CSCE 463/612: Networks and Distributed Processing

Homework 2 (100 pts)

Due date: 3/4/25

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 IRL servers 128.194.135.79 (IIS) and 128.194.135.85 (BIND). The latter returns the more-verbose answers shown below. You can also run your own DNS server on localhost, keeping in mind that Wireshark needs to be running on the loopback interface. Public servers include your ISP's DNS server, as well as Google's 8.8.8.8 and 8.8.4.4.

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:

```
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 provided below using a few examples from prior years. The Internet is constantly changing, so your results may be different, in which case nslookup should be used to verify correctness. It is also advisable to find a variety of additional test cases and perform a more exhaustive evaluation than given here.

```
Lookup  : yahoo.com
Query   : yahoo.com, type 1, TXID 0x0001
Server  : 128.194.135.85
********************************
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
        yahoo.com NS ns1.yahoo.com TTL = 172800
  ------------ [additional] ---------
        ns1.yahoo.com A 68.180.131.16 TTL = 172800
        ns2.yahoo.com A 68.142.255.16 TTL = 172800
        ns3.yahoo.com A 203.84.221.53 TTL = 172800
        ns4.yahoo.com A 98.138.11.157 TTL = 172800
        ns5.yahoo.com A 119.160.247.124 TTL = 172800
```

```
Lookup  : 128.194.138.19
Query   : 19.138.194.128.in-addr.arpa, type 12, TXID 0xAA03
Server  : 128.194.135.85
********************************
Attempt 0 with 45 bytes... response in 1099 ms with 199 bytes
  TXID 0xAA03 flags 0x8180 questions 1 answers 2 authority 3 additional 3
  succeeded with Rcode = 0
  ------------ [questions] ----------
        19.138.194.128.in-addr.arpa type 12 class 1
  ------------ [answers] ------------
        19.138.194.128.in-addr.arpa PTR mailbox.cs.tamu.edu TTL = 3600
        19.138.194.128.in-addr.arpa PTR imap.cs.tamu.edu TTL = 3600
  ------------ [authority] ----------
        194.128.in-addr.arpa NS ns3.tamu.edu TTL = 86399
        194.128.in-addr.arpa NS ns1.tamu.edu TTL = 86399
        194.128.in-addr.arpa NS ns2.tamu.edu TTL = 86399
  ------------ [additional] ---------
        ns1.tamu.edu A 128.194.254.4 TTL = 28800
        ns2.tamu.edu A 128.194.254.5 TTL = 172800
        ns3.tamu.edu A 192.195.87.5 TTL = 172800
```

```
Lookup  : www.google.com
Query   : www.google.com, type 1, TXID 0x34C9
Server  : 8.8.8.8
********************************
Attempt 0 with 32 bytes... response in 25 ms with 112 bytes
  TXID 0x34C9 flags 0x8180 questions 1 answers 5 authority 0 additional 0
  succeeded with Rcode = 0
  ------------ [questions] ----------
        www.google.com type 1 class 1
  ------------ [answers] ------------
        www.google.com A 74.125.227.244 TTL = 299
        www.google.com A 74.125.227.243 TTL = 299
        www.google.com A 74.125.227.240 TTL = 299
        www.google.com A 74.125.227.241 TTL = 299
        www.google.com A 74.125.227.242 TTL = 299
```

```
Lookup  : www.dhs.gov
Query   : www.dhs.gov, type 1, TXID 0x993A
Server  : 128.194.135.79
********************************
Attempt 0 with 29 bytes... response in 177 ms with 118 bytes
  TXID 0x993A flags 0x8180 questions 1 answers 3 authority 0 additional 0
  succeeded with Rcode = 0
  ------------ [questions] ----------
        www.dhs.gov type 1 class 1
  ------------ [answers] ------------
        www.dhs.gov CNAME www.dhs.gov.edgekey.net TTL = 3600
        www.dhs.gov.edgekey.net CNAME e6485.dscb.akamaiedge.net TTL = 300
        e6485.dscb.akamaiedge.net A 23.200.36.56 TTL = 20
```

```
Lookup  : www.dhs.gov
Query   : www.dhs.gov, type 1, TXID 0x0300
Server  : 128.194.135.85
********************************
Attempt 0 with 29 bytes... response in 6939 ms with 414 bytes
  TXID 0x0300 flags 0x8180 questions 1 answers 3 authority 8 additional 8
  succeeded with Rcode = 0
```

```
  ------------ [questions] ----------
        www.dhs.gov type 1 class 1
  ------------ [answers] ------------
        www.dhs.gov CNAME www.dhs.gov.edgekey.net TTL = 3600
        www.dhs.gov.edgekey.net CNAME e6485.dscb.akamaiedge.net TTL = 300
        e6485.dscb.akamaiedge.net A 23.200.36.56 TTL = 20
  ------------ [authority] ----------
        dscb.akamaiedge.net NS n4dscb.akamaiedge.net TTL = 4000
        dscb.akamaiedge.net NS n3dscb.akamaiedge.net TTL = 4000
        dscb.akamaiedge.net NS n6dscb.akamaiedge.net TTL = 4000
        dscb.akamaiedge.net NS n0dscb.akamaiedge.net TTL = 4000
        dscb.akamaiedge.net NS n7dscb.akamaiedge.net TTL = 4000
        dscb.akamaiedge.net NS n1dscb.akamaiedge.net TTL = 4000
        dscb.akamaiedge.net NS n5dscb.akamaiedge.net TTL = 4000
        dscb.akamaiedge.net NS n2dscb.akamaiedge.net TTL = 4000
  ------------ [additional] ---------
        n0dscb.akamaiedge.net A 64.86.135.233 TTL = 4000
        n1dscb.akamaiedge.net A 88.221.81.194 TTL = 6000
        n2dscb.akamaiedge.net A 165.254.51.172 TTL = 8000
        n3dscb.akamaiedge.net A 23.5.164.32 TTL = 4000
        n4dscb.akamaiedge.net A 165.254.51.176 TTL = 6000
        n5dscb.akamaiedge.net A 165.254.51.167 TTL = 8000
        n6dscb.akamaiedge.net A 165.254.51.175 TTL = 4000
        n7dscb.akamaiedge.net A 165.254.51.169 TTL = 6000
```

```
Lookup  : randomA.irl
Query   : randomA.irl, type 1, TXID 0x8601
Server  : 128.194.135.82
********************************
Attempt 0 with 29 bytes... response in 2 ms with 118 bytes
  TXID 0x8601 flags 0x8400 questions 5 answers 1 authority 0 additional 0
  succeeded with Rcode = 0
  ------------ [questions] ----------
      randomA.irl type 1 class 1
      Episode.IV type 1 class 3
      A.NEW.HOPE type 1 class 3
      civil.war type 1 class 3
      spaceships type 1 class 3
  ------------ [answers] ------------
      random.irl A 1.1.1.1 TTL = 30
```

To begin with, you should 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 format 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 in 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 Windows API 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. More examples:

```
Lookup  : google.c
Query   : google.c, type 1, TXID 0xC101
Server  : 128.194.135.85
********************************
Attempt 0 with 26 bytes... response in 1670 ms with 101 bytes
  TXID 0xC101 flags 8183 questions 1 answers 0 authority 1 additional 0
  failed with Rcode = 3
```

```
Lookup  : 12.190.0.107
Query   : 107.0.190.12.in-addr.arpa, type 12, TXID 0xB621
Server  : 128.194.135.85
********************************
Attempt 0 with 43 bytes... response in 8619 ms with 43 bytes
  TXID 0xB621 flags 0x8182 questions 1 answers 0 authority 0 additional 0
  failed with Rcode = 2
```

```
Lookup  : random2.irl
Query   : random2.irl, type 1, TXID 0xA445
Server  : 128.194.135.82
********************************
Attempt 0 with 29 bytes... response in 1 ms with 512 bytes
  TXID 0xA445 flags 0xEFEF questions 61423 answers 61423 authority 61423 additional 61423
  failed with Rcode = 15
```

```
Lookup  : random9.irl
Query   : random9.irl, type 1, TXID 0x0871
Server  : 128.194.135.82
********************************
Attempt 0 with 29 bytes... response in 1 ms with 55 bytes
  TXID 0x0872 flags 0x8400 questions 1 answers 1 authority 0 additional 0
  ++ invalid reply: TXID mismatch, sent 0x0871, received 0x0872
```

```
Lookup  : randomB.irl
Query   : randomB.irl, type 1, TXID 0xB09C
Server  : 128.194.135.82
********************************
Attempt 0 with 29 bytes... socket error 10040
```

```
Lookup  : google.com
Query   : google.com, type 1, TXID 0x0813
Server  : 128.194.135.9
********************************
Attempt 0 with 28 bytes... timeout in 10000 ms
Attempt 1 with 28 bytes... timeout in 10001 ms
Attempt 2 with 28 bytes... timeout in 10000 ms
```

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 malformatted DNS responses, which are blocked by the TAMU VPN. To get around this problem, you can test your code from a 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, …, 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 (to obtain 10040 in recvfrom, attempt reading the packet into a 512-byte buffer). 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., 0xC0 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.

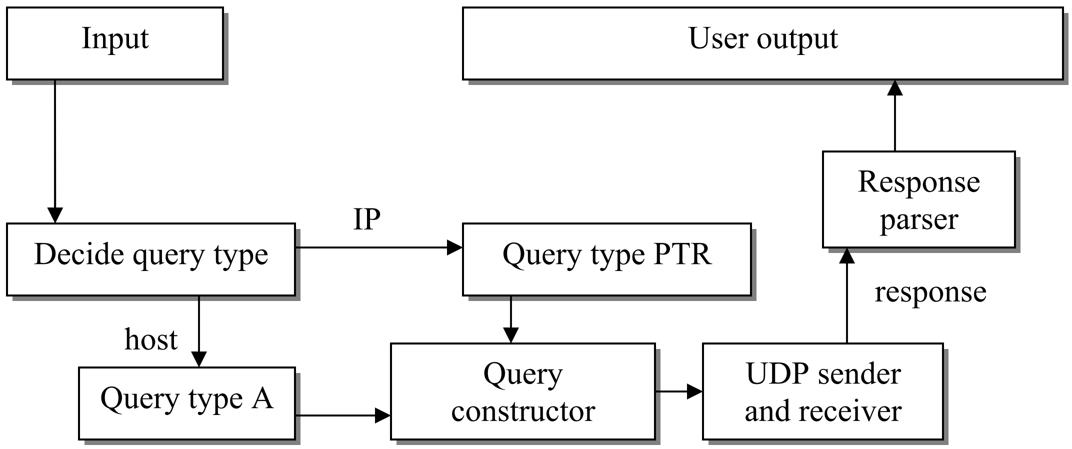

Figure 1. Flow-chart of the program.

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:

```
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:

```
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 str_1 = "www", str_2 = "google", str_3 = "com". The lengths of the corresponding strings are 3, 6, and 3 bytes. The last label has size 0 as shown in the figure.

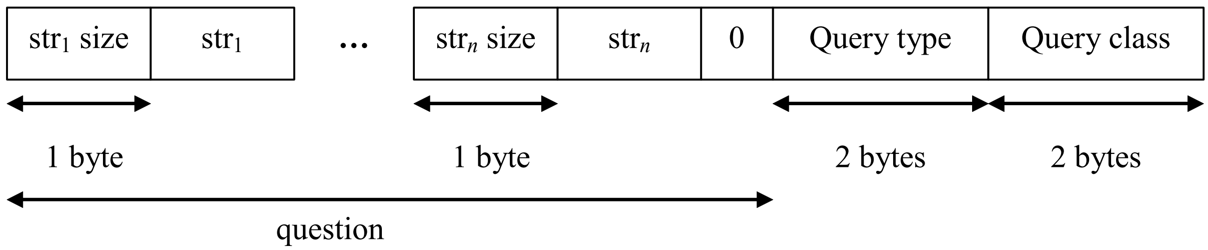

Figure 2. Question format.

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):

```
#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:

```
/* 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

```
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:

```
#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 DNSanswerHdr {
        u_short type;
        u_short class;
        u_int TTL;
        u_short len;
};
#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
DNSanswerHdr *dah = (DNSanswerHdr *)(buf + off);
// read dah->len and other fields
```

CSCE 463/612 Homework 2

Name: ______________________________

Function	Points	Break down	Item	Deduction
Printouts	58	2	No usage if incorrect arguments	
		6	Incorrect summary (lookup/query/server)	
		6	Incorrect attempt info (sent/received bytes, delay)	
		12	Incorrect fixed header fields or their format	
		2	Does not report successful Rcode	
		6	Incorrect questions (name/type/class)	
		8	Incorrect answers (name/type/value/TTL)	
		8	Incorrect authority (name/type/value/TTL)	
		8	Incorrect additional (name/type/value/TTL)	
Operation	17	3	Does not reject error Rcodes (e.g., random2.irl)	
		3	Improper or absent retransmission	
		3	Does not reject bogus TXID (e.g., random9.irl)	
		3	Fails to parse questions in randomA.irl	
		3	Does not report socket errors (e.g., randomB.irl)	
		2	Crashes on certain responses	
Report	25	8	Random.irl: 0, 3, 5, 6 (explain four ++ errors)	
		2	Random.irl: 1 (explain one ++ error)	
		3	Random.irl: 7 (explain one ++ error)	
		12	Random.irl: 4 (explain three ++ errors)	

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

Total points: ________________