1. Purpose
Understand how to design non-ASCII application-layer protocols and provide primitive reliable transport over UDP.

2. Description
Your goal is to implement a program that issues recursive queries to a user-provided DNS server and parses its responses. For testing while on campus (or VPN), you can use an IRL server 128.194.135.79. You can also run your own DNS server on localhost, but keep in mind that Wireshark may not be able to intercept these packets. Public servers include your ISP’s DNS server and Google’s 8.8.8.8, but there is a possibility they may temporarily block you after seeing too many invalid queries.

2.1. Code (75 pts)
Your program must accept two command-line arguments (if they are not present, report usage and quit). The first argument is the lookup string, which may be a hostname or IP, and the second is the DNS server IP to which the query is going. Examples:

```bash
hw2.exe www.xyz.com 128.194.135.79
hw2.exe 128.194.135.66 8.8.8.8
```

Your code must directly use UDP and parse DNS responses without any shortcuts (e.g., Platform SDK, boost, or other libraries). It is acceptable to use STL strings to assemble responses scattered across the packet, although a sequence of memcpy operations into a separate buffer is the preferred technique if you are comfortable with pointers.

For successful execution, the output format is given below using examples. Additional discussion and clarification are provided afterwards.

```
Lookup : yahoo.com
Query : yahoo.com, type 1, TXID 0x0001
Server : 128.194.135.84
******************************
Attempt 0 with 27 bytes... response in 448 ms with 273 bytes
TXID 0x0001, flags 0x8180, questions 1, answers 3, authority 5, additional 6
succeeded with Rcode = 0
----------- [questions] -----------
yahoo.com type 1 class 1
----------- [answers] -----------
yahoo.com A 98.139.183.24 TTL = 1800
yahoo.com A 206.190.36.45 TTL = 1800
yahoo.com A 98.138.253.109 TTL = 1800
----------- [authority] -----------
yahoo.com NS ns4.yahoo.com TTL = 172800
yahoo.com NS ns3.yahoo.com TTL = 172800
yahoo.com NS ns5.yahoo.com TTL = 172800
yahoo.com NS ns2.yahoo.com TTL = 172800
```
<table>
<thead>
<tr>
<th>Lookup</th>
<th>Query</th>
<th>Server</th>
<th>Attempt</th>
<th>Response Time</th>
<th>Bytes</th>
<th>TXID</th>
<th>Flags</th>
<th>Questions</th>
<th>Answers</th>
<th>Authority</th>
<th>Additional</th>
</tr>
</thead>
<tbody>
<tr>
<td>128.194.138.19</td>
<td>19.138.194.128.in-addr.arpa, type 12, TXID 0xAA03</td>
<td>128.194.135.84</td>
<td>0</td>
<td>1099 ms</td>
<td>199</td>
<td>0xAA03</td>
<td>0x8180</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>8.8.8.8</td>
<td><a href="http://www.google.com">www.google.com</a>, type 1, TXID 0x34C9</td>
<td>128.194.135.85</td>
<td>0</td>
<td>25 ms</td>
<td>112</td>
<td>0x34C9</td>
<td>0x8180</td>
<td>1</td>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>128.194.135.84</td>
<td><a href="http://www.dhs.gov">www.dhs.gov</a>, type 1, TXID 0x993A</td>
<td>128.194.135.85</td>
<td>0</td>
<td>177 ms</td>
<td>118</td>
<td>0x993A</td>
<td>0x8180</td>
<td>1</td>
<td>3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>128.194.135.84</td>
<td><a href="http://www.dhs.gov">www.dhs.gov</a>, type 1, TXID 0x0300</td>
<td>128.194.135.84</td>
<td>0</td>
<td>6939 ms</td>
<td>414</td>
<td>0x0300</td>
<td>0x8180</td>
<td>1</td>
<td>3</td>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>
To begin with, you report the original string provided by the user, the query that goes to DNS, and the server’s IP. For reverse DNS lookups, the query should have the special form we discussed in class. Following the query, you also need to specify its type – either DNS_A (1) or DNS_PTR (12) – and the TXID printed in hex and padded to four digits (use \%.4X in printf).

Each attempt should be labeled with the number of bytes you send into the socket. If the network produces any errors, report this together with a WSAGetLastError() code and quit. Otherwise, print the number of milliseconds spent in sendto/recvfrom and the number of bytes in the response.

Next, show the values of the fixed header, where the TXID and flags are output in hex and the other four fields in decimal. Parse the rest of the packet in each of the sections, grabbing information from CNAME, A, NS, and PTR responses and skipping over all other record types. Note that indentation given in the examples is required.

You must differentiate between successful lookups and failures, as well as detect network errors, report them to the user, and gracefully terminate the program. If the server does not respond within 10 seconds, perform a retransmission, up to a maximum of three attempts. Examples:
Note that lines beginning with ++ are indicative of malicious and/or corrupted responses, with one example random9.irl shown above. More such cases are discussed below.

2.2. Report (25 pts)

For the report, you should perform forensic investigation of our custom server 128.194.135.82 and determine what type of tweaking it applies to outgoing packets. Note that this server sends malformed DNS responses, which are blocked by the TAMU VPN. To get around this problem, you can test your code from the class server using Remote Desktop or download the custom DNS server from the course website and run it locally on your machine/Azure/ts/ts2.

The custom server accepts queries for strings in the form of randomX.irl, where $X \in \{0, 1, \ldots, 9, A, B\}$. For example, random9.irl increments your TXID by one, random2.irl generates a packet filled with 0xEF, randomA.irl sends multiple questions, and randomB.irl produces a response larger than the maximum allowed by DNS. The traces for these four cases are already...
shown above. You should be able to handle them as part of normal operation to get the full 75 points.

For the report and its 25 points, you need to demonstrate that your program can identify nine additional ++ errors:

++ invalid reply: packet smaller than fixed DNS header
++ invalid section: not enough records (e.g., declared 5 answers but only 3 found)
++ invalid record: jump beyond packet boundary
++ invalid record: truncated name (e.g., "6 goog" and the packet ends)
++ invalid record: truncated RR answer header (i.e., don’t have the full 10 bytes)
++ invalid record: truncated jump offset (e.g., 0xCO and the packet ends)
++ invalid record: jump into fixed DNS header
++ invalid record: jump loop
++ invalid record: RR value length stretches the answer beyond packet

Using experimentation and analysis, determine what types of corruption is performed in each of the cases below and show the corresponding traces from your program with the ++ error it detected.

1. (8 pts) Case 1: random0.irl, random3.irl, random5.irl, and random6.irl.
2. (2 pts) Case 2: random1.irl.
3. (3 pts) Case 3: random7.irl.
4. (12 pts) Case 4: random4.irl. Show three types of ++ errors produced by this query that are not present in any of the cases above and document your handling of each. Since these responses are randomized, you will need to run your program multiple times.

The cases above should cover all nine ++ errors stated earlier.

5. (extra credit, 10 pts): Figure out the algorithm behind random8.irl’s response. This query generates randomized replies, so you will need to run it several times to see what happens. It is not enough (or even necessary) to report the errors your code detects; instead, you should explain the essence of what the server is doing to the packet so that someone else can write code to implement something similar.

2.3. Overview
Organization of your program may look similar to the one in Figure 1.
Figure 1. Flow-chart of the program.

To decide whether the query is an IP or hostname, pass it through `inet_addr()`. If this function succeeds, proceed with a type-PTR query. Otherwise, use type-A.

After a UDP socket is opened, you must call `bind()` with port 0 to let the OS select the next available port for you:

```c
SOCKET sock = socket(AF_INET, SOCK_DGRAM, 0);  // handle errors
struct sockaddr_in local;
memset(&local, 0, sizeof(local));
local.sin_family = AF_INET;
local.sin_addr.s_addr = INADDR_ANY;
local.sin_port = htons(0);
if (bind(sock, (struct sockaddr*)&local, sizeof(local)) == SOCKET_ERROR)  // handle errors
    // Note that local.sin_addr specifies which local IP address you are binding the socket to, which may be important if you have multiple network cards in the computer. Since you do not have a preference in this homework, INADDR_ANY allows you to receive packets on all physical interfaces of the system.

There is no connect phase and sockets can be used immediately after binding:

```c
struct sockaddr_in remote;
memset(&remote, 0, sizeof(remote));
remote.sin_family = AF_INET;
remote.sin_addr = inet_addr(...);  // server’s IP
remote.sin_port = htons(53);  // DNS port on server
if (sendto(sock, buf, size, 0, (struct sockaddr*)&remote, sizeof(remote)) == SOCKET_ERROR)  // handle errors
    // The fixed DNS header is provided to you in the book and class slides. It is 12 bytes long and consists of six fields. Fill in the ID field, flags, and number of questions. Set the other three fields to zero. Following these 12 bytes is the question record described next.

Each query includes a variable-size question and a trailing fixed-size header shown in Figure 2. The question string is separated into *labels* based on the locations of the dot. For example,
“www.google.com” becomes str1 = “www”, str2 = “google”, str3 = “com”. The lengths of the corresponding strings are 3, 6, and 3 bytes. The last label has size 0 as shown in the figure.

<table>
<thead>
<tr>
<th>str1 size</th>
<th>str1</th>
<th>...</th>
<th>strn size</th>
<th>strn</th>
<th>0</th>
<th>Query type</th>
<th>Query class</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 byte</td>
<td></td>
<td></td>
<td>1 byte</td>
<td></td>
<td></td>
<td>2 bytes</td>
<td>2 bytes</td>
</tr>
</tbody>
</table>

question

Figure 2. Question format.

Query types are integer numbers specified in RFC 1035. Several useful constants:

/* DNS query types */
#define DNS_A  1  /* name -> IP */
#define DNS_NS 2  /* name server */
#define DNS_CNAME 5  /* canonical name */
#define DNS_PTR 12  /* IP -> name */
#define DNS_HINFO 13  /* host info/SOA */
#define DNS_MX  15  /* mail exchange */
#define DNS_AXFR 252  /* request for zone transfer */
#define DNS_ANY 255  /* all records */

There is only one useful query class:

/* query classes */
#define DNS_INET 1

To receive UDP responses from the server, use function recvfrom(). Each call to recvfrom() results in retrieval of one UDP packet that corresponds to the answer. It is therefore not necessary to form a loop around recvfrom() as in homework #1. Also note that the returned data is binary and cannot be directly uploaded into STL strings. To see socket error 10040 on randomB.irl, make sure to allocate a static buffer of MAX_DNS_SIZE = 512 bytes and attempt receipt into it.

Using a combination of experiments with Wireshark and RFCs 1034, 1035, your responsibility is to understand how the response is structured and write a parser for it. You may also find the following site useful: http://www.networksorcery.com/enp/protocol/dns.htm. It is recommended that you use Wireshark filters (a box near the top of the screen) to only display information related to DNS to avoid clutter on the screen (e.g., by typing “dns && ip.addr == 128.194.135.79” into the filter). Also note that Wireshark may have difficulty reading encrypted packets over VPN.

You should support both compressed and uncompressed answers. To recognize compression, check the string-size byte for being larger than 0xC0 (i.e., the two most-significant bits are 11). For this to work correctly, the byte needs to be converted to an unsigned char. If there is compression, the 14 bits following the binary 11 indicate the jump offset from the beginning of the packet. See the slides for more discussion.
2.4. Packet Loss

Since not all UDP packets are reliably delivered to your local DNS server, implement a simple retransmission scheme based on a timer. After each request is sent, enter into a wait state until you either receive a response from your local DNS server or the timer expires (use 10-second timeouts):

```c
#define MAX_ATTEMPTS 3
while (count++ < MAX_ATTEMPTS)
{
    // send request to the server...
    // get ready to receive
    fd_set fd;
    FD_ZERO (&fd);  // clear the set
    FD_SET (dns_sock, &fd); // add your socket to the set
    int available = select (0, &fd, NULL, NULL, &tp);
    if (available > 0)
    {
        recvfrom (...);
        // parse the response
        // break from the loop
    }
    // error checking here
}
```

2.5. Header Caveats

All 2-byte header fields are coded in network byte order and must be converted to/from your local host notation. The process of assembling flags involves ORing them, where each individual bit-flag is given by:

```c
/* flags */
#define DNS_QUERY (0 << 15)  /* 0 = query; 1 = response */
#define DNS_RESPONSE (1 << 15)
#define DNS_STDQUERY (0 << 11)  /* opcode - 4 bits */
#define DNS_AA  (1 << 10)  /* authoritative answer */
#define DNS_TC  (1 << 9)  /* truncated */
#define DNS_RD  (1 << 8)  /* recursion desired */
#define DNS_RA  (1 << 7)  /* recursion available */
```

For example, to set flags in outgoing packets, use

```c
htons(DNS_QUERY | DNS_RD | DNS_STDQUERY)
```

While two of these flags are zero and can be omitted, it is common practice to specify them anyway. This increases transparency of what you are doing.

Avoid manipulating individual bytes and instead use classes to write into binary arrays:

```c
#pragma pack(push,1)   // sets struct padding/alignment to 1 byte
class QueryHeader {
    USHORT qType;
    USHORT qClass;
};
```
class FixedDNSHeader {
    USHORT ID;
    USHORT flags;
    USHORT questions;
    USHORT answers;
    ...
};
#pragma pack(pop)  // restores old packing

char host[] = "www.google.com";
int pkt_size = strlen(host) + 2 + sizeof(FixedDNSHeader) + sizeof(QueryHeader);
char *buf = new char[pkt_size];

FixedDNSHeader *fdh = (FixedDNSHeader*) buf;
QueryHeader *qh = (QueryHeader*) (buf + pkt_size - sizeof(QueryHeader));

// fixed field initialization
fdh->ID = ...
fdh->flags = ...
...
qh->qType = ...
qh->qClass = ...
makeDNSquestion(fdh + 1, host);
sendto (sock, buf, ...);
delete buf;

A few common result codes are the following:

#define DNS_OK     0 /* success */
#define DNS_FORMAT 1 /* format error (unable to interpret) */
#define DNS_SERVERFAIL 2 /* can’t find authority nameserver */
#define DNS_ERROR 3 /* no DNS entry */
#define DNS_NOTIMPL 4 /* not implemented */
#define DNS_REFUSED 5 /* server refused the query */

2.6. Reading Raw Buffers

The proper way of working with fixed-size headers is to directly cast pointers into receive buffers instead of parsing results byte-by-byte. For example:

#define MAX_DNS_SIZE 512 // largest valid UDP packet
#pragma pack(push,1)  // sets struct padding/alignment to 1 byte
class FixedRR {
    u_short qType;
    u_short qClass;
    int TTL;
    ...
};
#pragma pack(pop)  // restores old packing

char buf[MAX_DNS_SIZE];
struct sockaddr_in response;
if (recvfrom (sock, buf, MAX_DNS_SIZE, 0, (struct sockaddr*) &response, ...) == ...)
    // error processing
    // check if this packet came from the server to which we sent the query earlier
    if (response.sin_addr != remote.sin_addr || response.sin_port != remote.sin_port)
        // bogus reply, complain
    FixedDNSHeader *fdh = (FixedDNSHeader*) buf;
    // read fdh->ID and other fields
    ...
    // parse questions and arrive to the answer section
// suppose off is the current position in the packet
FixedRR *frr = (FixedRR*)(buf + off);
// read frr->len and other fields
CSCE 463/612 Homework 2

Name: ______________________________

<table>
<thead>
<tr>
<th>Function</th>
<th>Points</th>
<th>Breakdown</th>
<th>Item</th>
<th>Deduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Printouts</td>
<td>58</td>
<td></td>
<td>No usage if incorrect arguments</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorrect summary (lookup/query/server)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorrect attempt info (sent/received bytes, delay)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorrect fixed header fields or their format</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Does not report successful Rcode</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorrect questions (name/type/class)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorrect answers (name/type/value/TTL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorrect authority (name/type/value/TTL)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Incorrect additional (name/type/value/TTL)</td>
<td></td>
</tr>
<tr>
<td>Operation</td>
<td>17</td>
<td></td>
<td>Does not reject error Rcodes (e.g., random2.irl)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Improper or absent retransmission</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Does not reject bogus TXID (e.g., random9.irl)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Fails to parse questions in randomA.irl</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Does not report socket errors (e.g., randomB.irl)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Crashes on certain responses</td>
<td></td>
</tr>
<tr>
<td>Report</td>
<td>25</td>
<td></td>
<td>Random.irl: 0, 3, 5, 6 (explain four ++ errors)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Random.irl: 1 (explain one ++ error)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Random.irl: 7 (explain one ++ error)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Random.irl: 4 (explain three ++ errors)</td>
<td></td>
</tr>
</tbody>
</table>

Additional deductions are possible for memory leaks and poor code structure.

Total points: ________________