# Let's hand write DNS messages

In this post, we'll explore the Domain Name Service (DNS) binary message format, and we'll write one by hand. This is deeper than you need to use DNS, but I think it's fun and educational to see how these things work under the hood.

We'll learn how to:

- Write binary DNS query messages
- Send our message as the body of a UDP datagram using Python
- Read the response from the DNS server

Writing binary sounds difficult, but I actually found it quite approachable. The DNS documentation is well written and easy to follow, and the message we'll write is short - only 29 bytes long.

## DNS overview

DNS is used to map human-readable domain names (such as `example.com`) to machine-readable IP addresses (like `93.184.216.34`). To use DNS, we send a query to a DNS server. This query contains the domain name we're looking up. The DNS server tries to look up that domain name's IP address in its internal data store. If it finds it, it returns it. If it can't find it, the server will forward the query to a different DNS server, which will repeat this process until the IP is found. DNS messages are usually sent using the UDP protocol.

DNS is documented in RFC 1035. All diagrams, and most of the information in this post was found in this RFC. I'd recommend referring to it if anything is unclear.

In this post, we'll use hexadecimal notation to simplify working with binary. I've added a brief note on how to convert between the two below[1].

## Request message format

All DNS messages have the same format:

```
+---------------------+
|        Header       |
+---------------------+
|       Question      | the question for the name server
+---------------------+
|        Answer       | Resource Records (RRs) answering the question
```

```
+---------------------+
|      Authority      | RRs pointing toward an authority
+---------------------+
|      Additional     | RRs holding additional information
+---------------------+
```

Query and request messages fill out different parts of the message. Our query will contain the `Header` and `Question` sections.

### Header

The header has the following format:

```
 0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      ID                       |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|QR|   Opcode  |AA|TC|RD|RA|   Z    |   RCODE   |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    QDCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ANCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    NSCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    ARCOUNT                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

In this diagram, each cell represents a single bit. In each row, there are sixteen columns, representing two bytes of data. The diagram is split into rows to make it easier to read, but the actual message is a continuous series of bytes.

As both queries and responses share a header format, some of the fields aren't relevant to our query, and will be set to `0`. A full desciption of each of these fields can be found in RFC1035 Section 4.1.1.

The fields which are relevant to us are:

- `ID`: An arbitrary 16 bit request identifier. The same ID is used in the response to the query so we can match them up. Let's go with `AA AA`.
- `QR`: A 1 bit flag specifying whether this message is a query (`0`) or a response (`1`). As we're sending a query, we'll set this bit to `0`.
- `Opcode`: A 4 bit field that specifies the query type. We're sending a standard query, so we'll set this to `0`. The possibilities are:
    - `0`: Standard query

    - `1`: Inverse query
    - `2`: Server status request
    - `3-15`: Reserved for future use

- `TC`: 1 bit flag specifying if the message has been truncated. Our message is short, and won't need to be truncated, so we can set this to `0`.

- `RD`: 1 bit flag specifying if recursion is desired. If the DNS server we send our request to doesn't know the answer to our query, it can recursively ask other DNS servers. We do wish recursion to be enabled, so we will set this to `1`.

- `QDCOUNT`: An unsigned 16 bit integer specifying the number of entries in the question section. We'll be sending `1` question.

**Full header**

Combining these fields, we can write out our header in hexadecimal:

```
AA AA - ID
01 00 - Query parameters
00 01 - Number of questions
00 00 - Number of answers
00 00 - Number of authority records
00 00 - Number of additional records
```

To get the query parameters, we concatenate the values of the fields `QR` to `RCODE`, remembering that fields not mentioned above are set to `0`. This gives `0000 0001 0000 0000`, which is `01 00` in hexadecimal. This represents a standard DNS query.

### Question

The question section has the format:

```
0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
/                     QNAME                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     QTYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     QCLASS                    |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

- `QNAME`: This contains the URL who's IP address we wish to find. It is encoded as a series of 'labels'. Each label corresponds to a section of the URL. The URL `example.com`

contains two sections, `example`, and `com`.

To construct a label, we URL-encode the section, producing a series of bytes. The label is that series of bytes, preceded by an unsigned integer byte containing the number of bytes in the section. To URL-encode our URL, we can just lookup the the ASCII code for each character.

The QNAME section is terminated with a zero byte (`00`).

- `QTYPE`: The DNS record type we're looking up. We'll be looking up `A` records, whose value is `1`.
- `QCLASS`: The class we're looking up. We're using the the internet, `IN`, which has a value of `1`.

We can write out our full question section:

```
07 65 - 'example' has length 7, e
78 61 - x, a
6D 70 - m, p
6C 65 - l, e
03 63 - 'com' has length 3, c
6F 6D - o, m
00    - zero byte to end the QNAME
00 01 - QTYPE
00 01 - QCLASS
```

An odd number of bytes are allowed in the `QNAME` section. No padding is required before the start of the `QTYPE` section.

## Sending the request

We send our DNS message as the body of a UDP request. The following Python code will take our hexadecimal DNS query, convert it to binary and send it to Google's DNS server, at the address `8.8.8.8:53`:

```
import binascii
import socket


def send_udp_message(message, address, port):
    """send_udp_message sends a message to UDP server

    message should be a hexadecimal encoded string
    """
    message = message.replace(" ", "").replace("\n", "")
```

```
    server_address = (address, port)

    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        sock.sendto(binascii.unhexlify(message), server_address)
        data, _ = sock.recvfrom(4096)
    finally:
        sock.close()
    return binascii.hexlify(data).decode("utf-8")


def format_hex(hex):
    """format_hex returns a pretty version of a hex string"""
    octets = [hex[i:i+2] for i in range(0, len(hex), 2)]
    pairs = [" ".join(octets[i:i+2]) for i in range(0, len(octets), 2)]
    return "\n".join(pairs)


message = "AA AA 01 00 00 01 00 00 00 00 00 00 " \
"07 65 78 61 6d 70 6c 65 03 63 6f 6d 00 00 01 00 01"

response = send_udp_message(message, "8.8.8.8", 53)
print(format_hex(response))
```

You can run this script by copying it into a file, `query.py` and running it from your terminal with: `$ python query.py`. It doesn't have any external dependencies, and should run with Python 2 or 3.

## Reading the response

Running the script prints out the response from the DNS server. Let's break the response into sections and see if we can work out what's happening.

### Header

The message starts out with a header, just like our query message:

```
AA AA - Same ID as before
81 80 - Different flags, we'll look at this below
00 01 - 1 question
00 01 - 1 answer
00 00 - No authority records
00 00 - No additional records
```

Let's convert `81 80` to binary:

```
8    1    8    0
1000 0001 1000 0000
```

Matching up these bits to the schema given above, we can see that:

- `QR` = `1`: This message is a response
- `AA` = `0`: This server isn't an authority for the domain name `example.com`
- `RD` = `1`: Recursion was desired for this request
- `RA` = `1`: Recursion is available on this DNS server
- `RCODE` = `0`: No errors reported

### Question section

The question section is identical to that of the query:

```
07 65 - 'example' has length 7, e
78 61 - x, a
6D 70 - m, p
6C 65 - l, e
03 63 - 'com' has length 3, c
6F 6D - o, m
00    - zero byte to end the QNAME
00 01 - QTYPE
00 01 - QCLASS
```

### Answer section

The answer section has a `resource record` format:

```
0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                                               |
/                                               /
/                      NAME                     /
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TYPE                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                     CLASS                     |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                      TTL                      |
|                                               |
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
|                    RDLENGTH                   |
```

```
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--|
/                     RDATA                     /
/                                               /
+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
```

```
C0 0C - NAME
00 01 - TYPE
00 01 - CLASS
00 00
18 4C - TTL
00 04 - RDLENGTH = 4 bytes
5D B8
D8 22 - RDDATA
```

- `NAME`: This is the URL who's IP address this response contains. This uses a compressed format:

  ```
  0  1  2  3  4  5  6  7  8  9  A  B  C  D  E  F
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  | 1  1|                OFFSET                   |
  +--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+--+
  ```

  The first two bits are set to `1`, and then the next 14 contain an unsigned interger, which counts the byte offset from the beginning of the message to a prior occurance of the same name.

  In this case, our name is `c0 0c`, which in binary is:

  ```
  1100 0000 0000 1100
  ```

  The byte offset is therefore 12. If we count through the bytes in the message, we can see that this points to the `07` at the start of the `example.com` name.

- `TYPE` and `CLASS`: These use the same naming schema as `QTYPE` and `QCLASS` above, and have the same values as above.

- `TTL`: A 32-bit unsigned integer specifying the time to live for this Response, measured in seconds. Before this time interval runs out, the result can be cached. After, it should be discarded.

- `RDLENGTH`: The byte length of the following `RDDATA` section. In this case, the length is 4.

- `RDDATA`: The data we've been looking for! These four bytes contain the four segments of our IP address: `93.184.216.34`.

## Extensions

We've seen how to construct DNS query. Here are some things you can now try:

- Construct a query for a domain name of your choice
- Query for a different record type
- Send a query with recursion disabled
- Send a query with a domain name that isn't registered

---

1. Hexadecimal (base 16) numbers are often used as a convenient shorthand for 4 bits of binary data. You can convert between the formats using this table:

| Decimal | Hex | Binary | Decimal | Hex | Binary |
|---|---|---|---|---|---|
| 0 | 0 | 0000 | 8 | 8 | 1000 |
| 1 | 1 | 0001 | 9 | 9 | 1001 |
| 2 | 2 | 0010 | 10 | A | 1010 |
| 3 | 3 | 0011 | 11 | B | 1011 |
| 4 | 4 | 0100 | 12 | C | 1100 |
| 5 | 5 | 0101 | 13 | D | 1101 |
| 6 | 6 | 0110 | 14 | E | 1110 |
| 7 | 7 | 0111 | 15 | F | 1111 |

   As you can see, we can represent any byte (8 bits) with two hexadecimal characters. ↩

---

**About the author**: James Routley is a backend developer at Monzo. For updates, follow him on Twitter or GitHub.