# Building a DNS server in Rust

The internet has a rich conceptual foundation, with many exciting ideas that enables it to function as we know it. One of the really cool ones is DNS. Before it was invented, everyone on the internet - which admittedly wasn't that many at that stage - relied on a shared file called HOSTS.TXT, maintained by the Stanford Research Institute. This file was synchronized manually through FTP, and as the number of hosts grew, so did the rate of change and the unfeasibility of the system. In 1983, Paul Mockapetris set out to find a long term solution to the problem and went on to design and implement DNS. It's a testament to his genius that his creation has been able to scale from a few thousand computers to the Internet as we know it today.

With the combined goal of gaining a deep understanding of DNS, of doing something interesting with Rust, and of scratching some of my own itches, I originally set out to implement my own DNS server. This document is not a truthful chronicle of that journey, but rather an idealized version of it, without all the detours I ended up taking. We'll gradually implement a full DNS server, starting from first principles.



# 1 - The DNS protocol

We'll start out by investigating the DNS protocol and use our knowledge thereof to implement a simple client.

Conventionally, DNS packets are sent using UDP transport and are limited to 512 bytes. As we'll see later, both of those rules have exceptions: DNS can be used over TCP as well, and using a mechanism known as eDNS we can extend the packet size. For now, we'll stick to the original specification, though.

DNS is quite convenient in the sense that queries and responses use the same format. This means that once we've written a packet parser and a packet writer, our protocol work is done. This differs from most Internet Protocols, which typically use different request and response structures. On a high level, a DNS packet looks as follows:

| Section | Size | Type | Purpose |
|---|---|---|---|
| Header | 12 Bytes | Header | Information about the query/response. |
| Question Section | Variable | List of Questions | In practice only a single question indicating the query name (domain) and the record type of interest. |
| Answer Section | Variable | List of Records | The relevant records of the requested type. |
| Authority Section | Variable | List of Records | An list of name servers (NS records), used for resolving queries recursively. |
| Additional Section | Variable | List of Records | Additional records, that might be useful. For instance, the corresponding A records for NS records. |

Essentially, we have to support three different objects: Header, Question and Record. Conveniently, the lists of records and questions are simply individual instances appended in a row, with no extras. The number of records in each section is provided by the header. The header structure looks as follows:

| RFC Name | Descriptive Name | Length | Description |
|---|---|---|---|
| ID | Packet Identifier | 16 bits | A random identifier is assigned to query packets. Response packets must reply with the same id. This is needed to differentiate responses due to the stateless nature of UDP. |
| QR | Query Response | 1 bit | 0 for queries, 1 for responses. |
| OPCODE | Operation Code | 4 bits | Typically always 0, see RFC1035 for details. |
| AA | Authoritative Answer | 1 bit | Set to 1 if the responding server is authoritative - that is, it “owns” - the domain queried. |
| TC | Truncated Message | 1 bit | Set to 1 if the message length exceeds 512 bytes. Traditionally a hint that the query can be reissued using TCP, for which the length limitation doesn’t apply. |

| RFC Name | Descriptive Name | Length | Description |
|---|---|---|---|
| RD | Recursion Desired | 1 bit | Set by the sender of the request if the server should attempt to resolve the query recursively if it does not have an answer readily available. |
| RA | Recursion Available | 1 bit | Set by the server to indicate whether or not recursive queries are allowed. |
| Z | Reserved | 3 bits | Originally reserved for later use, but now used for DNSSEC queries. |
| RCODE | Response Code | 4 bits | Set by the server to indicate the status of the response, i.e. whether or not it was successful or failed, and in the latter case providing details about the cause of the failure. |
| QDCOUNT | Question Count | 16 bits | The number of entries in the Question Section |
| ANCOUNT | Answer Count | 16 bits | The number of entries in the Answer Section |
| NSCOUNT | Authority Count | 16 bits | The number of entries in the Authority Section |
| ARCOUNT | Additional Count | 16 bits | The number of entries in the Additional Section |

The question is quite a bit less scary:

| Field | Type | Description |
|---|---|---|
| Name | Label Sequence | The domain name, encoded as a sequence of labels as described below. |
| Type | 2-byte Integer | The record type. |
| Class | 2-byte Integer | The class, in practice always set to 1. |

The tricky part lies in the encoding of the domain name, which we'll return to later.

Finally, we've got the records which are the meat of the protocol. Many record types exists, but for now we'll only consider a few essential. All records have the following preamble:

| Field | Type | Description |
|---|---|---|
| Name | Label Sequence | The domain name, encoded as a sequence of labels as described below. |
| Type | 2-byte Integer | The record type. |
| Class | 2-byte Integer | The class, in practice always set to 1. |
| TTL | 4-byte Integer | Time-To-Live, i.e. how long a record can be cached before it should be requeried. |
| Len | 2-byte Integer | Length of the record type specific data. |

Now we are all set to look a specific record types, and we'll start with the most essential: the A record, mapping a name to an ip.

| Field | Type | Description |
|---|---|---|
| Preamble | Record Preamble | The record preamble, as described above, with the length field set to 4. |
| IP | 4-byte Integer | An IP-adress encoded as a four byte integer. |

Having gotten this far, let's get a feel for this in practice by performing a lookup using the `dig` tool:

```
# dig +noedns google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +noedns google.com
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 36383
;; flags: qr rd ra ad; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

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

;; ANSWER SECTION:
google.com.             204     IN      A       172.217.18.142
```

```
;; Query time: 0 msec
;; SERVER: 192.168.1.1#53(192.168.1.1)
;; WHEN: Wed Jul 06 13:24:19 CEST 2016
;; MSG SIZE  rcvd: 44
```

We're using the `+noedns` flag to make sure we stick to the original format. There are a few things of note in the output above:

- We can see that `dig` explicitly describes the header, question and answer sections of the response packet.
- The header is using the OPCODE QUERY which corresponds to 0. The status (RESCODE) is set to NOERROR, which is 0 numerically. The id is 36383, and will change randomly with repeated queries. The Query Response (qr), Recursion Desired (rd), Recursion Available (ra). We can ignore `ad` for now, since it relates to DNSSEC. Finally, the header tells us that there is one question and one answer record.
- The question section shows us our question, with the `IN` indicating the class, and A telling us that we're performing a query for A records.
- The answer section contains the answer record, with googles IP. `204` is the TTL, IN is again the class, and A is the record type. Finally, we've got the google.com IP-adress.
- The final line tells us that the total packet size was 44 bytes.

There are still some details obscured from view here though, so let's dive deeper still and look at a hexdump of the packets. We can use `netcat` to listen on a part, and then direct `dig` to send the query there. In one terminal window we run:

```
# nc -u -l 1053 > query_packet.txt
```

Then in another window, do:

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

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +retry=0 -p 1053 @127.0.0.1 +noedns google.com
; (1 server found)
;; global options: +cmd
;; connection timed out; no servers could be reached
```

The failure is expected in this case, since `dig` will timeout when it doesn't receive a response. Since this fails, it exits. At this point `netcat` can be exited using Ctrl+C. We're left with a query packet in `packet.txt`. We can use our query packet to record a response packet as well:

```
# nc -u 8.8.8.8 53 < query_packet.txt > response_packet.txt
```

Give it a second, and the cancel using Ctrl+C. We are now ready to inspect our packets:

```
# hexdump -C query_packet.txt
00000000  86 2a 01 20 00 01 00 00  00 00 00 00 06 67 6f 6f  |.*. .........goo|
```

```
00000010  67 6c 65 03 63 6f 6d 00  00 01 00 01              |gle.com.....|
0000001c
# hexdump -C response_packet.txt
00000000  86 2a 81 80 00 01 00 01  00 00 00 00 06 67 6f 6f  |.*..........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 01 25 00 04  d8 3a d3 8e              |.....%...:..|
0000002c
```

Let's see if we can make some sense of this. We know from earlier that the header is 12 bytes long. For the query packet, the header bytes are: `86 2a 01 20 00 01 00 00 00 00 00 00` We can see that the last eight bytes corresponds to the length of the different sections, with the only one actually having any content being the question section which holds a single entry. The more interesting part is the first four bytes, which corresponds to the different fields of the header. First off, we know that we've got a 2-byte id, which is supposed to stay the same for both query and answer. Indeed we see that in this example it's set to `86 2a` in both hexdumps. The hard part to parse is the remaining two bytes. In order to make sense of them, we'll have to convert them to binary. Starting with the `01 20` of the query packet, we find (with the Most Significant Bit first):

```
0 0 0 0 0 0 0 1  0 0 1 0 0 0 0 0
- -+-+-+- - - -  - -+-+- -+-+-+-
Q    O    A T R  R   Z      R
R    P    A C D  A          C
     C                      O
     O                      D
     D                      E
     E
```

Except for the DNSSEC related bit in the `Z` section, this is as expected. `QR` is 0 since its a Query, `OPCODE` is also 0 since it's a standard lookup, the `AA`, `TC` and `RA` flags isn't relevant for queries while `RD` is set, since `dig` defaults to requesting recursive lookup. Finally, `RCODE` isn't used for queries either.

Moving on to the flag bytes of the response packet `81 80`:

```
1 0 0 0 0 0 0 1  1 0 0 0 0 0 0 0
- -+-+-+- - - -  - -+-+- -+-+-+-
Q    O    A T R  R   Z      R
R    P    A C D  A          C
     C                      O
     O                      D
     D                      E
     E
```

Since this is a response `QR` is set, and so is `RA` to indicate that the server do support recursion. Looking at the remaining eight bytes of the reply, we see that in addition to having a single question, we've also got a single answer record.

Immediately past the header, we've got the question. Let's break it down byte by byte:

```
                    query name              type   class
       ----------------------------------  -----  -----
HEX    06 67 6f 6f 67 6c 65 03 63 6f 6d 00  00 01  00 01
ASCII     g  o  o  g  l  e     c  o  m
DEC    6                    3           0      1      1
```

As outlined in the table earlier, it consists of three parts: query name, type and class. There's something interesting about the how the name is encoded, though – there are no dots present. Rather DNS encodes each name into a sequence of `labels`, with each label prepended by a single byte indicating its length. In the example above, "google" is 6 bytes and is thus preceded by `0x06`, while "com" is 3 bytes and is preceded by `0x00`. Finally, all names are terminated by a label of zero length, that is a null byte. Seems easy enough, doesn't it? Well, as we shall see soon there's another twist to it.

We've now reached the end of our query packet, but there is some data left to decode in the response packet. The remaining data is a single A record holding the corresponding IP address for google.com:

```
      name     type   class         ttl         len      ip
      ------  ------  ------  --------------  ------  --------------
HEX   c0  0c  00  01  00  01  00  00  01  25  00  04  d8  3a  d3  8e
DEC   192 12    1       1          293          4     216 58  211 142
```

Most of this is as expected: Type is 1 for `A record`, Class is 1 for `IN`, TTL in this case is 293 which seems reasonable, the data length is 4 which is as it should, and finally we learn that the IP of google is `216.58.211.142`. What then is going on with the name field? Where are the labels we just learned about?

Due to the original size constraints of DNS, of 512 bytes for a single packet, some type of compression was needed. Since most of the space required is for the domain names, and part of the same name tends to reoccur, there's some obvious space saving opportunity. For example, consider the following DNS query:

```
# dig @a.root-servers.net com

- snip -

;; AUTHORITY SECTION:
com.                172800  IN  NS      e.gtld-servers.net.
com.                172800  IN  NS      b.gtld-servers.net.
com.                172800  IN  NS      j.gtld-servers.net.
com.                172800  IN  NS      m.gtld-servers.net.
com.                172800  IN  NS      i.gtld-servers.net.
com.                172800  IN  NS      f.gtld-servers.net.
```

```
com.                 172800  IN  NS      a.gtld-servers.net.
com.                 172800  IN  NS      g.gtld-servers.net.
com.                 172800  IN  NS      h.gtld-servers.net.
com.                 172800  IN  NS      l.gtld-servers.net.
com.                 172800  IN  NS      k.gtld-servers.net.
com.                 172800  IN  NS      c.gtld-servers.net.
com.                 172800  IN  NS      d.gtld-servers.net.

;; ADDITIONAL SECTION:
e.gtld-servers.net. 172800  IN  A       192.12.94.30
b.gtld-servers.net. 172800  IN  A       192.33.14.30
b.gtld-servers.net. 172800  IN  AAAA    2001:503:231d::2:30
j.gtld-servers.net. 172800  IN  A       192.48.79.30
m.gtld-servers.net. 172800  IN  A       192.55.83.30
i.gtld-servers.net. 172800  IN  A       192.43.172.30
f.gtld-servers.net. 172800  IN  A       192.35.51.30
a.gtld-servers.net. 172800  IN  A       192.5.6.30
a.gtld-servers.net. 172800  IN  AAAA    2001:503:a83e::2:30
g.gtld-servers.net. 172800  IN  A       192.42.93.30
h.gtld-servers.net. 172800  IN  A       192.54.112.30
l.gtld-servers.net. 172800  IN  A       192.41.162.30
k.gtld-servers.net. 172800  IN  A       192.52.178.30
c.gtld-servers.net. 172800  IN  A       192.26.92.30
d.gtld-servers.net. 172800  IN  A       192.31.80.30

- snip -
```

Here we query one of the internet root servers for the name servers handling the .com TLD. Notice how `gtld-servers.net.` keeps reappearing – wouldn't it be convenient if we'd only have to include it once? One way to achieve this is to include a "jump directive", telling the packet parser to jump to another position, and finish reading the name there. As it turns out, that's exactly what we're looking at in our response packet.

I mentioned earlier that each label is preceeded by a single byte length. The additional thing we need to consider is that if the two Most Significant Bits of the length is set, we can instead expect the length byte to be followed by a second byte. These two bytes taken together, and removing the two MSB's, indicate the jump position. In the example above, we've got `0xC00C`. The bit pattern of the the two high bits expressed as hex is `0xC000` (in binary `11000000 00000000`), so we can find the jump position by xoring our two bytes with this mask to unset them: `0xC00C ^ 0xC000 = 12`. Thus we should jump to byte 12 of the packet and read from there. Recalling that the length the DNS header happens to be 12 bytes, we realize that it's instructing us to start reading from where the question part of the packet begins, which makes sense since the question starts with the query domain which in this case is "google.com". Once we've finished reading the name, we resume parsing where we left of, and move on to the record type.

### BytePacketBuffer

Now finally we know enough to start implementing! The first order of business is that we need some convenient method for manipulating the packets. For this, we'll use a `struct` called `BytePacketBuffer`.

```
pub struct BytePacketBuffer {
    pub buf: [u8; 512],
    pub pos: usize
}

impl BytePacketBuffer {

    // This gives us a fresh buffer for holding the packet contents, and a field for
    // keeping track of where we are.
    pub fn new() -> BytePacketBuffer {
        BytePacketBuffer {
            buf: [0; 512],
            pos: 0
        }
    }

    // When handling the reading of domain names, we'll need a way of
    // reading and manipulating our buffer position.

    fn pos(&self) -> usize {
        self.pos
    }

    fn step(&mut self, steps: usize) -> Result<()> {
        self.pos += steps;

        Ok(())
    }

    fn seek(&mut self, pos: usize) -> Result<()> {
        self.pos = pos;

        Ok(())
    }

    // A method for reading a single byte, and moving one step forward
    fn read(&mut self) -> Result<u8> {
```

```
    if self.pos >= 512 {
        return Err(Error::new(ErrorKind::InvalidInput, "End of buffer"));
    }
    let res = self.buf[self.pos];
    self.pos += 1;

    Ok(res)
}

// Methods for fetching data at a specified position, without modifying
// the internal position

fn get(&mut self, pos: usize) -> Result<u8> {
    if pos >= 512 {
        return Err(Error::new(ErrorKind::InvalidInput, "End of buffer"));
    }
    Ok(self.buf[pos])
}

fn get_range(&mut self, start: usize, len: usize) -> Result<&[u8]> {
    if start + len >= 512 {
        return Err(Error::new(ErrorKind::InvalidInput, "End of buffer"));
    }
    Ok(&self.buf[start..start+len as usize])
}

// Methods for reading a u16 and u32 from the buffer, while stepping
// forward 2 or 4 bytes

fn read_u16(&mut self) -> Result<u16>
{
    let res = ((try!(self.read()) as u16) << 8) |
              (try!(self.read()) as u16);

    Ok(res)
}

fn read_u32(&mut self) -> Result<u32>
{
    let res = ((try!(self.read()) as u32) << 24) |
              ((try!(self.read()) as u32) << 16) |
              ((try!(self.read()) as u32) << 8) |
              ((try!(self.read()) as u32) << 0);

    Ok(res)
}
```

```rust
// The tricky part: Reading domain names, taking labels into consideration.
// Will take something like [3]www[6]google[3]com[0] and append
// www.google.com to outstr.
fn read_qname(&mut self, outstr: &mut String) -> Result<()>
{
    // Since we might encounter jumps, we'll keep track of our position
    // locally as opposed to using the position within the struct. This
    // allows us to move the shared position to a point past our current
    // qname, while keeping track of our progress on the current qname
    // using this variable.
    let mut pos = self.pos();

    // track whether or not we've jumped
    let mut jumped = false;

    // Our delimeter which we append for each label. Since we don't want a dot at the
    // beginning of the domain name we'll leave it empty for now and set it to "." at
    // the end of the first iteration.
    let mut delim = "";
    loop {
        // At this point, we're always at the beginning of a label. Recall
        // that labels start with a length byte.
        let len = try!(self.get(pos));

        // If len has the two most significant bit are set, it represents a jump to
        // some other offset in the packet:
        if (len & 0xC0) == 0xC0 {
            // Update the buffer position to a point past the current
            // label. We don't need to touch it any further.
            if !jumped {
                try!(self.seek(pos+2));
            }

            // Read another byte, calculate offset and perform the jump by
            // updating our local position variable
            let b2 = try!(self.get(pos+1)) as u16;
            let offset = (((len as u16) ^ 0xC0) << 8) | b2;
            pos = offset as usize;

            // Indicate that a jump was performed.
            jumped = true;
        }

        // The base scenario, where we're reading a single label and
        // appending it to the output:
```

```
            else {
                // Move a single byte forward to move past the length byte.
                pos += 1;

                // Domain names are terminated by an empty label of length 0, so if the leng
                // we're done.
                if len == 0 {
                    break;
                }

                // Append the delimiter to our output buffer first.
                outstr.push_str(delim);

                // Extract the actual ASCII bytes for this label and append them to the outp

                let str_buffer = try!(self.get_range(pos, len as usize));
                outstr.push_str(&String::from_utf8_lossy(str_buffer).to_lowercase());

                delim = ".";

                // Move forward the full length of the label.
                pos += len as usize;
            }
        }

        // If a jump has been performed, we've already modified the buffer position state an
        // shouldn't do so again.
        if !jumped {
            try!(self.seek(pos));
        }

        Ok(())
    } // End of read_qname

} // End of BytePacketBuffer
```

#### ResultCode

Before we move on to the header, we'll add an enum for the values of `rescode` field:

```
#[derive(Copy,Clone,Debug,PartialEq,Eq)]
pub enum ResultCode {
    NOERROR = 0,
    FORMERR = 1,
```

```
    SERVFAIL = 2,
    NXDOMAIN = 3,
    NOTIMP = 4,
    REFUSED = 5
}

impl ResultCode {
    pub fn from_num(num: u8) -> ResultCode {
        match num {
            1 => ResultCode::FORMERR,
            2 => ResultCode::SERVFAIL,
            3 => ResultCode::NXDOMAIN,
            4 => ResultCode::NOTIMP,
            5 => ResultCode::REFUSED,
            0 | _ => ResultCode::NOERROR
        }
    }
}
```

### DnsHeader

Now we can get to work on the header. We'll represent it like this:

```
#[derive(Clone,Debug)]
pub struct DnsHeader {
    pub id: u16, // 16 bits

    pub recursion_desired: bool, // 1 bit
    pub truncated_message: bool, // 1 bit
    pub authoritative_answer: bool, // 1 bit
    pub opcode: u8, // 4 bits
    pub response: bool, // 1 bit

    pub rescode: ResultCode, // 4 bits
    pub checking_disabled: bool, // 1 bit
    pub authed_data: bool, // 1 bit
    pub z: bool, // 1 bit
    pub recursion_available: bool, // 1 bit

    pub questions: u16, // 16 bits
    pub answers: u16, // 16 bits
    pub authoritative_entries: u16, // 16 bits
    pub resource_entries: u16 // 16 bits
}
```

The implementation involves a lot of bit twiddling:

```rust
impl DnsHeader {
    pub fn new() -> DnsHeader {
        DnsHeader { id: 0,

                    recursion_desired: false,
                    truncated_message: false,
                    authoritative_answer: false,
                    opcode: 0,
                    response: false,

                    rescode: ResultCode::NOERROR,
                    checking_disabled: false,
                    authed_data: false,
                    z: false,
                    recursion_available: false,

                    questions: 0,
                    answers: 0,
                    authoritative_entries: 0,
                    resource_entries: 0 }
    }

    pub fn read(&mut self, buffer: &mut BytePacketBuffer) -> Result<()> {
        self.id = try!(buffer.read_u16());

        let flags = try!(buffer.read_u16());
        let a = (flags >> 8) as u8;
        let b = (flags & 0xFF) as u8;
        self.recursion_desired = (a & (1 << 0)) > 0;
        self.truncated_message = (a & (1 << 1)) > 0;
        self.authoritative_answer = (a & (1 << 2)) > 0;
        self.opcode = (a >> 3) & 0x0F;
        self.response = (a & (1 << 7)) > 0;

        self.rescode = ResultCode::from_num(b & 0x0F);
        self.checking_disabled = (b & (1 << 4)) > 0;
        self.authed_data = (b & (1 << 5)) > 0;
        self.z = (b & (1 << 6)) > 0;
        self.recursion_available = (b & (1 << 7)) > 0;

        self.questions = try!(buffer.read_u16());
        self.answers = try!(buffer.read_u16());
        self.authoritative_entries = try!(buffer.read_u16());
        self.resource_entries = try!(buffer.read_u16());

        // Return the constant header size
```

```
        Ok(())
    }
}
```

### QueryType

Before moving on to the question part of the packet, we'll need a way to represent the record type being queried:

```
#[derive(PartialEq,Eq,Debug,Clone,Hash,Copy)]
pub enum QueryType {
    UNKNOWN(u16),
    A, // 1
}

impl QueryType {
    pub fn to_num(&self) -> u16 {
        match *self {
            QueryType::UNKNOWN(x) => x,
            QueryType::A => 1,
        }
    }

    pub fn from_num(num: u16) -> QueryType {
        match num {
            1 => QueryType::A,
            _ => QueryType::UNKNOWN(num)
        }
    }
}
```

### DnsQuestion

The enum allows us to easily add more record types later on. Now for the question entries:

```
#[derive(Debug,Clone,PartialEq,Eq)]
pub struct DnsQuestion {
    pub name: String,
    pub qtype: QueryType
}

impl DnsQuestion {
    pub fn new(name: String, qtype: QueryType) -> DnsQuestion {
        DnsQuestion {
```

```
            name: name,
            qtype: qtype
        }
    }

    pub fn read(&mut self, buffer: &mut BytePacketBuffer) -> Result<()> {
        try!(buffer.read_qname(&mut self.name));
        self.qtype = QueryType::from_num(try!(buffer.read_u16())); // qtype
        let _ = try!(buffer.read_u16()); // class

        Ok(())
    }
}
```

Having done the hard part of reading the domain names as part of our `BytePacketBuffer` struct, it turns out to be quite compact.

### DnsRecord

We'll obviously need a way of representing the actual dns records as well, and again we'll use an enum for easy expansion:

```
#[derive(Debug,Clone,PartialEq,Eq,Hash,PartialOrd,Ord)]
#[allow(dead_code)]
pub enum DnsRecord {
    UNKNOWN {
        domain: String,
        qtype: u16,
        data_len: u16,
        ttl: u32
    }, // 0
    A {
        domain: String,
        addr: Ipv4Addr,
        ttl: u32
    }, // 1
}
```

Since there are many types of records, we'll add the ability to keep track of record types we haven't yet encountered. The enum will also allow us to easily add new records later on. The actual implementation of `DnsRecord` looks like this:

```
impl DnsRecord {

    pub fn read(buffer: &mut BytePacketBuffer) -> Result<DnsRecord> {
        let mut domain = String::new();
```

```
        try!(buffer.read_qname(&mut domain));

        let qtype_num = try!(buffer.read_u16());
        let qtype = QueryType::from_num(qtype_num);
        let _ = try!(buffer.read_u16()); // class, which we ignore
        let ttl = try!(buffer.read_u32());
        let data_len = try!(buffer.read_u16());

        match qtype {
            QueryType::A  => {
                let raw_addr = try!(buffer.read_u32());
                let addr = Ipv4Addr::new(((raw_addr >> 24) & 0xFF) as u8,
                                         ((raw_addr >> 16) & 0xFF) as u8,
                                         ((raw_addr >> 8) & 0xFF) as u8,
                                         ((raw_addr >> 0) & 0xFF) as u8);

                Ok(DnsRecord::A {
                    domain: domain,
                    addr: addr,
                    ttl: ttl
                })
            },
            QueryType::UNKNOWN(_) => {
                try!(buffer.step(data_len as usize));

                Ok(DnsRecord::UNKNOWN {
                    domain: domain,
                    qtype: qtype_num,
                    data_len: data_len,
                    ttl: ttl
                })
            }
        }
    }
}
```

### DnsPacket

Finally, let's put it all together in a struct called DnsPacket:

```
#[derive(Clone, Debug)]
pub struct DnsPacket {
    pub header: DnsHeader,
    pub questions: Vec<DnsQuestion>,
    pub answers: Vec<DnsRecord>,
```

```
    pub authorities: Vec<DnsRecord>,
    pub resources: Vec<DnsRecord>
}

impl DnsPacket {
    pub fn new() -> DnsPacket {
        DnsPacket {
            header: DnsHeader::new(),
            questions: Vec::new(),
            answers: Vec::new(),
            authorities: Vec::new(),
            resources: Vec::new()
        }
    }

    pub fn from_buffer(buffer: &mut BytePacketBuffer) -> Result<DnsPacket> {
        let mut result = DnsPacket::new();
        try!(result.header.read(buffer));

        for _ in 0..result.header.questions {
            let mut question = DnsQuestion::new("".to_string(),
                                                QueryType::UNKNOWN(0));
            try!(question.read(buffer));
            result.questions.push(question);
        }

        for _ in 0..result.header.answers {
            let rec = try!(DnsRecord::read(buffer));
            result.answers.push(rec);
        }
        for _ in 0..result.header.authoritative_entries {
            let rec = try!(DnsRecord::read(buffer));
            result.authorities.push(rec);
        }
        for _ in 0..result.header.resource_entries {
            let rec = try!(DnsRecord::read(buffer));
            result.resources.push(rec);
        }

        Ok(result)
    }
}
```

### Putting it all together

Let's use the `response_packet.txt` we generated earlier to try it out!

```
fn main() {
    let mut f = File::open("response_packet.txt").unwrap();
    let mut buffer = BytePacketBuffer::new();
    f.read(&mut buffer.buf).unwrap();

    let packet = DnsPacket::from_buffer(&mut buffer).unwrap();
    println!("{:?}", packet.header);

    for q in packet.questions {
        println!("{:?}", q);
    }
    for rec in packet.answers {
        println!("{:?}", rec);
    }
    for rec in packet.authorities {
        println!("{:?}", rec);
    }
    for rec in packet.resources {
        println!("{:?}", rec);
    }
}
```

Running it will print:

```
DnsHeader {
    id: 34346,
    recursion_desired: true,
    truncated_message: false,
    authoritative_answer: false,
    opcode: 0,
    response: true,
    rescode: NOERROR,
    checking_disabled: false,
    authed_data: false,
    z: false,
    recursion_available: true,
    questions: 1,
    answers: 1,
    authoritative_entries: 0,
    resource_entries: 0
}
DnsQuestion {
    name: "google.com",
```

```
    qtype: A
}
A {
    domain: "google.com",
    addr: 216.58.211.142,
    ttl: 293
}
```

In the next chapter, we'll add network connectivity: Chapter 2 - Building a stub resolver

# 2 - Building a stub resolver

While it's slightly satisfying to know that we're able to succesfully parse DNS packets, it's not much use to just read them off disk. As our next step, we'll use it to build a `stub resolver`, which is a DNS client that doesn't feature any built-in support for recursive lookup and that will only work with a DNS server that does. Later we'll implement an actual recursive resolver to lose the need for a server.

### Extending BytePacketBuffer for writing

In order to be able to service a query, we need to be able to not just read packets, but also write them. To do so, we'll need to extend `BytePacketBuffer` with some additional methods:

```rust
impl BytePacketBuffer {

    - snip -

    fn write(&mut self, val: u8) -> Result<()> {
        if self.pos >= 512 {
            return Err(Error::new(ErrorKind::InvalidInput, "End of buffer"));
        }
        self.buf[self.pos] = val;
        self.pos += 1;
        Ok(())
    }

    fn write_u8(&mut self, val: u8) -> Result<()> {
        try!(self.write(val));

        Ok(())
    }
```

```
    fn write_u16(&mut self, val: u16) -> Result<()> {
        try!(self.write((val >> 8) as u8));
        try!(self.write((val & 0xFF) as u8));

        Ok(())
    }

    fn write_u32(&mut self, val: u32) -> Result<()> {
        try!(self.write(((val >> 24) & 0xFF) as u8));
        try!(self.write(((val >> 16) & 0xFF) as u8));
        try!(self.write(((val >> 8) & 0xFF) as u8));
        try!(self.write(((val >> 0) & 0xFF) as u8));

        Ok(())
    }
```

We'll also need a function for writing query names in labeled form:

```
    fn write_qname(&mut self, qname: &str) -> Result<()> {

        let split_str = qname.split('.').collect::<Vec<&str>>();

        for label in split_str {
            let len = label.len();
            if len > 0x34 {
                return Err(Error::new(ErrorKind::InvalidInput, "Single label exceeds 63 char
            }

            try!(self.write_u8(len as u8));
            for b in label.as_bytes() {
                try!(self.write_u8(*b));
            }
        }

        try!(self.write_u8(0));

        Ok(())
    }

} // End of BytePacketBuffer
```

### Extending DnsHeader for writing

Building on our new functions we can extend our protocol representation structs. Starting with `DnsHeader`:

```
impl DnsHeader {

    - snip -

    pub fn write(&self, buffer: &mut BytePacketBuffer) -> Result<()> {
        try!(buffer.write_u16(self.id));

        try!(buffer.write_u8( ((self.recursion_desired as u8)) |
                              ((self.truncated_message as u8) << 1) |
                              ((self.authoritative_answer as u8) << 2) |
                              (self.opcode << 3) |
                              ((self.response as u8) << 7) as u8) );

        try!(buffer.write_u8( (self.rescode.clone() as u8) |
                              ((self.checking_disabled as u8) << 4) |
                              ((self.authed_data as u8) << 5) |
                              ((self.z as u8) << 6) |
                              ((self.recursion_available as u8) << 7) ));

        try!(buffer.write_u16(self.questions));
        try!(buffer.write_u16(self.answers));
        try!(buffer.write_u16(self.authoritative_entries));
        try!(buffer.write_u16(self.resource_entries));

        Ok(())
    }

}
```

### Extending DnsQuestion for writing

Moving on to `DnsQuestion`:

```
impl DnsQuestion {

    - snip -

    pub fn write(&self, buffer: &mut BytePacketBuffer) -> Result<()> {

        try!(buffer.write_qname(&self.name));

        let typenum = self.qtype.to_num();
        try!(buffer.write_u16(typenum));
        try!(buffer.write_u16(1));
```

```
        Ok(())
    }

}
```

#### Extending DnsRecord for writing

`DnsRecord` is for now quite compact as well, although we'll eventually add quite a bit of code here to handle different record types:

```
impl DnsRecord {

    - snip -

    pub fn write(&self, buffer: &mut BytePacketBuffer) -> Result<usize> {

        let start_pos = buffer.pos();

        match *self {
            DnsRecord::A { ref domain, ref addr, ttl } => {
                try!(buffer.write_qname(domain));
                try!(buffer.write_u16(QueryType::A.to_num()));
                try!(buffer.write_u16(1));
                try!(buffer.write_u32(ttl));
                try!(buffer.write_u16(4));

                let octets = addr.octets();
                try!(buffer.write_u8(octets[0]));
                try!(buffer.write_u8(octets[1]));
                try!(buffer.write_u8(octets[2]));
                try!(buffer.write_u8(octets[3]));
            },
            DnsRecord::UNKNOWN { .. } => {
                println!("Skipping record: {:?}", self);
            }
        }

        Ok(buffer.pos() - start_pos)
    }

}
```

#### Extending DnsPacket for writing

Putting it all together in `DnsPacket`:

```
impl DnsPacket {

    - snip -

    pub fn write(&mut self, buffer: &mut BytePacketBuffer) -> Result<()>
    {
        self.header.questions = self.questions.len() as u16;
        self.header.answers = self.answers.len() as u16;
        self.header.authoritative_entries = self.authorities.len() as u16;
        self.header.resource_entries = self.resources.len() as u16;

        try!(self.header.write(buffer));

        for question in &self.questions {
            try!(question.write(buffer));
        }
        for rec in &self.answers {
            try!(rec.write(buffer));
        }
        for rec in &self.authorities {
            try!(rec.write(buffer));
        }
        for rec in &self.resources {
            try!(rec.write(buffer));
        }

        Ok(())
    }

}
```

### Implementing a stub resolver

We're ready to implement our stub resolver. Rust includes a convenient `UDPSocket` which does most of the work.

```
fn main() {
    // Perform an A query for google.com
    let qname = "google.com";
    let qtype = QueryType::A;

    // Using googles public DNS server
    let server = ("8.8.8.8", 53);

    // Bind a UDP socket to an arbitrary port
```

```
    let socket = UdpSocket::bind(("0.0.0.0", 43210)).unwrap();

    // Build our query packet. It's important that we remember to set the
    // `recursion_desired` flag. As noted earlier, the packet id is arbitrary.
    let mut packet = DnsPacket::new();

    packet.header.id = 6666;
    packet.header.questions = 1;
    packet.header.recursion_desired = true;
    packet.questions.push(DnsQuestion::new(qname.to_string(), qtype));

    // Use our new write method to write the packet to a buffer...
    let mut req_buffer = BytePacketBuffer::new();
    packet.write(&mut req_buffer).unwrap();

    // ...and send it off to the server using our socket:
    socket.send_to(&req_buffer.buf[0..req_buffer.pos], server).unwrap();

    // To prepare for receiving the response, we'll create a new `BytePacketBuffer`,
    // and ask the socket to write the response directly into our buffer.
    let mut res_buffer = BytePacketBuffer::new();
    socket.recv_from(&mut res_buffer.buf).unwrap();

    // As per the previous section, `DnsPacket::from_buffer()` is then used to
    // actually parse the packet after which we can print the response.
    let res_packet = DnsPacket::from_buffer(&mut res_buffer).unwrap();
    println!("{:?}", res_packet.header);

    for q in res_packet.questions {
        println!("{:?}", q);
    }
    for rec in res_packet.answers {
        println!("{:?}", rec);
    }
    for rec in res_packet.authorities {
        println!("{:?}", rec);
    }
    for rec in res_packet.resources {
        println!("{:?}", rec);
    }
}
```

Running it will print:

```
DnsHeader {
    id: 6666,
    recursion_desired: true,
```

```
    truncated_message: false,
    authoritative_answer: false,
    opcode: 0,
    response: true,
    rescode: NOERROR,
    checking_disabled: false,
    authed_data: false,
    z: false,
    recursion_available: true,
    questions: 1,
    answers: 1,
    authoritative_entries: 0,
    resource_entries: 0
}
DnsQuestion {
    name: "google.com",
    qtype: A
}
A {
    domain: "google.com",
    addr: 216.58.209.110,
    ttl: 79
}
```

The next chapter covers implementing a richer set of record types: Chapter 3 - Adding more Record Types

## 3 - Adding more Record Types

Let's use our program to do a lookup for ''yahoo.com''.

```rust
let qname = "www.yahoo.com";
```

Running it yields:

```
DnsHeader {
    id: 6666,
    recursion_desired: true,
    truncated_message: false,
    authoritative_answer: false,
    opcode: 0,
    response: true,
    rescode: NOERROR,
    checking_disabled: false,
    authed_data: false,
    z: false,
```

```
    recursion_available: true,
    questions: 1,
    answers: 3,
    authoritative_entries: 0,
    resource_entries: 0
}
DnsQuestion {
    name: "www.yahoo.com",
    qtype: A
}
UNKNOWN {
    domain: "www.yahoo.com",
    qtype: 5,
    data_len: 15,
    ttl: 259
}
A {
    domain: "fd-fp3.wg1.b.yahoo.com",
    addr: 46.228.47.115,
    ttl: 19
}
A {
    domain: "fd-fp3.wg1.b.yahoo.com",
    addr: 46.228.47.114,
    ttl: 19
}
```

That's odd – we're getting an UNKNOWN record as well as two A records. The UNKNOWN record, with query type 5 is a CNAME. There are quite a few DNS record types, many of which doesn't see any use in practice. That said, let's have a look at a few essential ones:

| ID | Name | Description | Encoding |
|---|---|---|---|
| 1 | A | Alias - Mapping names to IP addresses | Preamble + Four bytes for IPv4 adress |
| 2 | NS | Name Server - The DNS server address for a domain | Preamble + Label Sequence |
| 5 | CNAME | Canonical Name - Maps names to names | Preamble + Label Sequence |
| 15 | MX | Mail eXchange - The host of the mail server for a domain | Preamble + 2-bytes for priority + Label Sequence |
| 28 | AAAA | IPv6 alias | Premable + Sixteen bytes for IPv6 adress |

### Extending QueryType with more record types

Let's go ahead and add them to our code! First we'll update our `QueryType` enum:

```
#[derive(PartialEq,Eq,Debug,Clone,Hash,Copy)]
pub enum QueryType {
    UNKNOWN(u16),
    A, // 1
    NS, // 2
    CNAME, // 5
    MX, // 15
    AAAA, // 28
}
```

We'll also need to change our utility functions.

```
impl QueryType {
    pub fn to_num(&self) -> u16 {
        match *self {
            QueryType::UNKNOWN(x) => x,
            QueryType::A => 1,
            QueryType::NS => 2,
            QueryType::CNAME => 5,
            QueryType::MX => 15,
            QueryType::AAAA => 28,
        }
    }

    pub fn from_num(num: u16) -> QueryType {
        match num {
            1 => QueryType::A,
            2 => QueryType::NS,
            5 => QueryType::CNAME,
            15 => QueryType::MX,
            28 => QueryType::AAAA,
            _ => QueryType::UNKNOWN(num)
        }
    }
}
```

### Extending DnsRecord for reading new record types

Now we need a way of holding the data for these records, so we'll make some modifications to `DnsRecord`.

```
#[derive(Debug,Clone,PartialEq,Eq,Hash,PartialOrd,Ord)]
#[allow(dead_code)]
pub enum DnsRecord {
    UNKNOWN {
        domain: String,
        qtype: u16,
        data_len: u16,
        ttl: u32
    }, // 0
    A {
        domain: String,
        addr: Ipv4Addr,
        ttl: u32
    }, // 1
    NS {
        domain: String,
        host: String,
        ttl: u32
    }, // 2
    CNAME {
        domain: String,
        host: String,
        ttl: u32
    }, // 5
    MX {
        domain: String,
        priority: u16,
        host: String,
        ttl: u32
    }, // 15
    AAAA {
        domain: String,
        addr: Ipv6Addr,
        ttl: u32
    }, // 28
}
```

Here comes the bulk of the work. We'll need to extend the functions for writing and reading records. Starting with read, we amend it with additional code for each record type. First off, we've got the common preamble:

```
pub fn read(buffer: &mut BytePacketBuffer) -> Result<DnsRecord> {
    let mut domain = String::new();
    try!(buffer.read_qname(&mut domain));

    let qtype_num = try!(buffer.read_u16());
    let qtype = QueryType::from_num(qtype_num);
```

```
let _ = try!(buffer.read_u16());
let ttl = try!(buffer.read_u32());
let data_len = try!(buffer.read_u16());

match qtype {

    // Handle each record type separately, starting with the A record
    // type which remains the same as before.
    QueryType::A  => {
        let raw_addr = try!(buffer.read_u32());
        let addr = Ipv4Addr::new(((raw_addr >> 24) & 0xFF) as u8,
                                 ((raw_addr >> 16) & 0xFF) as u8,
                                 ((raw_addr >> 8) & 0xFF) as u8,
                                 ((raw_addr >> 0) & 0xFF) as u8);

        Ok(DnsRecord::A {
            domain: domain,
            addr: addr,
            ttl: ttl
        })
    },

    // The AAAA record type follows the same logic, but with more numbers to keep
    // track off.
    QueryType::AAAA => {
        let raw_addr1 = try!(buffer.read_u32());
        let raw_addr2 = try!(buffer.read_u32());
        let raw_addr3 = try!(buffer.read_u32());
        let raw_addr4 = try!(buffer.read_u32());
        let addr = Ipv6Addr::new(((raw_addr1 >> 16) & 0xFFFF) as u16,
                                 ((raw_addr1 >> 0) & 0xFFFF) as u16,
                                 ((raw_addr2 >> 16) & 0xFFFF) as u16,
                                 ((raw_addr2 >> 0) & 0xFFFF) as u16,
                                 ((raw_addr3 >> 16) & 0xFFFF) as u16,
                                 ((raw_addr3 >> 0) & 0xFFFF) as u16,
                                 ((raw_addr4 >> 16) & 0xFFFF) as u16,
                                 ((raw_addr4 >> 0) & 0xFFFF) as u16);

        Ok(DnsRecord::AAAA {
            domain: domain,
            addr: addr,
            ttl: ttl
        })
    },

    // NS and CNAME both have the same structure.
```

```
QueryType::NS => {
    let mut ns = String::new();
    try!(buffer.read_qname(&mut ns));

    Ok(DnsRecord::NS {
        domain: domain,
        host: ns,
        ttl: ttl
    })
},

QueryType::CNAME => {
    let mut cname = String::new();
    try!(buffer.read_qname(&mut cname));

    Ok(DnsRecord::CNAME {
        domain: domain,
        host: cname,
        ttl: ttl
    })
},

// MX is almost like the previous two, but with one extra field for priority.
QueryType::MX => {
    let priority = try!(buffer.read_u16());
    let mut mx = String::new();
    try!(buffer.read_qname(&mut mx));

    Ok(DnsRecord::MX {
        domain: domain,
        priority: priority,
        host: mx,
        ttl: ttl
    })
},

// And we end with some code for handling unknown record types, as before.
QueryType::UNKNOWN(_) => {
    try!(buffer.step(data_len as usize));

    Ok(DnsRecord::UNKNOWN {
        domain: domain,
        qtype: qtype_num,
        data_len: data_len,
        ttl: ttl
    })
```

```
        }
    }
}
```

It's a bit of a mouthful, but there are no especially complicated records in their own right – it's seeing them all together that makes it look a bit unwieldy.

#### Extending BytePacketBuffer for setting values in place

Before we move on to writing records, we'll have to add two more functions to `BytePacketBuffer`:

```rust
impl BytePacketBuffer {

    - snip -

    fn set(&mut self, pos: usize, val: u8) -> Result<()> {
        self.buf[pos] = val;

        Ok(())
    }

    fn set_u16(&mut self, pos: usize, val: u16) -> Result<()> {
        try!(self.set(pos,(val >> 8) as u8));
        try!(self.set(pos+1,(val & 0xFF) as u8));

        Ok(())
    }

}
```

#### Extending DnsRecord for writing new record types

Now we can amend `DnsRecord::write`. Here's our new function:

```rust
pub fn write(&self, buffer: &mut BytePacketBuffer) -> Result<usize> {

    let start_pos = buffer.pos();

    match *self {
        DnsRecord::A { ref domain, ref addr, ttl } => {
            try!(buffer.write_qname(domain));
            try!(buffer.write_u16(QueryType::A.to_num()));
            try!(buffer.write_u16(1));
            try!(buffer.write_u32(ttl));
            try!(buffer.write_u16(4));
```

```
        let octets = addr.octets();
        try!(buffer.write_u8(octets[0]));
        try!(buffer.write_u8(octets[1]));
        try!(buffer.write_u8(octets[2]));
        try!(buffer.write_u8(octets[3]));
    },
    DnsRecord::NS { ref domain, ref host, ttl } => {
        try!(buffer.write_qname(domain));
        try!(buffer.write_u16(QueryType::NS.to_num()));
        try!(buffer.write_u16(1));
        try!(buffer.write_u32(ttl));

        let pos = buffer.pos();
        try!(buffer.write_u16(0));

        try!(buffer.write_qname(host));

        let size = buffer.pos() - (pos + 2);
        try!(buffer.set_u16(pos, size as u16));
    },
    DnsRecord::CNAME { ref domain, ref host, ttl } => {
        try!(buffer.write_qname(domain));
        try!(buffer.write_u16(QueryType::CNAME.to_num()));
        try!(buffer.write_u16(1));
        try!(buffer.write_u32(ttl));

        let pos = buffer.pos();
        try!(buffer.write_u16(0));

        try!(buffer.write_qname(host));

        let size = buffer.pos() - (pos + 2);
        try!(buffer.set_u16(pos, size as u16));
    },
    DnsRecord::MX { ref domain, priority, ref host, ttl } => {
        try!(buffer.write_qname(domain));
        try!(buffer.write_u16(QueryType::MX.to_num()));
        try!(buffer.write_u16(1));
        try!(buffer.write_u32(ttl));

        let pos = buffer.pos();
        try!(buffer.write_u16(0));

        try!(buffer.write_u16(priority));
        try!(buffer.write_qname(host));
```

```
            let size = buffer.pos() - (pos + 2);
            try!(buffer.set_u16(pos, size as u16));
        },
        DnsRecord::AAAA { ref domain, ref addr, ttl } => {
            try!(buffer.write_qname(domain));
            try!(buffer.write_u16(QueryType::AAAA.to_num()));
            try!(buffer.write_u16(1));
            try!(buffer.write_u32(ttl));
            try!(buffer.write_u16(16));

            for octet in &addr.segments() {
                try!(buffer.write_u16(*octet));
            }
        },
        DnsRecord::UNKNOWN { .. } => {
            println!("Skipping record: {:?}", self);
        }
    }

    Ok(buffer.pos() - start_pos)
}
```

Again, quite a bit of extra code, but thankfully the last thing we've got to do. We're still not using the write part, but it'll come in handy once we write our server.

#### Testing the new record types

Now we're ready to retry our ''yahoo.com" query:

```
DnsHeader {
    id: 6666,
    recursion_desired: true,
    truncated_message: false,
    authoritative_answer: false,
    opcode: 0,
    response: true,
    rescode: NOERROR,
    checking_disabled: false,
    authed_data: false,
    z: false,
    recursion_available: true,
    questions: 1,
    answers: 3,
    authoritative_entries: 0,
```

```
    resource_entries: 0
}
DnsQuestion {
    name: "www.yahoo.com",
    qtype: A
}
CNAME {
    domain: "www.yahoo.com",
    host: "fd-fp3.wg1.b.yahoo.com",
    ttl: 3
}
A {
    domain: "fd-fp3.wg1.b.yahoo.com",
    addr: 46.228.47.115,
    ttl: 19
}
A {
    domain: "fd-fp3.wg1.b.yahoo.com",
    addr: 46.228.47.114,
    ttl: 19
}
```

For good measure, let's try doing an MX lookup as well:

```
let qname = "yahoo.com";
let qtype = QueryType::MX;
```

Which yields:

```
- snip -
DnsQuestion {
    name: "yahoo.com",
    qtype: MX
}
MX {
    domain: "yahoo.com",
    priority: 1,
    host: "mta6.am0.yahoodns.net",
    ttl: 1794
}
MX {
    domain: "yahoo.com",
    priority: 1,
    host: "mta7.am0.yahoodns.net",
    ttl: 1794
}
MX {
    domain: "yahoo.com",
```

```
    priority: 1,
    host: "mta5.am0.yahoodns.net",
    ttl: 1794
}
```

Next up: Chapter 4 - Baby's first DNS server

# 4 - Baby's first DNS server

Haven gotten this far, we're ready to make our first attempt at writing an actual server. Real DNS servers come in two different varieties:

- Authoritative Server - A DNS server hosting one or more "zones". For instance, the authoritative servers for the zone google.com are ns1.google.com, ns2.google.com, ns3.google.com and ns4.google.com.
- Caching Server - A DNS server that services DNS lookups by first checking its cache to see if it already knows of the record being requested, and if not performing a recursive lookup to figure it out. This includes the DNS server that is likely running on your home router as well as the DNS server that your ISP assigns to you through DHCP, and Google's public DNS servers 8.8.8.8 and 8.8.4.4.

Strictly speaking, there's nothing to stop a server from doing both things, but in practice these two roles are typically mutually exclusive. This also explains the significance of the flags `RD` (Recursion Desired) and `RA` (Recursion Available) in the packet header – a stub resolver querying a caching server will set the `RD` flag, and since the server allows such queries it will perform the lookup and send a reply with the `RA` flag set. This won't work for an Authoritative Server which will only reply to queries relating to the zones hosted, and as such will send an error response to any queries with the `RD` flag set.

Don't take my word for it, though! Let's verify that this is the case. First off, let's use `8.8.8.8` for looking up *yahoo.com*:

```
# dig @8.8.8.8 yahoo.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +recurse @8.8.8.8 yahoo.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 53231
;; flags: qr rd ra; QUERY: 1, ANSWER: 3, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 512
;; QUESTION SECTION:
```

```
;yahoo.com.         IN  A

;; ANSWER SECTION:
yahoo.com.      1051    IN  A   98.138.253.109
yahoo.com.      1051    IN  A   98.139.183.24
yahoo.com.      1051    IN  A   206.190.36.45

;; Query time: 1 msec
;; SERVER: 8.8.8.8#53(8.8.8.8)
;; WHEN: Fri Jul 08 11:43:55 CEST 2016
;; MSG SIZE  rcvd: 86
```

This works as expected. Now let's try sending the same query to one of the servers hosting the *google.com* zone:

```
# dig @ns1.google.com yahoo.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +recurse @ns1.google.com yahoo.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: REFUSED, id: 12034
;; flags: qr rd; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

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

;; Query time: 10 msec
;; SERVER: 216.239.32.10#53(216.239.32.10)
;; WHEN: Fri Jul 08 11:44:07 CEST 2016
;; MSG SIZE  rcvd: 27
```

Notice how the status of the response says `REFUSED`! `dig` also warns us that while the `RD` flag was set in the query, the server didn't set the `RA` flag in the response. We can still use the same server for *google.com*, however:

```
dig @ns1.google.com google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +recurse @ns1.google.com google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 28058
;; flags: qr aa rd; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0
;; WARNING: recursion requested but not available

;; QUESTION SECTION:
```

```
;google.com.            IN  A

;; ANSWER SECTION:
google.com.     300 IN  A   216.58.211.142

;; Query time: 10 msec
;; SERVER: 216.239.32.10#53(216.239.32.10)
;; WHEN: Fri Jul 08 11:46:27 CEST 2016
;; MSG SIZE  rcvd: 44
```

No error this time – however, `dig` still warns us that recursion is unavailable. We can explicitly unset it using `+norecurse` which gets rid of the warning:

```
# dig +norecurse @ns1.google.com google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +norecurse @ns1.google.com google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 15850
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

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

;; ANSWER SECTION:
google.com.     300 IN  A   216.58.211.142

;; Query time: 10 msec
;; SERVER: 216.239.32.10#53(216.239.32.10)
;; WHEN: Fri Jul 08 11:47:52 CEST 2016
;; MSG SIZE  rcvd: 44
```

This final query is the type of query that we'd expect to see a caching server send as part of recursively resolving the name.

For our first foray into writing our own server, we'll do something even simpler by implementing a server that simply forwards queries to another caching server, i.e. a "DNS proxy server". Having already done most of the hard work, it's a rather quick effort!

### Separating lookup into a separate function

We'll start out by doing some quick refactoring, moving our lookup code into a separate function. This is for the most part the same code as we had in our `main` function in the previous chapter, with the only change being that we handle errors gracefully using `try!`.

```rust
fn lookup(qname: &str, qtype: QueryType, server: (&str, u16)) -> Result<DnsPacket> {
    let socket = try!(UdpSocket::bind(("0.0.0.0", 43210)));

    let mut packet = DnsPacket::new();

    packet.header.id = 6666;
    packet.header.questions = 1;
    packet.header.recursion_desired = true;
    packet.questions.push(DnsQuestion::new(qname.to_string(), qtype));

    let mut req_buffer = BytePacketBuffer::new();
    packet.write(&mut req_buffer).unwrap();
    try!(socket.send_to(&req_buffer.buf[0..req_buffer.pos], server));

    let mut res_buffer = BytePacketBuffer::new();
    socket.recv_from(&mut res_buffer.buf).unwrap();

    DnsPacket::from_buffer(&mut res_buffer)
}
```

### Implementing our first server

Now we'll write our server code. First, we need get some things in order.

```rust
fn main() {
    // Forward queries to Google's public DNS
    let server = ("8.8.8.8", 53);

    // Bind an UDP socket on port 2053
    let socket = UdpSocket::bind(("0.0.0.0", 2053)).unwrap();

    // For now, queries are handled sequentially, so an infinite loop for servicing
    // requests is initiated.
    loop {

        // With a socket ready, we can go ahead and read a packet. This will
        // block until one is received.
        let mut req_buffer = BytePacketBuffer::new();
        let (_, src) = match socket.recv_from(&mut req_buffer.buf) {
            Ok(x) => x,
            Err(e) => {
                println!("Failed to read from UDP socket: {:?}", e);
                continue;
            }
        };
```

```
// Here we use match to safely unwrap the `Result`. If everything's as expected,
// the raw bytes are simply returned, and if not it'll abort by restarting the
// loop and waiting for the next request. The `recv_from` function will write the
// data into the provided buffer, and return the length of the data read as well
// as the source adress. We're not interested in the length, but we need to keep
// track of the source in order to send our reply later on.

// Next, `DnsPacket::from_buffer` is used to parse the raw bytes into
// a `DnsPacket`. It uses the same error handling idiom as the previous statement.

let request = match DnsPacket::from_buffer(&mut req_buffer) {
    Ok(x) => x,
    Err(e) => {
        println!("Failed to parse UDP query packet: {:?}", e);
        continue;
    }
};

// Create and initialize the response packet
let mut packet = DnsPacket::new();
packet.header.id = request.header.id;
packet.header.recursion_desired = true;
packet.header.recursion_available = true;
packet.header.response = true;

// Being mindful of how unreliable input data from arbitrary senders can be, we
// need make sure that a question is actually present. If not, we return `FORMERR`
// to indicate that the sender made something wrong.
if request.questions.is_empty() {
    packet.header.rescode = ResultCode::FORMERR;
}

// Usually a question will be present, though.
else {
    let question = &request.questions[0];
    println!("Received query: {:?}", question);

    // Since all is set up and as expected, the query can be forwarded to the target
    // server. There's always the possibility that the query will fail, in which cas
    // the `SERVFAIL` response code is set to indicate as much to the client. If
    // rather everything goes as planned, the question and response records as copi
    // into our response packet.
    if let Ok(result) = lookup(&question.name, question.qtype, server) {
        packet.questions.push(question.clone());
        packet.header.rescode = result.header.rescode;
```

```
                for rec in result.answers {
                    println!("Answer: {:?}", rec);
                    packet.answers.push(rec);
                }
                for rec in result.authorities {
                    println!("Authority: {:?}", rec);
                    packet.authorities.push(rec);
                }
                for rec in result.resources {
                    println!("Resource: {:?}", rec);
                    packet.resources.push(rec);
                }
            } else {
                packet.header.rescode = ResultCode::SERVFAIL;
            }

            // The only thing remaining is to encode our response and send it off!

            let mut res_buffer = BytePacketBuffer::new();
            match packet.write(&mut res_buffer) {
                Ok(_) => {},
                Err(e) => {
                    println!("Failed to encode UDP response packet: {:?}", e);
                    continue;
                }
            };

            let len = res_buffer.pos();
            let data = match res_buffer.get_range(0, len) {
                Ok(x) => x,
                Err(e) => {
                    println!("Failed to retrieve response buffer: {:?}", e);
                    continue;
                }
            };

            match socket.send_to(data, src) {
                Ok(_) => {},
                Err(e) => {
                    println!("Failed to send response buffer: {:?}", e);
                    continue;
                }
            };
        }
    } // End of request loop
```

```
} // End of main
```

The match idiom for error handling is used again and again here, since we want to avoid terminating our request loop at all cost. It's a bit verbose, and normally we'd like to use `try!` instead. Unfortunately that's unavailable to us here, since we're in the `main` function which doesn't return a `Result`.

All done! Let's try it! We start our server in one terminal, and use `dig` to perform a lookup in a second terminal.

```
# dig @127.0.0.1 -p 2053 google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> @127.0.0.1 -p 2053 google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 47200
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

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

;; ANSWER SECTION:
google.com.     68   IN   A   216.58.211.142

;; Query time: 1 msec
;; SERVER: 127.0.0.1#2053(127.0.0.1)
;; WHEN: Fri Jul 08 12:07:44 CEST 2016
;; MSG SIZE  rcvd: 54
```

Looking at our server terminal we see:

```
Received query: DnsQuestion { name: "google.com", qtype: A }
Answer: A { domain: "google.com", addr: 216.58.211.142, ttl: 96 }
```

Success! In less than 800 lines of code, we've built a DNS server able to respond to queries with several different record types!

In the next chapter, we'll get rid of our dependence on an existing resolver: Chapter 5 - Recursive Resolve

# 5 - Recursive Resolve

Our server is working, but being reliant on another server to actually perform the lookup is annoying and less than useful. Now is a good time to dwelve into the details of how a name is really resolved.

Assuming that no information is known since before, the question is first issued to one of the Internet's 13 root servers. Why 13? Because that's how many that fits into a 512 byte DNS packet (strictly speaking, there's room for 14, but some margin was left). You might think that 13 seems a bit on the low side for handling all of the internet, and you'd be right – there are 13 logical servers, but in reality many more. You can read more about it here. Any resolver will need to know of these 13 servers before hand. A file containing all of them, in bind format, is available and called named.root. These servers all contain the same information, and to get started we can pick one of them at random. Looking at `named.root` we see that the IP-adress of *a.root-servers.net* is 198.41.0.4, so we'll go ahead and use that to perform our initial query for *www.google.com*.

```
# dig +norecurse @198.41.0.4 www.google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +norecurse @198.41.0.4 www.google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 64866
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 13, ADDITIONAL: 16

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.google.com.           IN  A

;; AUTHORITY SECTION:
com.           172800  IN  NS  e.gtld-servers.net.
com.           172800  IN  NS  b.gtld-servers.net.
com.           172800  IN  NS  j.gtld-servers.net.
com.           172800  IN  NS  m.gtld-servers.net.
com.           172800  IN  NS  i.gtld-servers.net.
com.           172800  IN  NS  f.gtld-servers.net.
com.           172800  IN  NS  a.gtld-servers.net.
com.           172800  IN  NS  g.gtld-servers.net.
com.           172800  IN  NS  h.gtld-servers.net.
com.           172800  IN  NS  l.gtld-servers.net.
com.           172800  IN  NS  k.gtld-servers.net.
com.           172800  IN  NS  c.gtld-servers.net.
com.           172800  IN  NS  d.gtld-servers.net.

;; ADDITIONAL SECTION:
e.gtld-servers.net. 172800  IN  A   192.12.94.30
b.gtld-servers.net. 172800  IN  A   192.33.14.30
b.gtld-servers.net. 172800  IN  AAAA    2001:503:231d::2:30
j.gtld-servers.net. 172800  IN  A   192.48.79.30
```

```
m.gtld-servers.net. 172800  IN  A   192.55.83.30
i.gtld-servers.net. 172800  IN  A   192.43.172.30
f.gtld-servers.net. 172800  IN  A   192.35.51.30
a.gtld-servers.net. 172800  IN  A   192.5.6.30
a.gtld-servers.net. 172800  IN  AAAA    2001:503:a83e::2:30
g.gtld-servers.net. 172800  IN  A   192.42.93.30
h.gtld-servers.net. 172800  IN  A   192.54.112.30
l.gtld-servers.net. 172800  IN  A   192.41.162.30
k.gtld-servers.net. 172800  IN  A   192.52.178.30
c.gtld-servers.net. 172800  IN  A   192.26.92.30
d.gtld-servers.net. 172800  IN  A   192.31.80.30

;; Query time: 24 msec
;; SERVER: 198.41.0.4#53(198.41.0.4)
;; WHEN: Fri Jul 08 14:09:20 CEST 2016
;; MSG SIZE  rcvd: 531
```

The root servers don't know about *www.google.com*, but they do know about *com*, so our reply tells us where to go next. There are a few things to take note of:

- We are provided with a set of NS records, which are in the authority section. NS records tells us *the name* of the name server handling a domain.
- The server is being helpful by passing along A records corresponding to the NS records, so we don't have to perform a second lookup.
- We didn't actually perform a query for *com*, but rather *www.google.com*. However, the NS records all refer to *com*.

Let's pick a server from the result and move on. *192.5.6.30* for *a.gtld-servers.net* seems as good as any.

```
# dig +norecurse @192.5.6.30 www.google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +norecurse @192.5.6.30 www.google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 16229
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 4, ADDITIONAL: 5

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
;; QUESTION SECTION:
;www.google.com.            IN  A

;; AUTHORITY SECTION:
google.com.     172800  IN  NS  ns2.google.com.
google.com.     172800  IN  NS  ns1.google.com.
```

```
google.com.     172800  IN  NS  ns3.google.com.
google.com.     172800  IN  NS  ns4.google.com.

;; ADDITIONAL SECTION:
ns2.google.com.     172800  IN  A   216.239.34.10
ns1.google.com.     172800  IN  A   216.239.32.10
ns3.google.com.     172800  IN  A   216.239.36.10
ns4.google.com.     172800  IN  A   216.239.38.10

;; Query time: 114 msec
;; SERVER: 192.5.6.30#53(192.5.6.30)
;; WHEN: Fri Jul 08 14:13:26 CEST 2016
;; MSG SIZE  rcvd: 179
```

We're still not at *www.google.com*, but at least we have a set of servers that handle the *google.com* domain now. Let's give it another shot by sending our query to *216.239.32.10*.

```
# dig +norecurse @216.239.32.10 www.google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> +norecurse @216.239.32.10 www.google.com
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 20432
;; flags: qr aa; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

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

;; ANSWER SECTION:
www.google.com.     300 IN  A   216.58.211.132

;; Query time: 10 msec
;; SERVER: 216.239.32.10#53(216.239.32.10)
;; WHEN: Fri Jul 08 14:15:11 CEST 2016
;; MSG SIZE  rcvd: 48
```

And here we go! The IP of *www.google.com* as we desired. Let's recap:

- *a.root-servers.net* tells us to check *a.gtld-servers.net* which handles com
- *a.gtld-servers.net* tells us to check *ns1.google.com* which handles google.com
- *ns1.google.com* tells us the IP of *www.google.com*

This is rather typical, and most lookups will only ever require three steps, even without caching. It's still possible to have name servers for subdomains, and further ones for sub-subdomains, though. In practice, a DNS server will maintain a cache, and most TLD's will be known since before. That means that most queries will only ever require two lookups by the server, and commonly one or zero.

### Extending DnsPacket for recursive lookups

Before we can get on, we'll need a few utility functions on `DnsPacket`.

```
impl DnsPacket {

    - snip -

    // It's useful to be able to pick a random A record from a packet. When we
    // get multiple IP's for a single name, it doesn't matter which one we
    // choose, so in those cases we can now pick one at random.
    pub fn get_random_a(&self) -> Option<String> {
        if !self.answers.is_empty() {
            let idx = random::<usize>() % self.answers.len();
            let a_record = &self.answers[idx];
            if let DnsRecord::A{ ref addr, .. } = *a_record {
                return Some(addr.to_string());
            }
        }

        None
    }

    // We'll use the fact that name servers often bundle the corresponding
    // A records when replying to an NS query to implement a function that returns
    // the actual IP for an NS record if possible.
    pub fn get_resolved_ns(&self, qname: &str) -> Option<String> {

        // First, we scan the list of NS records in the authorities section:
        let mut new_authorities = Vec::new();
        for auth in &self.authorities {
            if let DnsRecord::NS { ref domain, ref host, .. } = *auth {
                if !qname.ends_with(domain) {
                    continue;
                }

                // Once we've found an NS record, we scan the resources record for a matchi
                // A record...
                for rsrc in &self.resources {
                    if let DnsRecord::A{ ref domain, ref addr, ttl } = *rsrc {
                        if domain != host {
                            continue;
```

```
                    }

                    let rec = DnsRecord::A {
                        domain: host.clone(),
                        addr: *addr,
                        ttl: ttl
                    };

                    // ...and push any matches to a list.
                    new_authorities.push(rec);
                }
            }
        }
    }

    // If there are any matches, we pick the first one.
    if !new_authorities.is_empty() {
        if let DnsRecord::A { addr, .. } = new_authorities[0] {
            return Some(addr.to_string());
        }
    }

    None
} // End of get_resolved_ns

// However, not all name servers are as that nice. In certain cases there won't
// be any A records in the additional section, and we'll have to perform *another*
// lookup in the midst. For this, we introduce a method for returning the host
// name of an appropriate name server.
pub fn get_unresolved_ns(&self, qname: &str) -> Option<String> {

    let mut new_authorities = Vec::new();
    for auth in &self.authorities {
        if let DnsRecord::NS { ref domain, ref host, .. } = *auth {
            if !qname.ends_with(domain) {
                continue;
            }

            new_authorities.push(host);
        }
    }

    if !new_authorities.is_empty() {
        let idx = random::<usize>() % new_authorities.len();
        return Some(new_authorities[idx].clone());
    }
```

```
        None
    } // End of get_unresolved_ns

} // End of DnsPacket
```

### Implementing recursive lookup

We move swiftly on to our new `recursive_lookup` function:

```
fn recursive_lookup(qname: &str, qtype: QueryType) -> Result<DnsPacket> {

    // For now we're always starting with *a.root-servers.net*.
    let mut ns = "198.41.0.4".to_string();

    // Since it might take an arbitrary number of steps, we enter an unbounded loop.
    loop {
        println!("attempting lookup of {:?} {} with ns {}", qtype, qname, ns);

        // The next step is to send the query to the active server.
        let ns_copy = ns.clone();

        let server = (ns_copy.as_str(), 53);
        let response = try!(lookup(qname, qtype.clone(), server));

        // If there are entries in the answer section, and no errors, we are done!
        if !response.answers.is_empty() &&
           response.header.rescode == ResultCode::NOERROR {

            return Ok(response.clone());
        }

        // We might also get a `NXDOMAIN` reply, which is the authoritative name servers
        // way of telling us that the name doesn't exist.
        if response.header.rescode == ResultCode::NXDOMAIN {
            return Ok(response.clone());
        }

        // Otherwise, we'll try to find a new nameserver based on NS and a corresponding A
        // record in the additional section. If this succeeds, we can switch name server
        // and retry the loop.
        if let Some(new_ns) = response.get_resolved_ns(qname) {
            ns = new_ns.clone();

            continue;
```

```
        }

        // If not, we'll have to resolve the ip of a NS record. If no NS records exist,
        // we'll go with what the last server told us.
        let new_ns_name = match response.get_unresolved_ns(qname) {
            Some(x) => x,
            None => return Ok(response.clone())
        };

        // Here we go down the rabbit hole by starting _another_ lookup sequence in the
        // midst of our current one. Hopefully, this will give us the IP of an appropriate
        // name server.
        let recursive_response = try!(recursive_lookup(&new_ns_name, QueryType::A));

        // Finally, we pick a random ip from the result, and restart the loop. If no such
        // record is available, we again return the last result we got.
        if let Some(new_ns) = recursive_response.get_random_a() {
            ns = new_ns.clone();
        } else {
            return Ok(response.clone())
        }
    }
} // End of recursive_lookup
```

### Trying out recursive lookup

The only thing remaining is to change our main function to use `recursive_lookup`:

```
fn main() {

    - snip -

            println!("Received query: {:?}", question);
            if let Ok(result) = recursive_lookup(&question.name, question.qtype) {
                packet.questions.push(question.clone());
                packet.header.rescode = result.header.rescode;

    - snip -

}
```

Let's try it!

```
# dig @127.0.0.1 -p 2053 www.google.com

; <<>> DiG 9.10.3-P4-Ubuntu <<>> @127.0.0.1 -p 2053 www.google.com
```

```
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: NOERROR, id: 41892
;; flags: qr rd ra; QUERY: 1, ANSWER: 1, AUTHORITY: 0, ADDITIONAL: 0

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

;; ANSWER SECTION:
www.google.com.     300 IN  A   216.58.211.132

;; Query time: 76 msec
;; SERVER: 127.0.0.1#2053(127.0.0.1)
;; WHEN: Fri Jul 08 14:31:39 CEST 2016
;; MSG SIZE  rcvd: 62
```

Looking at our server window, we see:

```
Received query: DnsQuestion { name: "www.google.com", qtype: A }
attempting lookup of A www.google.com with ns 198.41.0.4
attempting lookup of A www.google.com with ns 192.12.94.30
attempting lookup of A www.google.com with ns 216.239.34.10
Answer: A { domain: "www.google.com", addr: 216.58.211.132, ttl: 300 }
```

This mirrors our manual process earlier. We can now successfully resolve a domain starting from the list of root servers. We've now got a fully functional, albeit suboptimal, DNS server.

There are many things that we could do better. For instance, there is no true concurrency in this server. We can neither send nor receive queries over TCP. We cannot use it to host our own zones, and allow it to act as an authorative server. The lack of support for DNSSEC leaves us open to DNS poisoning attacks where a malicious server can return records relating to somebody else's domain.

Many of these problems have been fixed in my own project hermes, so you can head over there to investigate how I did it, or continue on your own from here. Or maybe you've had enough of DNS for now… :) Regardless, I hope you've gained some new insight into how DNS works.