A minimal low-level DNS resolver

Victor Gabaldon Garcia
Electronics and Computer Science
University of Southampton
vgg1n21@soton.ac.uk

***Abstract*— This paper explores the implementation of a minimal DNS server based on the original specifications. It covers the fundamental components of the DNS protocol, including the DNS message structure, header format, and resource record representation, including the mechanisms for domain name encoding. The implementation mainly focuses on parsing DNS messages received through UDP, handling queries, and sending responses back. Overall, this project demonstrates the conciseness and simplicity of DNS communication and provides a practical introduction to writing a low-level resolver.**

I. Introduction

The Domain Name System (DNS) is a hierarchical distributed name service that identifies IP addresses with human-friendly names, such as `www.example.com` [1]. The DNS plays an essential role in web browsing, particularly in cloud services and content delivery networks (CDNs), thanks to the ability to provide the IP address of the nearest server to the user [2]. The system operates using a distributed network of specialised servers called DNS servers, which can be authoritative or recursive depending on their roles and responsibilities.

For the sake of simplicity, this project's implementation follows the original specifications in RFC 1034 [3] and RFC 1035 [4], excluding more recent mechanisms like eDNS or DNS over TCP.

II. DNS Message

DNS messages are specific types of network messages used in the DNS protocol to facilitate the translation of domain names into IP addresses. There are two kinds of DNS messages: queries and responses, both of which use the same format and are distinguished by a flag bit. The top-level format of the message is divided into five sections, some of which may be empty: header, question, answer, authority, and additional. The header indicates the number of items in each section, enabling the correct interpretation of the message structure.

Header	Question	Answer	Authority	Additional

Table 1: Top-level layout of the DNS message.

A. Header

Name	Description	Bits
ID	Transaction id	16
Flags	Flags of the message	16
QDCOUNT	Number of queries in the message	16
ANCOUNT	Number of answers in the message	16
NSCOUNT	Number of authoritative records in the message	16
ARCOUNT	Number of additional records in the message	16

Table 2: Structure of the DNS header.

Name	Description	Bits
QR	Indicates if the message is a query (0) or a reply (1)	1
OPCODE	The type can be QUERY (standard query, 0), IQUERY (inverse query, 1), or STATUS (server status request, 2)	4
AA	Authoritative Answer, in a response, indicates if the DNS server is authoritative for the queried hostname	1
TC	Truncation, indicates the message was truncated due to excessive length	1
RD	Recursion desired, indicates if the client means a recursive query	1
RA	Recursion available, indicates (in a response) if the replying DNS server supports recursion	1
Z	Zero, reserved for future use	3
RCODE	Reponse code, can be NOERROR (0), FORMERR (1, Format error), SERVFAIL (2), NXDOMAIN (3, Nonexistent domain), etc	4

Table 3: Flags in the DNS message header.

The header (12 bytes) is always present in the message and has the same structure for queries and answers. It contains a unique identifier for the transaction, a set of message flags, and the number of questions, answers, authority and additional resource records (RRs) included in the message body.

The flags specify various parameters and control information that help interpret and process the DNS message correctly. These include whether it is a query or response (QR), type of DNS query (OPCODE), whether the answer is authoritative (AA), if the message was truncated (TC), if the client desires recursive resolution (RD), if the server supports recursion (RA), a series of bits reserved for future use (Z) and the status code of the DNS response (RCODE) [5].

B. Question Section

The question section is used to carry the information about what is being asked in the form of one or more queries. This fragment is present in both requests and responses and does not usually change. Each query is made of a name (QNAME), type (QTYPE), and class (QCLASS) fields [4]. The name is encoded using fragments, with each fragment's length specified by an octet at the beginning of it, and terminated by a null octet. For instance, `www.example.com` would be encoded as

03 77 77 77 **07** 65 78 61 6D 70 6C 65 **03** 63 6F 6D **00**
w w w e x a m p l e c o m

Some of the query types include A (IPv4, 1), AAAA (IPv6, 28), or CNAME (alias domain, 5).

Name	Description	Bits
QNAME	Domain name, such www.example.com	Var.
QTYPE	Type of the query, such as AAAA (28)	16
QCLASS	Class of the query, usually IN (0x0001) for the Internet	16

Table 4: Composition of a query.

C. Answer Section

Once the server has processed the query, zero, one or more resource records may be returned. The response header specifies the number of RRs included in the answer section. The answer, authority, and additional records sections all share the same format.

A resource record references the name (NAME), type (TYPE), and class (CLASS) of the query, and specifies the time to live (TTL) in seconds, followed by the data length (RDLENGTH) and the actual data (RDATA), such as an IP address.

Name	Description	Bits
NAME	Domain name, such www.example.com	Var.
TYPE	Type of the RR, such as AAAA (28)	16
CLASS	Class of the response data, usually IN (0x0001) for the Internet	16
TTL	Time interval (in seconds) that the resource may be cached for	32
RDLENGTH	Length (in octets) of the data	16
RDATA	Actual data representing the resource, e.g. 2a00:1440:4007::810d:2013	Var.

Table 5: Composition of a resource record.

An interesting aspect of RRs is how names are encoded as an offset of the DNS message using just two bytes, instead of repeating the data in QNAME. For example in `c0 0c`, the first byte marks the start of the encoding and the second one indicates that 12 bytes need to be skipped from the beginning of the message before starting to read the string.

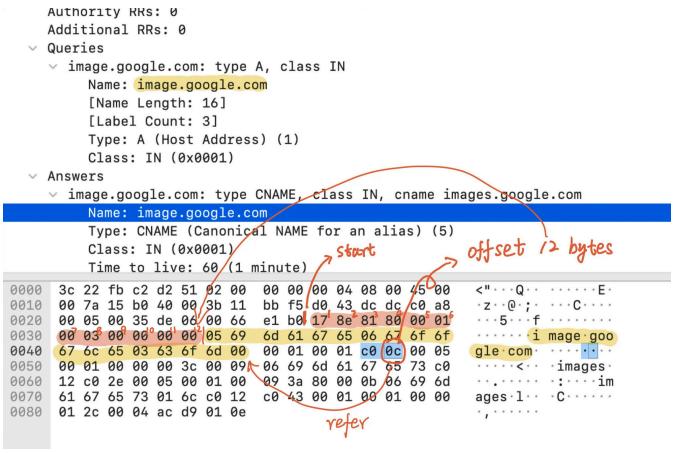

Figure 1: Overview of the NAME encoding technique, *https://cabulous.medium.com/dns-message-how-to-read-query-and-response-message-cfebcb4fe817*.

This is a smart and highly efficient way of referencing a domain name without wasting bytes in repeating data that is already in the query section.

D. Authoritative and Additional Records

Authoritative records are RRs that point to the authoritative name servers for a particular domain. These servers, maintained by the domain DNS administrators, are the official source of information and play a crucial role in the DNS by providing definitive and up-to-date records.

Additional records give supplementary information that may be helpful to the requester, such as the IP address of other name servers that are not directly requested but may help connect with authoritative servers.

III. Implementation

I have implemented a minimal low-level DNS server in Rust that processes a DNS message sent through UDP and responds with another message. When a query message is received, it is stored in a buffer, deserialised, processed, serialised, and sent back to the origin. Following the specification, multiple types and methods have been defined to represent and manipulate the message information.

The following `struct` represents the DNS message (excluding authoritative and additional records), which is shared for both queries and response messages.

```
pub struct DnsMessage {
    pub header: Header,
    pub queries: Vec<Query>,
    pub answers: Vec<Answer>,
}
```

The `Header` consists of six 16-bit fields representing the transaction ID, flags, and number of elements in each section.

```
#[derive(Debug, PartialEq, Clone, Copy, DekuRead,
DekuWrite)]
pub struct Header {
    pub transaction_id: u16,
    pub flags: Flags,
    pub question: u16,
    pub answer: u16,
    pub authority: u16,
    pub additional: u16,
}
```

The bytes received from the UDP socket are parsed into these types following a top-down approach, which enables us to handle the logic to resolve the query. In some cases, like `Flags`, the library `deku` was used to read and write data by bits to then be decoded into types like `u8` or `bool`.

```
#[derive(Debug, PartialEq, DekuRead, DekuWrite, Clone,
Copy)]
#[deku(endian = "big")]
pub struct Flags {
    #[deku(bits=1)]
    pub qr: bool,
    #[deku(bits=4)]
    pub opcode: u8,
    #[deku(bits=1)]
    pub aa: bool,
    #[deku(bits=1)]
    pub tc: bool,
    #[deku(bits=1)]
    pub rd: bool,
    #[deku(bits=1)]
    pub ra: bool,
    #[deku(bits=3)]
    pub z: u8,
    #[deku(bits=4)]
    pub rcode: u8
}
```

A struct named `Query` containing the fields `name`, `record_type`, and `class` was created to parse a query. The field `question` in `Header` indicates the number of queries to be parsed.

```
#[derive(Debug, Clone, PartialEq, DekuRead, DekuWrite)]
pub struct Query {
    pub name: Vec<u8>,
    pub record_type: u16,
    pub class: u16,
}
```

On the other hand, a separate type was created to represent an answer, including the same fields plus `ttl`, `data_length`, and `data`. Notice how the field `name` is no longer a byte array but a 16-bit unsigned integer.

```
#[derive(Debug, Clone, PartialEq, DekuRead, DekuWrite)]
pub struct Answer {
    name: u16,
    record_type: u16,
    class: u16,
    ttl: u32,
    data_length: u16,
    #[deku(count = "data_length")]
    data: Vec<u8>
}
```

As explained earlier, the name of the RRs in the answer section is encoded by offsetting the DNS message. Following the specification, I wrote a function that takes an `Answer`, the name to be encoded, and the message in bytes. Essentially, it attempts to parse a string for each possible offset and returns the encoded value if the string is equal to the provided name. The complexity is $O(n^2)$, which is not ideal and could be improved using more efficient pattern-matching techniques; however, this does not currently have a noticeable impact on performance.

```
fn encode_name(&mut self, name: &str, message: &[u8])
-> Result<u16> {
  let mut i = 0u16;

  while let Some(rest) = message.get(i as usize..) {
    if let Ok(parsed_name) = parse_name(rest) {
      if parsed_name == name {
        return Ok(0xc000 + i);
      }
    }

    i += 1;
  }

  Err(anyhow!("Name not found in message."))
}
```

For simplicity and demonstration purposes, I stored a few DNS records in a JSON file containing the domain name, the resource type, and the data. When the server receives a DNS query, it looks for the domain name in the JSON file and, if found, creates an `Answer` struct with the relevant information. The QR header flag in the original message is then flipped (to indicate the message is a response), the an-

swer count updated, and the answer sections appended before sending the message back to the origin.

To test this, we can run the implemented server on port 1053 and use the tool `dig`[1] on it to resolve the A records for `google.com`. As we can observe in Figure 2, the server successfully returned the IP address `142.250.200.46`, as written in the JSON file.

```
> dig +retry=0 -p 1053 @127.0.0.1 +noedns google.com

; <<>> DiG 9.10.6 <<>> +retry=0 -p 1053 @127.0.0.1 +noedns google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 54599
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

;; QUESTION SECTION:
;google.com.                    IN      A

;; ANSWER SECTION:
google.com.             26      IN      A       142.250.200.46

;; Query time: 1 msec
;; SERVER: 127.0.0.1#1053(127.0.0.1)
;; WHEN: Thu May 09 01:07:12 BST 2024
;; MSG SIZE  rcvd: 44
```

Figure 2: Query to the domain `google.com` using the implemented server.

```
> hexdump -C query_packet.txt
00000000  43 b0 01 20 00 01 00 00  00 00 00 00 06 67 6f 6f  |C.. .........goo|
00000010  67 6c 65 03 63 6f 6d 00  00 01 00 01              |gle.com.....|
0000001c
> hexdump -C response_packet.txt
00000000  43 b0 81 80 00 01 00 01  00 00 00 00 06 67 6f 6f  |C............goo|
00000010  67 6c 65 03 63 6f 6d 00  00 01 00 01 c0 0c 00 01  |gle.com.........|
00000020  00 01 00 00 00 1a 00 04  8e fa c8 2e              |............|
0000002c
```

Figure 3: Comparison of the query and response messages, highlighting the differences.

IV. Challenges

Throughout this project, a few difficulties arose. Getting started and understanding the DNS protocol in depth was a challenging task. After reading multiple online resources to have a broad overview of the message format, I delved into RFCs 1034 and 1035 for a deeper and more precise comprehension.

It was not easy to determine the best strategy for parsing and writing bytes. Libraries like `deku` were convenient for parsing sections like flags, where information is encoded in bits, but not as much for name encoding. In the end, custom read/write functions have been written for every section, some of which depend on the library's functions. A parser based on a byte buffer was also written to keep track of the current position.

Due to differences in endianness, some bytes in the stream were swapped, resulting in an encoding error. Each misrepresentation was closely inspected and fixed by forcing a big-endian interpretation, which is the convention in computer networks [6].

V. Conclusion and Future Work

Through this project, I have learned in detail about the DNS protocol and what information is precisely encoded in a DNS message. The implemented server currently supports a limited set of features but effectively interprets and responds with valid messages, working with DNS tools such as `dig`.

In the future, the following features could be supported:

- Storage and management of records within an actual database
- Including authoritative and additional records in the response
- Supporting other record types like CNAME, MX, and TXT
- Handling additional DNS flags such as RCODE, RD, RA, and TC, as well as authentication flags, which are not specified in RFC 1035

References

[1] H. Wu, X. Dang, L. Wang, and L. He, "Information fusion-based method for distributed domain name system cache poisoning attack detection and identification," *IET Information Security*, vol. 10, no. 1, pp. 37–44, 2016, doi: 10.1049/iet-ifs.2014.0386.

[2] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally distributed content delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, Sep. 2002, doi: 10.1109/MIC.2002.1036038.

[3] "DOMAIN NAMES - CONCEPTS AND FACILITIES," Nov. 1987. doi: 10.17487/RFC1034.

[4] "DOMAIN NAMES - IMPLEMENTATION AND SPECIFICATION," Nov. 1987. doi: 10.17487/RFC1035.

[5] "Domain Name System (DNS) Parameters." Accessed: May 07, 2024. [Online]. Available: https://www.iana.org/assignments/dns-parameters/dns-parameters.xhtml

[6] J. K. Reynolds and J. Postel, "Assigned Numbers," Oct. 1994. doi: 10.17487/RFC1700.

[1] https://en.wikipedia.org/wiki/Dig_(command)