Week 7 - Wednesday

# COMP 3400

# Last time

- What did we talk about last time?
- Started TCP programming: HTTP

# Questions?

# Project 2

# TCP Socket Programming

# Sample request

- HTTP requests and responses start with header lines
  - Each ends with CRLF (**`\r\n`**), with an extra CRLF after all headers
  - Each **`\r\n`** would simply look like a newline, but we show them below for clarity
- The most common client request is GET
- It must have a line like the following:

```
GET /path HTTP/version\r\n
```

  - **`path`** is the file being requested
  - **`version`** is the HTTP version, usually 1.0, 1.1, or 2

```
GET /index.html HTTP/1.0\r\n
Accept: text/html\r\n
Accept-Encoding: gzip, deflate, br\r\n
Accept-Language: en-US,en;q=0.5\r\n
User-Agent: Mozilla/5.0\r\n
\r\n
```

# netcat

- In order to test clients and servers, it's useful to have a program that allows a text-only TCP connection
- netcat is a common one
  - Installed as executable **nc** on the Ubuntu machines in the lab
- The first line below connects to example.com on port 80
- The next three lines are header lines typed by the user
- Lines marked in green are responses from the server

```
$ nc -v example.com 80
GET / HTTP/1.1
Host: example.com
Connection: close

HTTP/1.1 200 OK
…
```

# Doing it in code

- Once you've created a client socket and successfully connected to a server using the address information from the last class, you can send this data in code by creating a string
  - **`snprintf()`** is also a good choice for arbitrarily formatted data

```
size_t length = 500;
char buffer[length + 1];
memset (buffer, 0, sizeof (buffer));

// Copy first line and shrink remaining length
strncpy (buffer, "GET /web/index.html HTTP/1.0\r\n", length);
length = 500 - strlen (buffer);

// Concatenate each additional header line
strncat (buffer, "Accept: text/html\r\n\r\n", length);
length = 500 - strlen (buffer);
write (socketfd, buffer, strlen (buffer));  // Send over socket
```

# Sample response

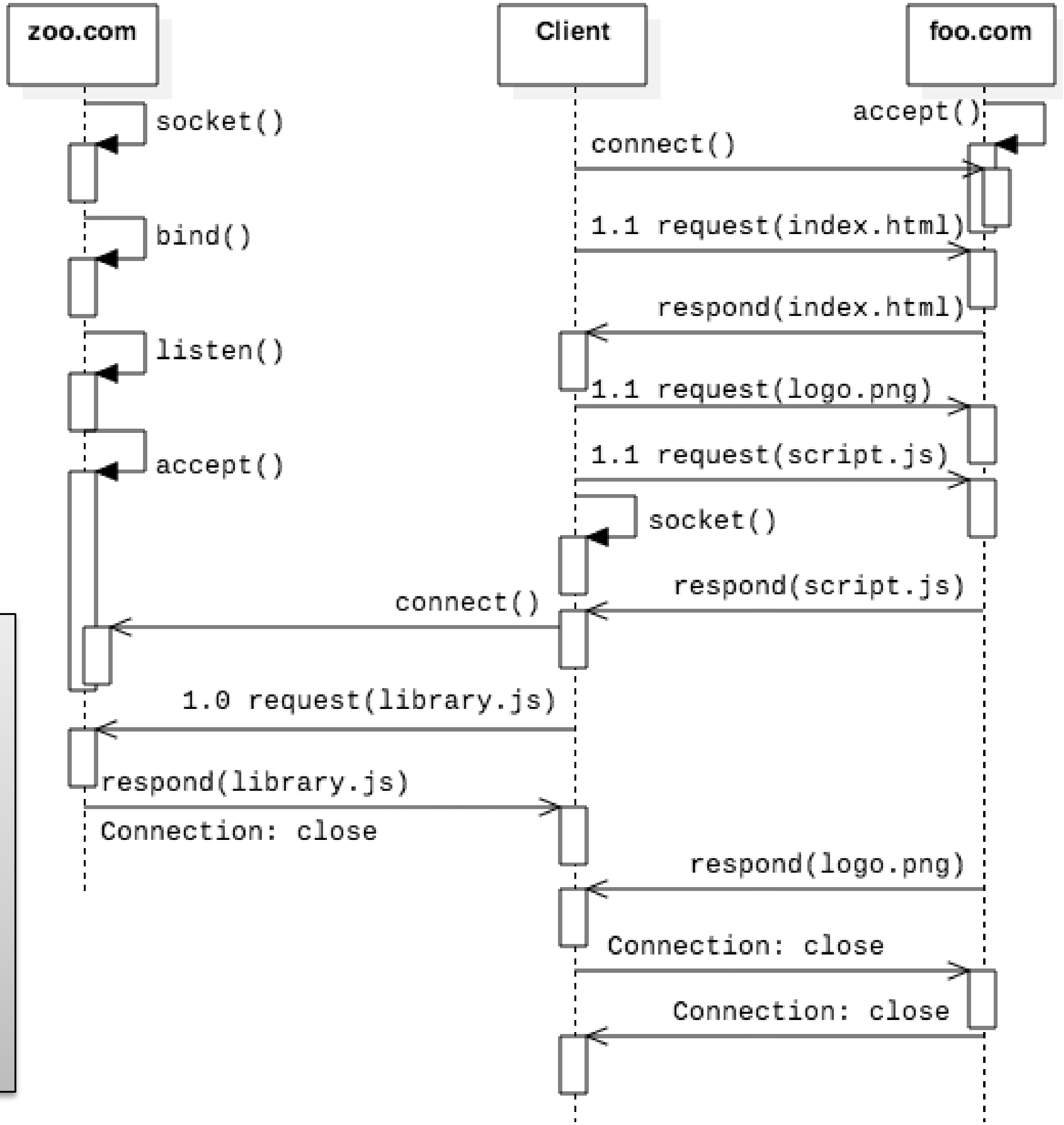
- After sending that data, the response will look something like the following:

```
HTTP/1.1 200 OK\r\n
Content-Type: text/html; charset=UTF-8\r\n
Date: Sun, 28 Feb 2021 22:20:28 GMT\r\n
Content-Length: 1256\r\n
Connection: close\r\n
\r\n
<!doctype html>\n
<html>\n
<head>\n
    <title>Example Domain</title>\n
</head>\n
<body>\n
<div>\n
    <h1>Example Domain</h1>\n
    <p>This domain is for use in illustrative examples in documents.</p>\n
</div>\n
</body>\n
</html>\n
```

# Persistent connections

- HTTP/1.0 was one and done
- HTTP/1.1 allows for persistent connections, so that multiple requests can be made over the same TCP connection
- HTML that requires multiple requests is below
- The sequence diagram showing the communication is on the right

```html
<html>
<head>
<script src="http://zoo.com/library.js" />
<script src="script.js" />
</head>
<body><img src="logo.png" /></body>
</html>
```



# HTTP headers

- A successful response to an HTTP request usually starts with:

```
HTTP/1.1 200 OK
```

- But there are many other common status codes:

| Status | Text | Explanation |
|---|---|---|
| 200 | `OK` | Request was successful |
| 301 | `Moved Permanently` | File has been moved to a new location |
| 400 | `Bad Request` | The HTTP request had incorrect syntax |
| 401 | `Unauthorized` | The request requires user authentication |
| 403 | `Forbidden` | Access to the resource is not allowed |
| 404 | `Not Found` | No file was found based on Request-URI |
| 500 | `Internal Server Error` | The server had an unexpected error or fault |
| 503 | `Service Unavailable` | The server is unavailable or not accepting new requests |

# Receiving headers

- Sending headers is easy because you know how much data you've got
- Receiving is harder, requiring a fixed length buffer with what you hope is plenty of room

```
#define HEADER_MAX 8192

// Allocate a buffer to handle initial responses up to 8 KB
char buffer[HEADER_MAX + 1];
ssize_t bytes = read (socketfd, buffer, HEADER_MAX);
assert (bytes > 0);

// If we can't find the CRLFCRLF, the header was too long
char *eoh = strnstr (buffer, "\r\n\r\n", HEADER_MAX);
if (eoh == NULL)
  {
    fprintf (stderr, "Header exceeds 8 KB maximum\n");
    close (socketfd);
    return EXIT_FAILURE;
  }
// Replace the CRLF CRLF with \0 to split the header and body
eoh[2] = '\0';
```

# Processing headers

- After reading headers on the previous slides, we can look through each one
- One critical thing is to find the length of the content, so we can allocate enough space for it

```
char *line = buffer;
char *eol = strstr (line, "\r\n");
size_t body_length = 0;
while (eol != NULL)  // While there are more CRLFs
  {
    eol[0] = '\0';    // Null-terminate each line
    printf ("HEADER LINE: %s\n", line);

    // Find content length
    if (! strncmp (line, "Content-Length: ", 16))
      {
        char *len = strchr (line, ' ') + 1;
        content_length = strtol (len, NULL, 10);
      }

    line = eol + 2; // Move to the next line
    eol = strstr (line, "\r\n");
  }
```

# Getting the content

- On the previous slide, we found the length of the content
- It's possible that the content was so small we read it into our 8 KB buffer
- Otherwise, we'll need to allocate more space

```
int length = strlen(eoh + 4);
char *content = malloc(length + 1);
strcpy(content, eoh + 4);
if (content_length > length) // if false, all data received
  {
    // Increase the content size and read additional data
    // Bytes needed is the Content-Length minus bytes already received
    content = realloc (content, content_length);
    bytes = read (socketfd, content + length, content_length - length);
  }
```

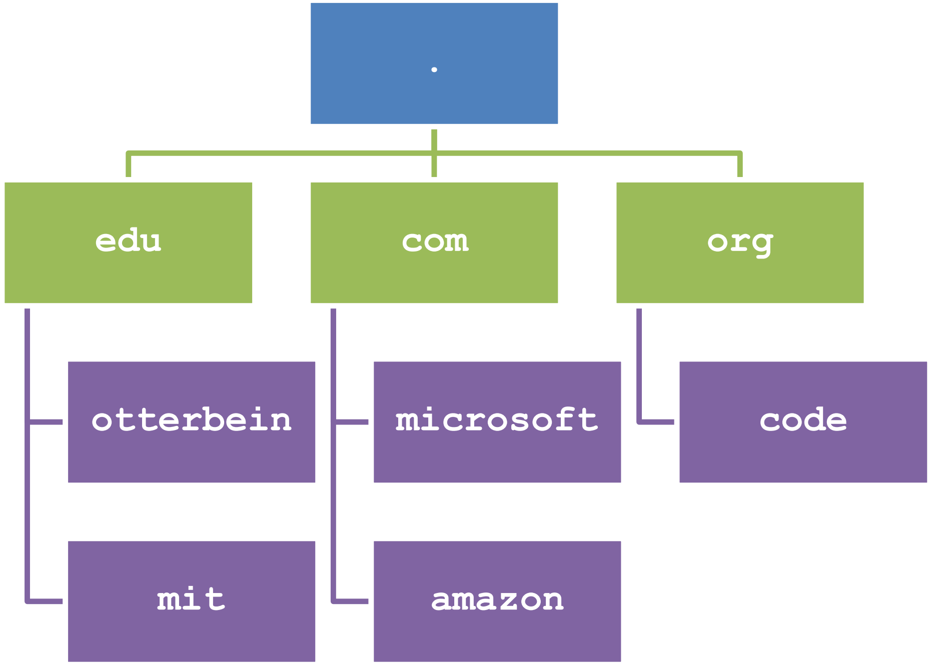
# UDP Socket Programming

# UDP socket programming

- As with TCP, it's hard to give meaningful examples of code without using some application-level protocol
- For TCP, we did HTTP
- For UDP, we'll do DNS
- **DNS**, the **Domain Name System**, is the distributed network of servers that translates domain names into IP addresses

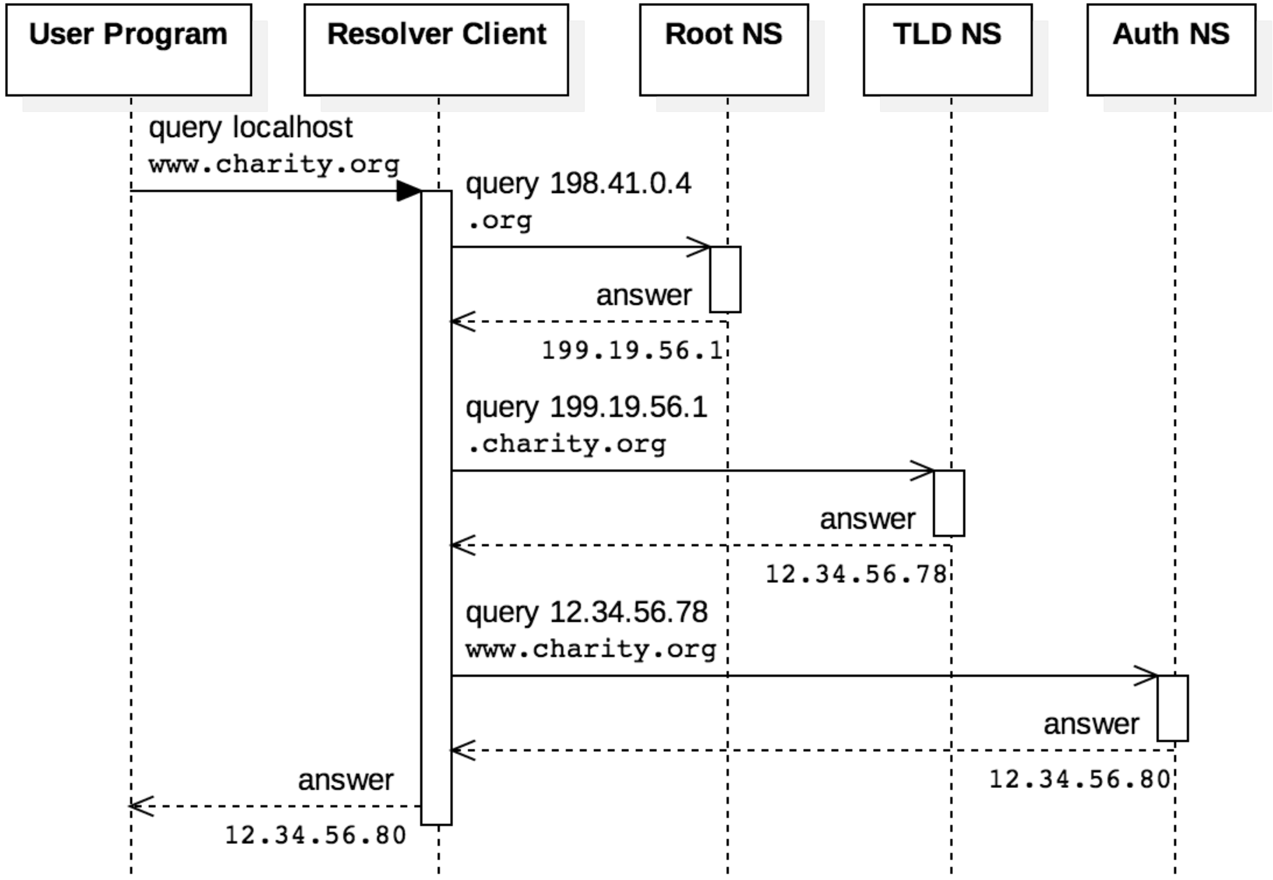
# DNS

- ICANN maintains the root structure of DNS
- Below the root are top level domains (TLDs) like **`com`**, **`edu`**, **`org`**, **`net`** and a lot of weird newish ones like **`engineering`** and **`pink`**
- Different companies manage each TLD
- Domains can be looked up from the TLD that houses it
  - **`edu`** knows where **`otterbein.edu`** can be found
  - Dots separate each entity
- It's a kind of little endian ordering where the leftmost entity is the most specific, growing more general to the right
- DNS is case insensitive



# DNS queries

- Queries can be iterative:
  - Ask the root, get a response for the TLD
  - Ask the TLD for the domain you want
  - Get a response closer to what you're looking for and repeat
  - Shown on the right
- Queries can also be recursive:
  - Ask a name server, it handles everything
- To make the system efficient, servers cache domains that have been asked for recently
- There's a time-to-live value that says how long a cached domain should be kept



# DNS resource record structure

- DNS information is sent in resource records, which have the following form:
  - NAME is the human-readable domain name
  - TYPE is gives the kind of record
    - A is an IP address
    - CNAME is a canonical name
    - NS is an authoritative name server
  - CLASS is what protocol, often IN for Internet
  - TTL is time-to-live in a cache
  - RDLENGTH is the length of the data in the record
  - RDATA is the data
- NAME and RDATA are variable length, and all other fields are 16 bits

| NAME |
|---|
| TYPE |
| CLASS |
| TTL |
| RDLENGTH |
| RDATA |

# DNS requests

- Like HTTP, DNS is a request-response protocol
- Unlike HTTP, DNS uses UDP and messages aren't as human readable
- DNS messages contain five fields: header, question, answer, authority, and additional
  - Headers start with a random ID to keep messages straight
- Example request to resolve `example.com`:

| Field | Data in Hex | Meaning |
|---|---|---|
| Header | `1234` | `XID=0x1234` |
| | `0100` | `OPCODE=SQUERY` |
| | `0001 0000 0000 0000` | 1 question field |
| Question | `0765 7861 6d70 6c65 0363 6f6d 00` | `QNAME=EXAMPLE.COM` |
| | `0001 0001` | `QCLASS=IN, QTYPE=A` |
| Answer | | |
| Authority | | |
| Additional | | |

Note:

Instead of dots, `QNAME` gives the number of characters for each name part

| Character | 7 | e | x | a | m | p | l | e | 3 | c | o | m | 0 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Hex | 07 | 65 | 78 | 61 | 6d | 70 | 6c | 65 | 03 | 63 | 6f | 6d | 00 |

# DNS responses

- Here's a reasonable response to the request from the previous slide
- Don't worry about the OPCODE, it's a set of bits laid out according to DNS rules
- QNAME uses a special code to indicate that the name is 12 bytes into this response (to avoid repetition)

| Field | Data in Hex | Meaning |
|---|---|---|
| Header | 1234 | XID=0x1234 |
| | 8180 | OPCODE=SQUERY, RESPONSE, RA |
| | 0001 0001 0000 0000 | 1 question and 1 answer |
| Question | 0765 7861 6d70 6c65 0363 6f6d 00 | QNAME=EXAMPLE.COM |
| | 0001 0001 | QCLASS=IN, QTYPE=A |
| Answer | c00c | QNAME=EXAMPLE.COM [compressed] |
| | 0001 | QTYPE=A |
| | 0001 | QCLASS=IN |
| | 0000 e949 | TTL = 0xe949 = 59721 |
| | 04 | RDLENGTH = 4 |
| | 0x5db8d822 [93.184.216.34] | RDATA |
| Authority | | |
| Additional | | |

# Ticket Out the Door

# Upcoming

# Next time…

- Finish UDP socket programming
- Broadcasting

# Reminders

- Keep working on Project 2
- Read sections 4.6 and 4.7