Week 7 - Friday

COMP 3400

Last time

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

Questions?

Project 2

Reminders about `sizeof`

`sizeof`

- In C, the **`sizeof`** operator was designed to get the size of types and variables in **bytes**
- It should be used to get information known at **compile time**
- It can never know the length of:
  - Files
  - Strings
  - Dynamically allocated memory
- Yes, it's called **`sizeof`**, but a lot of things have non-intuitive names in CS

Testing your `sizeof` knowledge

```
int array[100];
char word1[] = "goats";
char word2[50] = "goats";
char* word3 = "goats";
int x = 500;
char* data = malloc(100);
int fd = open("file.txt", O_RDONLY);
```

- Given the above code, what is the value of each?
  - `sizeof(array)`
  - `sizeof(array) - 1`
  - `sizeof(array - 1)`
  - `sizeof(word1)`
  - `sizeof(word2)`
  - `sizeof(word3)`
  - `sizeof("goats")`
  - `sizeof(x)`
  - `sizeof(data)`
  - `sizeof(fd)`
- Answers given on next slide

Answers

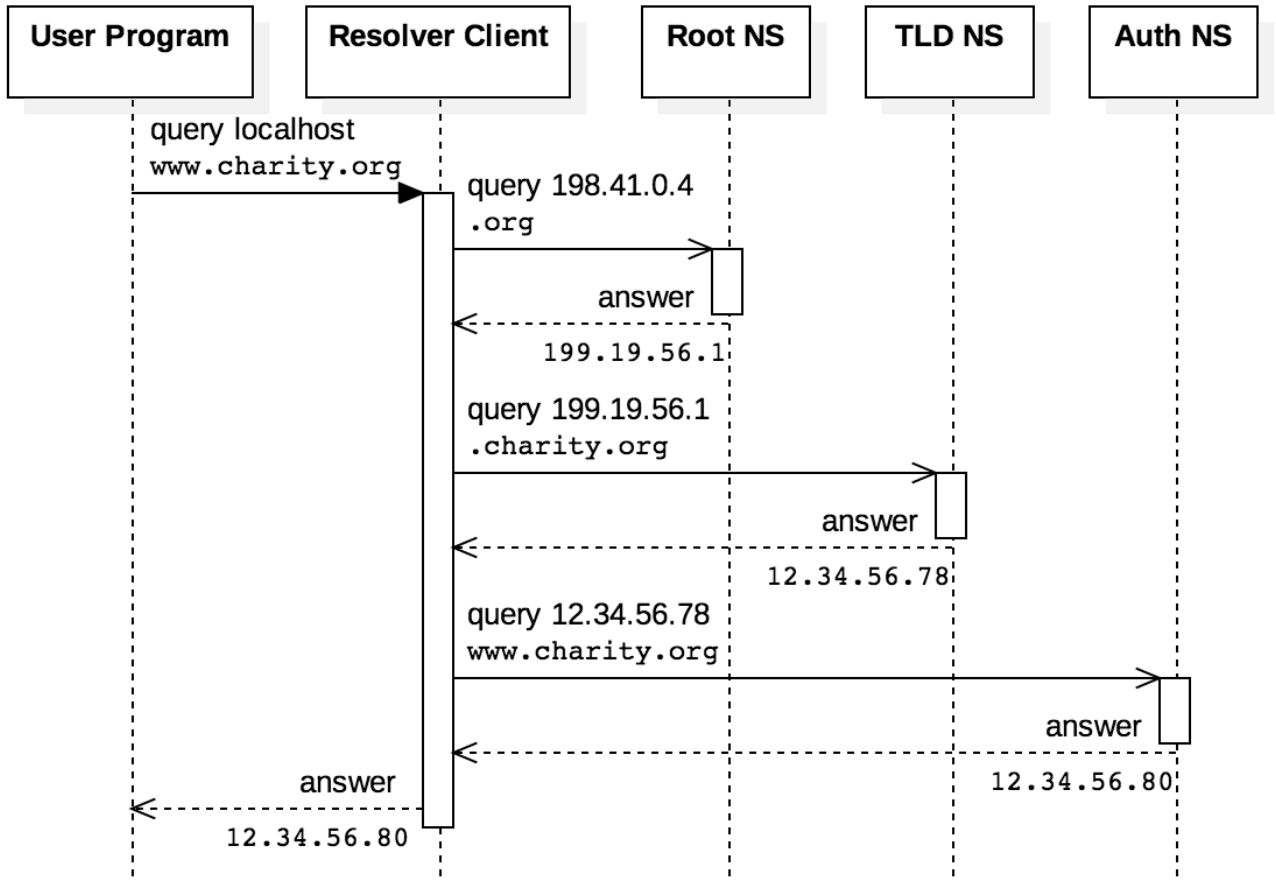
```
int array[100];
char word1[] = "goats";
char word2[50] = "goats";
char* word3 = "goats";
int x = 500;
char* data = malloc(100);
int fd = open("file.txt", O_RDONLY);
```

- Note that these answers are based on the Ubuntu in the lab, which uses 64-bit addresses
  - `sizeof(array)` 400
  - `sizeof(array) - 1` 399
  - `sizeof(array - 1)` 8
  - `sizeof(word1)` 6
  - `sizeof(word2)` 50
  - `sizeof(word3)` 8
  - `sizeof("goats")` 6
  - `sizeof(x)` 4
  - `sizeof(data)` 8
  - `sizeof(fd)` 4

Back to DNS

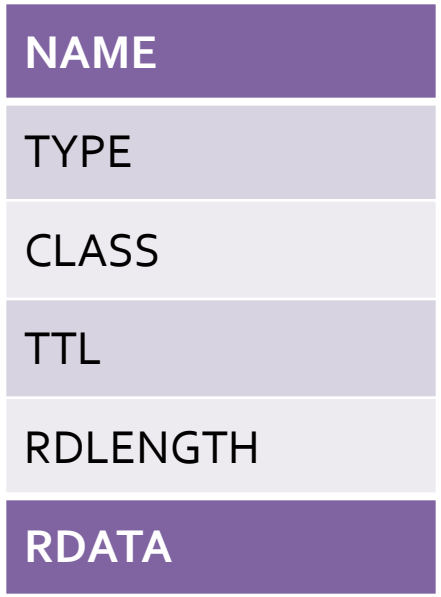
DNS queries

- 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



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		

Brief interlude

- Did you ever wonder how long a domain name can be?
- Each part of the name has a maximum of 63 characters
- The whole thing can't be more than 253 characters
- Examples:
  - The Welsh village Llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogoch registered **llanfairpwllgwyngyllgogerychwyrndrobwllllantysiliogogogochuchaf.org.uk** in honor of the *uchaf* or upper part of their village
  - German mathematician Gerard Steffens registered **3.14159265358979323846264338327950288419716939937510582097494459 2.eu** in honor of pi
- In 2000 (when both the web and Verizon were fresh and new), Verizon registered **verizonsucks.com** to keep anyone else from using it
  - The hacker magazine 2600 registered **verizonreallysucks.com**
  - Verizon sued the magazine's publisher
  - In retaliation, the magazine registered the domain **VerizonShouldSpendMoreTimeFixingItsNetworkAndLessMoneyOnLawyers.com**

Putting it into code

- DNS isn't part of the POSIX standard, so we need our own structs to hold the data

```
typedef struct {
  uint16_t xid;      // Randomly chosen identifier
  uint16_t flags;    // Bit-mask to indicate request/response
  uint16_t qdcount;  // Number of questions
  uint16_t ancount;  // Number of answers
  uint16_t nscount;  // Number of authority records
  uint16_t arcount;  // Number of additional records
} dns_header_t;

typedef struct {
  char *name;        // Pointer to the domain name in memory
  uint16_t dnstype;  // The QTYPE (1 = A)
  uint16_t dnsclass; // The QCLASS (1 = IN)
} dns_question_t;
```

Preparing to send

- The following code:
  - Creates a UDP socket
  - Makes an IPv4 address with the OpenDNS server 208.67.222.222, which is **0xd043dede** in hex on the DNS port of 53
  - Initializes a **dns_header_t** with appropriate values

```
int socketfd = socket (AF_INET, SOCK_DGRAM, 0);
struct sockaddr_in address;
address.sin_family = AF_INET;                    // IPv4
address.sin_addr.s_addr = htonl (0xd043dede); // 208.67.222.222 (0xd043dede)
address.sin_port = htons (53);                   // Port 53

// DNS header
dns_header_t header;
memset (&header, 0, sizeof (dns_header_t));
header.xid= htons (0x1234);     // Randomly chosen ID
header.flags = htons (0x0100); // Q=0, RD=1
header.qdcount = htons (1);     // Sending 1 question
```

Horrible code to fill in the name

- The following code (pretty slickly) fills in the weird naming scheme that requires a count for the length of each name part before it

```
dns_question_t question;
question.dnstype = htons (1);  // QTYPE 1=A
question.dnsclass = htons (1); // QCLASS 1=IN
question.name = calloc (strlen (hostname) + 2, sizeof (char)); // 2 more: \0 and first count
memcpy (question.name + 1, hostname, strlen (hostname));
uint8_t *prev = (uint8_t *) question.name;
uint8_t count = 0; // Count the bytes in a field

for (size_t i = 0; i < strlen (hostname); ++i) // Look for . locations
  {
      if (hostname[i] == '.') // End of a name part
      {
        *prev = count;         // Store the count into the location before the part
        prev = question.name + i + 1; // Update the prev pointer to the new location
        count = 0;
      }
      else
        ++count;
  }
*prev = count; // Store count for last part
```

Finally sending

- Before sending, everything must be packaged into one chunk of memory

```
// Final packet size
size_t packetlen = sizeof (header) + strlen (hostname) + 2 +
  sizeof (question.dnstype) + sizeof (question.dnsclass);
uint8_t *packet = calloc (packetlen, sizeof (uint8_t));
uint8_t *p = (uint8_t *)packet;

memcpy (p, &header, sizeof (header)); // Copy in the header
p += sizeof (header);

// Copy the question name, QTYPE, and QCLASS fields
memcpy (p, question.name, strlen (hostname) + 2);
p += strlen (hostname) + 2;
memcpy (p, &question.dnstype, sizeof (question.dnstype));
p += sizeof (question.dnstype);
memcpy (p, &question.dnsclass, sizeof (question.dnsclass));

// Finally, send the packet over UDP
sendto (socketfd, packet, packetlen, 0, (struct sockaddr *) &addr,
        (socklen_t) sizeof (addr));
```

Getting an answer back

- The DNS standard says that a message will never be more than 512 bytes
- Thus, we can just read into a fixed-size buffer

```
socklen_t length = 0;
uint8_t response[512];
memset (&response, 0, 512);

// Receive the response
ssize_t bytes = recvfrom (socketfd, response, 512, 0, (struct sockaddr *)
&addr, &length);
```

Interpreting that answer

- The following struct gives us a way to interpret the elements of the answer
- Note the **`__attribute__((packed))`** at the bottom
  - This compiler flag keeps the compiler from reorganizing the fields
  - It's necessary so that everything matches the output we expect from the DNS server
  - Compilers will often change struct fields around for greater efficiency

```
typedef struct {
  uint16_t compression;
  uint16_t type;
  uint16_t class;
  uint32_t ttl;
  uint16_t length;
  struct in_addr addr;
} __attribute__((packed)) dns_record_a_t;
```

Reconstructing the name

- The following code reconstructs the name, putting dots back in it, and lets us see where the data after it is

```
dns_header_t *response_header = (dns_header_t *)response;
assert ((ntohs (response_header->flags) & 0xf) == 0); // Check for error

// Get a pointer to the start of the question name
uint8_t *start_of_name = (uint8_t *) (response + sizeof (dns_header_t));
uint8_t total = 0;
uint8_t *field_length = start_of_name;
while (*field_length != 0)
  {
    // Put a dot back in the name and advance to next length
    total += *field_length + 1;
    *field_length = '.';
    field_length = start_of_name + total;
  }
```

Actual DNS information

- Finally, after the name, we can skip a null byte, qtype, qclass to get to the answers
- Note that we have to be careful to change the data from network to host endianness

```
dns_record_a_t *records = (dns_record_a_t *) (field_length + 5);
for (int i = 0; i < ntohs (response_header->ancount); ++i)
  {
    printf ("TYPE: %" PRId16 "\n", ntohs (records[i].type));
    printf ("CLASS: %" PRId16 "\n", ntohs (records[i].class));
    printf ("TTL: %" PRIx32 "\n", ntohl (records[i].ttl));
    printf ("IPv4: %08" PRIx32 "\n", ntohl (records[i].addr));
    printf ("IPv4: %s\n", inet_ntoa (records[i].addr));
  }
```

DNS madness

- It's hard to follow all the code that we're going through in class
- Try to comb through it on your own
  - Note that there are a few mistakes in the book
- Reading and understanding code is one of the most valuable skills you can develop
- The good news: A full DNS client program is given in section 5.8 of the book if you want to see all the code uninterrupted

Upcoming

Next time…

- Broadcasting
- Deeper into the Internet

Reminders

- Keep working on Project 2!
- Read sections 5.1, 5.2, and 5.3