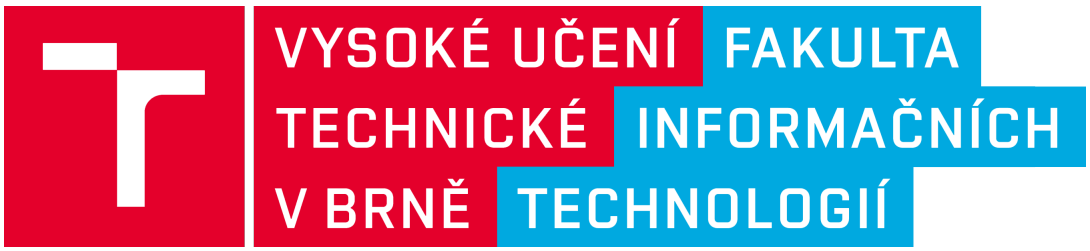

Technical documentation

Síťové aplikace a správa sítí

Variant – 3 - DNS resolver

October 31, 2023

David Novák (xnovak2r)

Contents

1 About

The aim of this project was to create a simple DNS resolver in C or C++ programming languages with my implementation being in the C programming language. The DNS resolver has to be able to send a DNS query to a server specified by either the name of the server or the server IP address.

It can send either normal or reverse query, with the former requiring a hostname to be specified and the latter requiring an IP address. It is also possible choose whether to send a recursive or non-recursive query. DNS resolved is able to send either type A, type AAAA or type PTR query. After it receives an answer back from the server, the DNS resolver parses the header of the received DNS message and if there were no errors, it proceeds to parse any resource records received. DNS resolver is able to parse the type A, type AAAA and type CNAME resource records, but it is unable to parse any other records, such as the NS type records commonly sent as authority answer.

The DNS is using UDP communication, therefore it tries to communicate with server on the default port 53, however, it is possible to specify any other valid port. The DNS resolver has been tested on Fedora OS and servers Merlin.fit.vutbr.cz and Eva.fit.vutbr.cz. It should be possible to run the DNS resolver on other linux/unix distributions as well, but due to usage of certain system specific libraries it will not function properly on Windows OS.

2 How to use DNS resolver

The DNS resolver program is called **dns** and it is created from the **dns.c** source file via included **Makefile**. After using **make** executable **dns** will be created and it is possible to run it with **[-x] [-6] [-r] [-p port] -s server address** arguments. The **-s server** and **address** are required arguments which specify the server, where DSN query will be sent and address that will be used as QNAME in DNS query respectively. Both the server and address arguments are quite benevolent in what they accept. Any server or address (for example address **fit.vutbr.cz.**) will be accepted even if given as **www.fit.vutbr.cz.**, **http(s)://www.fit.vutbr.cz.**, **fit.vutbr.cz** or a combination of these impressions.

Both server and address may be given as IP version 4 or 6, but for IP to function correctly with address, flag **-x** has to be specified. That causes the DNS resolver to do a reverse query which sends *IP.in-addr.arpa* form in case of IP version 4 or *IP.in6.apra* in case of IP version 6. If IP version 6 is used, DNS resolver will function properly even when short form of IP is given.

When the **-6** flag is used, DNS query will ask server for *AAAA* type record which corresponds to an IP version 6 address, instead of the default *A* type record which corresponds to an IP version 4.

It is possible to run program with the **-r** flag. This will cause program to do a recursive query instead of a non-recursive one.

When the **-p port** flag is used, user may specify a port. Value of port has to be a valid unsigned integer. This port will be used instead of the default port 53 when communicating with server.

It is also possible to run *make* as either **make clean**, this will delete any previously compiled

dns executable or as **make test**. Make test will compile the dns executable file and run a short bash script with 10 basic tests of the DNS resolver program.

Please note, that even though the order of *-flag* arguments does not matter, it might be necessary to pass the address as the last argument.

Please also note that depending on OS the DNS resolver is being run from, it might be necessary to specify the *-lregex* and *-lresolv* libraries in Makefile.

3 Implementation

Project is implemented in C programming language and uses standard C libraries, the regex library and libraries used for internet communication.

3.1 Arguments

Argument handling is managed by **getopt** function with switch statement resolving the individual arguments program was ran with. While implementing the handling of *-p port* arguments I encountered a strange error that my IDE (*CLion*) warned me about. For some reason there was an error while declaring **char *end;** after **case 'p':**. I later stopped worrying about this error, because it seemed like nothing was wrong while building or running the program on my Fedora OS or server Eva.fit.vutbr.cz. Unfortunately one I got to doing tests on server Merlin.fit.vutbr.cz the error resurfaced.

```
xnovak2r@merlin: ~/Dokumenty/isa_2023$ make test
gcc -Wall -Wextra -o dns dns.c -lregex
dns.c: In function ‘main’:
dns.c:568:17: error: a label can only be part of a statement and a declaration is not a statement
  568 |                 char *end;
      |                 ^~~~
dns.c:569:17: error: expected expression before ‘long’
  569 |                 long p = strtol(optarg, &end, 10);
      |                 ^~~~
dns.c:575:21: error: ‘p’ undeclared (first use in this function)
  575 |                 if (p < PORT_MIN || p > PORT_MAX) {
      |                     ^
dns.c:575:21: note: each undeclared identifier is reported only once for each function it appears in
make: *** [dns] Error 1
```

Figure 1: Error

After I added a otherwise completely unnecessary if statement that does nothing, it compiled just fine. I did not figure out why this error occurred and why adding anything in between the *case* and *char *end* declaration fixed it.

3.2 Structures

The *dns.c* DNS resolver used to have 3 structures. The first structure is still present, it is called **DNS_Header** and it contains all of the items a DNS query header should have. The second is still somewhat present and it is called **DNS_Question**. It used to have fields for the name,

type and class of the DNS question part of DNS query, however, after a lot of troubles with the name part, the name part has been removed. The third structure was called **DNS_Message** and it did contain both of the aforementioned structures.

The structure *DNS_Message* was coppied to a buffer using the function **memcpy** and DNS query was then send to server. This approach ran into two problems. The first problem was an added padding in between the structures, which ruined the structure of the DNS query. This was solved by giving the *DNS_Message* structure attribute **packed**. The other and the more significant of the two problems was the name part of *DNS_Question*. When declared as pointer to a string, the memcpy function did copy the address of the name and when it was declared as *name[length]*, determining the length was a problem.

Looking at this problem now, it might have been possible to resize the array size with functions such as realloc, but at the time I chose the easier way out by removing the name part and the *DNS_Question* and copying the *DNS_Header*, name and *DNS_Question* parts individually and therefore the structure *DNS_Message* was no longer needed.

3.3 Converting name

Since the address given by user will probably not be in a proper DNS form, ready to just be copied to buffer, function **convert_name** is called. It takes pointer to a string containing the address as an argument, because this way any changes made to the address will be propagated back to main. The address is first stripped of any unnecessary additions that it might have come with and than it is scanned for the dots that divide individual domains. Position of these domain dividers is saved to an array.

Since C programming language does not have a function, that would increases the size of an array and append value to the end of that array and since the number of domains is not known before, I do create an empty array and increase its size every time a new domain is found. This is done with function **increment_array_size** that takes as arguments a pointer to an array and a pointer to the current size to the array. The function increments the size and reallocates the memory to fit the current size.

After finding all the dots and saving their position, the dots are replaced with number that corresponds with length of the domain they precede. Once this is done, a new string is created, because to fully adhere to the DNS name convention, there needs to be length of the first domain at the start of the address. To fulfill this requirement a new string of bigger size is created, the first domain length is put as the first character of said string and a
0 character is put as the second character of the string. This is done so that the *strcat* function which is used to append the rest of the address to the new string, functions correctly.

As the last thing the new address in DNS format is duplicated to the old one.

3.4 Response parsing

The DNS resolver first parses the DNS header. This is done by function **parse_header** and uses function *memcpy* to copy the beginning of the DNS response to the structure **DNS_Header**. This allows an easy parsing of the DNS header, since all of the items of the header are neatly placed where they belong inside the structure.

Once this is done, the function checks if it did receive a response with correct ID and checks the flags. Based on the flags is created a string that will be later printed out, containing the information about whether the answer was authoritative, recursive or truncated. This string is later passed back to main through a pointer as well as the number of different responses. This function return value is the value of the last four bits of the flags item, which is the error code that server sent. Depending on the result of the query, the DNS resolver will either print out the appropriate error message and terminate or print out the aforementioned string and start parsing the rest of the answer.

Parsing of resource records is done with the function **parse_RR_and_Print** which itself calls function **print_data**. Before starting the parsing of the first resource record, the function *memmove* is called and the DNS message is moved so that it doesn't contain the header anymore. *Memmove* will be called from now on a few times so that an unparsed part of the received message is always at the beginning.

Function **parse_RR_and_Print** is called for each record as many times as it was specified in the DNS header. It uses the structure **DNS_Question** in the same way function **parse_header** used the structure **DNS_Header**, but to a lesser extend, since it doesn't contain all of the items of a resource record. **parse_RR_and_Print** first checks if the first two bits are *11*. If they are, that means that the remaining 14 bits serve as the pointer to the start of the name of the resource record. If they are not, that means that the name is starting from the beginning of this resource record. Either way, the name is printed out character by character while replacing any domain name length bytes with dots, except for the first one, which is skipped. The only exception being if the type of the resource record is not of a supported type. After that the rest of resource record parsing is relatively easy except when the beginning of data part is reached.

The data part of resource record contains length and the data themselves. To print out the data the function **print_data** is utilized. This function print out the data in a appropriate format for the resource record type.

4 Literature

MOCKAPETRIS, P. (1987, November 1). RFC 1035: Domain names - implementation and specification. IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc1035
Ksinant, V., Huitema, C., Thomson, Dr. S., & Souissi, M. (2003, October 7). RFC 3596: DNS extensions to support IP version 6. IETF Datatracker. https://datatracker.ietf.org/doc/html/rfc3596