

CSci 491-01: Internet Programming

Homework 2, due December 5, 2008 (100 pts)

1. Purpose

Understand how to design non-ASCII application-layer protocols and learn how to provide reliable transfer over UDP.

2. Description

2.1. Overview

Your goal is to implement a DNS resolver that runs over UDP. The user inputs strings that can be either host names or IP addresses, which need to be resolved through DNS. Your program must directly use UDP and parse DNS responses without using any shortcuts from system calls (like `getaddrinfo()`). The answers returned by the local DNS server must be displayed to the user including any additional records and multiple answers. A complete specification of packet headers and the various fields is contained in RFCs 1034-1035.

The program can be run in two modes – *interactive* (i.e., using command-line input) and *batch* (i.e., using input file `dns-in.txt`). In the former case, the code will return a detailed answer to the query provided by the user (see examples below). In the latter case, the code will read the input file (one question per line) and perform lookups using N threads, where N is specified in the command line. To distinguish between the modes, check if the first argument to the program is an integer. If so, assume this integer is the number of threads for batch lookups. Otherwise, assume the interactive mode.

Requirements for interactive lookups:

1. Your code must be able to decide whether input is an IP or a hostname based on the syntax of the string. Invalid IPs should be rejected immediately without contacting DNS (see below for more).
2. You must be able to send A and PTR requests based on user input and parse CNAME, A, and PTR responses. You must parse responses in both the *answer* section and the *additional-records* section; however, you may skip the authoritative section even if the number of answers there is non-zero.
3. You must be able to handle compressed resource records (RRs) as most DNS servers will return compressed data. The compression scheme in DNS is very simple and is documented fully in RFC 1035.
4. You must differentiate between successful lookups and failures, as well as detect errors and interpret them for the user. For example, return code 3 signifies a non-existent DNS name (print "No DNS entry") and code 2 means that the authoritative server cannot be found/contacted by your local DNS server (print "Authoritative DNS server not found"). If your program times out waiting for the local DNS server, display "Local DNS server timeout." For all other errors, print the numerical error and exit.
5. The code must be able to dynamically find the local DNS server for your computer (no hardcoding of its IP).

6. The program must not crash or exceed array boundaries under any circumstances (sanity checks for all pointers and fixed headers).

Requirements for batch lookups:

1. The main thread must read all strings from `dns-in.txt` into a shared queue `inputQ`, then start N threads, which will draw items from the queue, perform DNS lookups, write all answers into another shared queue `outputQ`, and continue looping until `inputQ` becomes empty. When all done, the main thread will write the items from `outputQ` into file `dns-out.txt`. The order of answers on output does not have to match that on input. Note that errors must be recorded as well.
2. The code must correctly implement multi-threading and synchronization on shared objects. Hint: attaching timestamps (request/response time) to each answer will allow you to reconstruct how long each query took. This info can also be used to compute the number of queries per second completed by the program.

Sample interaction:

```
$ resolver www.cnn.com
```

```
Answer(s):
```

```
www.cnn.com is aliased to cnn.com  
cnn.com is 64.236.24.28  
cnn.com is 64.236.29.120  
cnn.com is 64.236.16.20  
cnn.com is 64.236.16.52  
cnn.com is 64.236.16.84  
cnn.com is 64.236.16.116  
cnn.com is 64.236.24.12  
cnn.com is 64.236.24.20
```

```
$ resolver google.com
```

```
Answer(s):
```

```
google.com is 64.233.167.99  
google.com is 72.14.207.99  
google.com is 64.233.187.99
```

```
Additional answer(s):
```

```
ns1.google.com is 216.239.32.10  
ns2.google.com is 216.239.34.10  
ns3.google.com is 216.239.36.10  
ns4.google.com is 216.239.38.10
```

```
$ resolver www.google.commmmm
```

```
No DNS entry
```

```
$ resolver some.slow.domain
```

```
Authoritative DNS server not found
```

```
$ resolver some.weird.domain
```

```
Local DNS server timeout
```

```
$ resolver 128.194.30.1
```

```
Answer(s):
```

```
128.194.30.1 is hend-2-fowl-nb-e-3b.net.tamu.edu
```

```
$ resolver 128.194.138.12
```

```
Answer(s):
```

```
128.194.138.12 is mailhost.cs.tamu.edu
```

```
128.194.138.12 is pop.cs.tamu.edu
```

```
128.194.138.12 is imap.cs.tamu.edu
```

```
128.194.138.12 is mail.cs.tamu.edu
```

```
128.194.138.12 is pine.cs.tamu.edu
```

```
128.194.138.12 is pophost.cs.tamu.edu
```

```
$ resolver 216.239.37.99
```

```
No DNS entry
```

```
$ resolver 216.239.37.399
```

```
Invalid IP address
```

```
$ resolver 1500
```

```
Starting batch mode with 1500 threads...
```

```
Reading input file... found 4300500 entries
```

```
...
```

```
Completed 4300500 queries
```

```
    Successful: 62%
```

```
    No DNS record: 25%
```

```
    No auth DNS server: 10%
```

```
    Local DNS timeout: 3%
```

```
    Average delay: 250 ms
```

```
    Average retx attempts: 1.23
```

```
    <some additional statistics>
```

```
Writing output file... finished with 9887321 entries
```

Requirements for the report:

1. Document your code as in homework #1.
2. Show sample output from your program and its handling of all types of cases outlined above (you do not need to use the same exact input, but rather examples that are similar in spirit). To find an example for each case, perform DNS lookups on the entire list of hostnames collected in homework #1. Show several examples of hosts that produce "authoritative DNS not found" and "local DNS timeout." Find examples of IPs that are aliased to other IPs.
3. Examine the issue of packet loss by analyzing the number of times you had to transmit requests to the local DNS server before the attempt was successful (i.e., you received some response). *Note that this value is the attempt number to which the server has responded, not the total number of attempts made.* Plot a distribution similar to the one shown in Figure 1(a), except your maximum number of attempts will be 3 instead of 6. The numbers in the figure will add up to 100%.

- Show the histogram of lookup delays for all *successfully resolved* hostnames. Histograms are plotted by partitioning a dataset into fixed bins and counting the fraction of data in each bin. See Figure 1(b) for an example that uses 200-ms bins. The numbers in the figure will also add up to 100%.
- Document the percentage of successful lookups for the webpage list and compare this number to that obtained using `getaddrinfo()` in homework #1.
- Using multiple threads starting from 10 and going up to 300 in some increments, document the performance of your resolver on the webpage list (i.e., lookups per second) in comparison to `getaddrinfo()`. Specify the average CPU utilization for each case. Draw conclusions on the efficiency of your implementation. See Table 1 for an idea (numbers in the table do not necessarily have to correspond to yours).

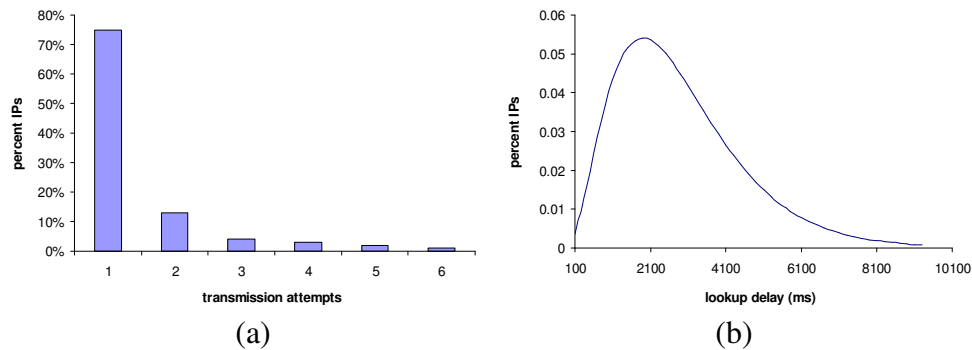


Figure 1. Sample graphs for this homework.

getaddrinfo()						
Threads	Attempted	Found	% Success	Time (sec)	Lookups/sec	CPU
10	142242	130863	92.00%	319.9	444.65	80%
...						
300	142242	130447	91.71%	209.7	678.31	100%
Your code						
Threads	Attempted	Found	% Success	Time (sec)	Lookups/sec	CPU
10	142242	131930	92.75%	108.1	1315.84	20%
...						
300	142242	131935	92.75%	103.8	1370.35	30%

Table 1. Sample comparison with `getaddrinfo()`.

2.2. Details

Your program will parse the input and first attempt to guess whether it refers to a host name or an IP address. If the user string contains only digits 0-9 and dots, assume that it is an IP address; otherwise, assume it is a host name. To check the validity of an IP address, pass it through `inet_addr()`, which returns `INADDR_NONE` if the string is not legitimate. For host names, do not perform any checks and directly supply the string to the DNS resolver. The organization of your program may look similar to the one shown in Figure 2.

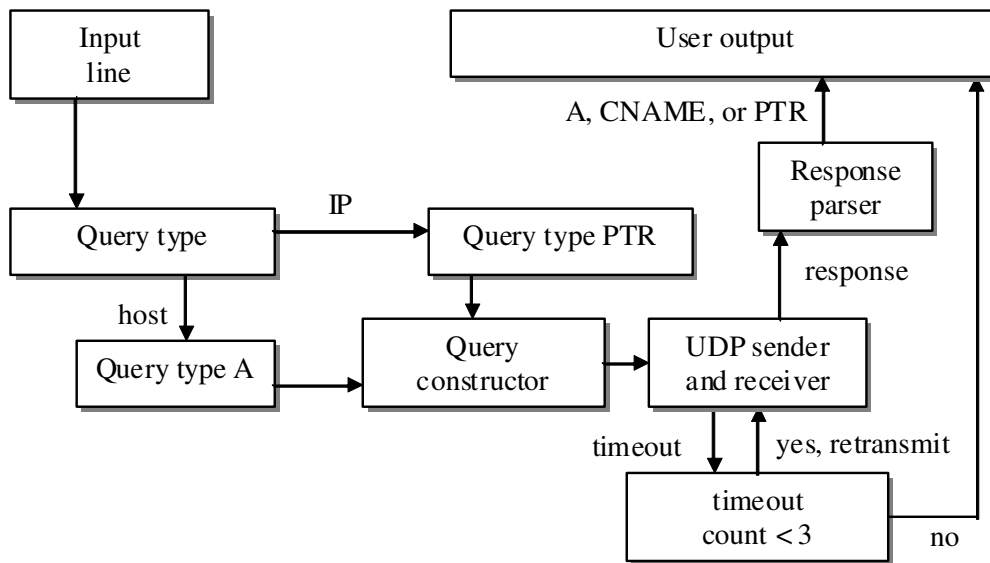


Figure 2. Flow-chart of the program.

UDP sockets are opened with `socket (AF_INET, SOCK_DGRAM, 0)` and your destination port number is 53, which should be passed to function `sendto()` when you are ready to transmit a request. After a socket is opened, bind it to port 0, which allows the OS to select the next available port for you. There is no connect phase and sockets can be used immediately after `bind()`. The basic 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 the number of queries. Set the other three fields to zero. Following these 12 bytes is the question field described below.

Each query includes a variable-size question and a *trailing* fixed-size header as shown in Figure 3. The question string is separated into substrings 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 substring is null-terminated as shown in the figure.

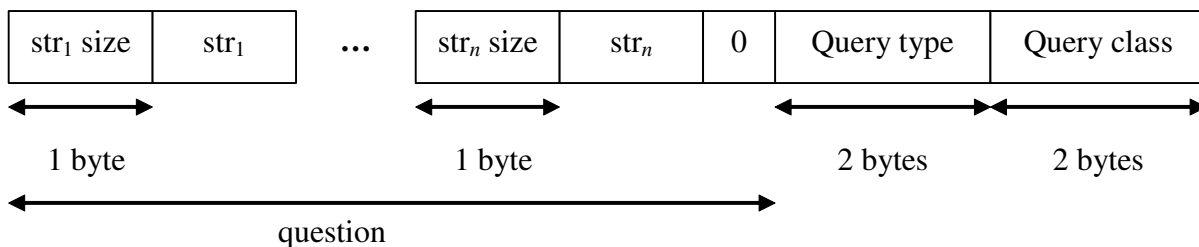


Figure 3. Question header.

There is only one useful query class:

```

/* query classes */
#define DNS_INET 1

```

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

```

/* 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 */
#define DNS_MX 15 /* mail exchange */
#define DNS_AXFR 252 /* request for zone transfer */
#define DNS_ANY 255 /* all records */

```

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 (i.e., DNS queries and replies cannot be larger than one packet). It is therefore not necessary to form a receive loop around `recvfrom()` as done in homework #1. Also note that the returned data is binary and cannot be uploaded into STL strings, which means that you must process raw char buffers as part of this homework.

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>. You can set up Wireshark filters to only display information related to DNS (i.e., port 53) to avoid clutter on the screen. See documentation on Wireshark's website.

Note that you should support both compressed and uncompressed answers. To recognize compression, check the string-size byte for being larger than 192 (i.e., the two most-significant bits are 11) and see examples in the RFC.

2.3. Packet Loss

Since not all UDP packets are reliably delivered to your local DNS server, implement a simple retransmission 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:

```

while (count++ < 3) {
    // 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
    }
    // some error checking here
}

```

In this code, `tp` is a `timeval` structure that specifies the timeout before the function returns back (it is recommended that you set the timeout to 30 seconds). If `available` is zero, the function returned after a timeout and a new request should be sent to the DNS server. Otherwise, issue a `recvfrom()` and parse the result (do not forget to check for errors returned by `select()` and `reinsert dns_sock` to `fd` between calling `select`). If you receive three timeouts in a row, abort the lookup and report the corresponding error to the user. **Do not hang forever since some DNS requests never get a response.**

Also note that in order to properly correlate responses to queries, you must use the ID field of the DNS header. Ambiguity arises when you transmit multiple copies of the same query, but then have no way of knowing which of the original questions generated the response (you need this knowledge in order to plot Figure 1). It is recommended that each thread have its own UDP socket and maintain its own counter for the ID field that is incremented with each new query packet transmitted. This also allows the program to ignore outdated replies in response to earlier questions.

2.4. Header Caveats

All numbers are coded in the network byte order and must be converted to/from your local host notation. This applies to the flags and other fields/codes below:

```
/* flags */
#define DNS_QUERY      (0 << 15)      /* 0 = query; 1 = response */
#define DNS_RESPONSE  (1 << 15)

#define DNS_STDQUERY   (0)             /* opcode - 4 bits */
#define DNS_INVQUERY   (1 << 11)
#define DNS_SRVSTATUS  (1 << 12)

#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 */

#define DNS_OK         0               /* rcode = reply codes */
#define DNS_FORMAT     1               /* format error (unable to interpret) */
#define DNS_SERVERFAIL 2               /* server failure */
#define DNS_ERROR      3               /* no DNS entry */
#define DNS_NOTIMPL    4               /* not implemented */
#define DNS_REFUSED    5               /* server refused the query */
```

For example, to set flags for query and recursion desired, use `htons(DNS_QUERY | DNS_RD)`. Note that all queries must be *standard* (i.e., `DNS_STDQUERY`) and that *inverse* queries (i.e., `DNS_INVQUERY`) are not the same as *reverse* queries (i.e., of type PTR). Most DNS servers do not support inverse queries. To obtain a reverse-DNS mapping, issue a *standard* PTR query on the string `<backwards IP>.in-addr.arpa` as recommended in RFC 1035 and discussed in class.

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

```
class queryHeader {
    u_short class;
    u_short type;
};

class fixedDNSheader {
    u_short ID;
    u_short flags;
    u_short questions;
    ...
};

fixedDNSheader dns_header;
queryHeader query_header;

// fixed field initialization
dns_header.ID =
dns_header.flags =
...

Question q;          // your class

string host = "www.google.com";
q.CreateQuestion(host);
// q.rawQuerybuffer now holds a raw request
int size = q.size() + sizeof (fixedDNSheader) +
    sizeof(queryHeader);
u_char *pkt = new u_char [size];
q.MakePacket (pkt, &dns_header, &query_header);
sendto (sock, pkt, ...);
delete pkt; q.freeRawbuffer();
```

2.5. DNS Lookup Issues

Dynamically determine in your program what your local DNS server is (parse /etc/resolv.conf).

2.6. Reading Raw Buffers

You can cast pointers into receive buffers instead of parsing results byte-by-byte:

```
class fixedRR {
    u_short type;
    u_short class;
    int TTL;
    ...
};

char buf[512];          // max packet size
recvfrom (sock, buf, ...);
fixedDNSheader *fdh = (fixedDNSheader*)buf;
// read fdh->ID and other fields
// skip over variable fields to the answer(s) section
fixedRR *frr = (fixedRR*)(buf + offset);
// read frr->type and other fields
```