of the header only
- **DestAddr & SrcAddr (32)**
GLOBAL ADDRESSES

- **Properties**
  - globally unique
  - hierarchical: network + host
  - 4 Billion IP address, half are A type, ¼ is B type, and 1/8 is C type

- **Format**

- **Dot notation**
  - 10.3.2.4
  - 128.96.33.81
  - 192.12.69.77
CIDR: CLASSLESS INTERDOMAIN ROUTING

• Original IP address design: **limited** network sizes
  • 256, 65536, or 16777216 hosts per network
    • Not very flexible!

• CIDR enables any power-of-two network size
  • Networks with 16 hosts, or 32 hosts, etc.
  • Number of bits assigned to the host part of the address indicated with a “/”
CIDR EXAMPLES

- 192.168.1.1/16
  - First 16 bits = network, last 16 bits = \(2^{16}\) hosts

- 206.109.3.1/24
  - First 24 bits = network, last 8 = 256 hosts

- 212.110.9.1/30
  - First 30 bits = network, only 4 hosts in that network!
ASSIGNING ADDRESSES VIA DHCP

• DHCP server is responsible for providing configuration information to hosts
• There is at least one DHCP server for an administrative domain
• DHCP server maintains a pool of available addresses
DHCP IN ACTION

- Newly booted or attached host sends DHCPDISCOVER message to a special IP address (255.255.255.255)

- DHCP relay agent unicasts the message to DHCP server and waits for the response
Outline

1. Homework 1 strategy
2. Internetworking overview
3. IP and IP addresses
4. DNS and naming
5. DNS API
DNS HOSTNAME VERSUS IP ADDRESS

- **DNS host name** (e.g. www.cs.ucsd.edu)
  - **Mnemonic** name appreciated by humans
  - **Variable length**, full alphabet of characters
  - Provides **little** (if any) information about location

- **IP address** (e.g. 128.112.136.35)
  - Numerical address appreciated by **routers**
  - **Fixed length**, decimal number
  - **Hierarchical** address space, related to host location
MANY USES OF DNS

• Hostname to IP address translation
  • IP address to hostname translation (reverse lookup)
• Host name aliasing: other DNS names for a host
  • Alias host names point to canonical hostname
• Email: Lookup domain’s mail server by domain name
ORIGINAL DESIGN OF DNS

• Per-host file named /etc/hosts (1982)
  • Flat namespace: each line = IP address & DNS name
  • SRI (Menlo Park, California) kept the master copy
  • Everyone else downloads regularly

• But, a single server doesn’t scale
  • Traffic implosion (lookups and updates)
  • Single point of failure

• Need a distributed, hierarchical collection of servers
DNS: GOALS AND NON-GOALS

• A wide-area **distributed database**

• Goals:
  • **Scalability**; decentralized maintenance
  • **Robustness**
  • Global scope
    • Names mean the same thing everywhere
  • Distributed updates/queries
  • Good **performance**

• But **don’t need** strong consistency properties
DOMAIN NAME SYSTEM (DNS)

- Hierarchical name space divided into contiguous sections called **zones**
  - Zones are distributed over a collection of DNS servers
- Hierarchy of DNS servers:
  - **Root** servers (identity hardwired into other servers)
  - Top-level domain (TLD) servers
  - **Authoritative** DNS servers
- Performing the translations:
  - **Local** DNS servers located near clients
  - **Resolver** software running on clients
DNS IS HIERARCHICAL

- Hierarchy of namespace matches hierarchy of servers
- Set of nameservers answers queries for names within zone
- Nameservers store names and links to other servers in tree

• 13 root servers. *Does this scale?*
• 13 root servers. *Does this scale?*

• *Each server is really a* **cluster of servers (some geographically distributed), replicated via IP anycast**
TLD AND AUTHORITATIVE SERVERS

• **Top-level domain (TLD) servers**
  • Responsible for com, org, net, edu, etc, and all top-level country domains: uk, fr, ca, jp
  • Network Solutions maintains servers for com TLD
  • Educause non-profit for edu TLD

• **Authoritative DNS servers**
  • An organization’s DNS servers, providing authoritative information for that organization
  • May be maintained by organization itself, or ISP
LOCAL NAME SERVERS

- Do not strictly belong to hierarchy
- Each ISP (or company, or university) has one
  - Also called default or caching name server
- When host makes DNS query, query is sent to its local DNS server
  - Acts as proxy, forwards query into hierarchy
  - Does work for the client
DNS RESOURCE RECORDS

• DNS is a distributed database storing resource records
• Resource record includes: (name, type, value, time-to-live)

Type = A (address)
• name = hostname
• value is IP address

Type = NS (name server)
• name = domain (e.g. princeton.edu)
• value is hostname of authoritative name server for this domain

Type = CNAME
• name = alias for some “canonical” (real) name
• value is canonical name

Type = MX (mail exchange)
• name = domain
• value is name of mail server for that domain
Most queries and responses are UDP datagrams

Two types of queries:

**Recursive**: Nameserver responds with answer or error

Client \[\text{www.ucsd.edu?}\] Nameserver

Answer: \text{www.ucsd.edu A 132.239.180.101}

**Iterative**: Nameserver may respond with a referral

Client \[\text{www.ucsd.edu?}\] Nameserver

Referral: \text{.edu NS a.edu-servers.net.}
RECURSIVE DNS IN ACTION

client

www.cs.ucsd.edu

local DNS proxy

cs.ucsd.edu
ucsd=IPaddr

cs=IPaddr'

edu DNS server

ucsd DNS server

cs DNS server

www=IPaddr'.
**Recursive query**

- Less burden on entity initiating the query
- More burden on nameserver (has to return an answer to the query)
- Most root and TLD servers won’t answer (shed load)
  - Local name server answers recursive query

**Iterative query**

- More burden on query initiator
- Less burden on nameserver (simply refers the query to another server)
DNS CACHING

- Performing all these queries takes time
  - And all this before actual communication takes place
- Caching can greatly reduce overhead
  - The top-level servers very rarely change
    - Popular sites visited often
  - Local DNS server often has the information cached
- How DNS caching works
  - All DNS servers cache responses to queries
  - Responses include a time-to-live (TTL) field
    - Server deletes cached entry after TTL expires
Outline

1. Homework 1 strategy
2. Internetworking overview
3. IP and IP addresses
4. DNS and naming
5. DNS API
MAPPING NAMES TO ADDRESSES

GETADDRINFO(3) Linux Programmer's Manual GETADDRINFO(3)

NAME
getaddrinfo, freeaddrinfo, gai_strerror - network address and service translation

SYNOPSIS
#include <sys/types.h>
#include <sys/socket.h>
#include <netdb.h>

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

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
LINKED LIST OF ‘ADDRINFO’ STRUCTS

```c
struct addrinfo {
    int ai_flags;
    int ai_family;
    int ai_socktype;
    int ai_protocol;
    socklen_t ai_addrlen;
    struct sockaddr *ai_addr;
    char *ai_canonname;
    struct addrinfo *ai_next;
};
```

- Q: Why a linked list?
- Q: Which of the multiple results should you use?
ANNOUNCEMENTS

Homework 1 deadline pushed to Wednesday at 5pm

- Some updates on the FAQ, posted to the class website

Reading due: Donahoo and Calvert, Chapter 2

Start working on Project 1!
UC San Diego