Implementing a DNS Client

In this exercise you'll implement a simple DNS client. Your client should be able to send queries to a DNS server and parse the response. By the end, you should be able to specify a domain name and query type (such as A or NS) and see the parsed output printed to the command line, as you would with a tool like `dig` or `nslookup`.

Getting Started

You can do this exercise using either high or low level APIs in Golang. We highly encourage you to use the low level APIs—ie working directly with sockets—for educational purposes, even if you might use the higher level APIs in practice. Fundamentally, you will need to create a socket, bind to a port, encode and send a query message, then receive and decode a response message. If you can perform these fundamental steps, you will be much better placed to do any kind of networking programming in any language.

NOTE

As a destination DNS server, you can use Google Public DNS at IP address 8.8.8.8, and don't forget to use the correct port!

Opening a Socket

On Unix-family operating systems, our main interface to the outside world is the socket[1]. The standard libraries of most languages provide a thin wrapper around the socket-related system calls that your operating system provides.[2] You will need to find the right APIs in Golang, keeping in mind that:

- The data will go over the internet, not to another process on the same machine;
- You will use IPv4 to keep things simple; and,
- You will use UDP.

Binding to a Port

Once you have your socket, you must also instruct the operating system to route any appropriate messages to your process. The system call for this is `bind`, and your library function is likely to be similarly named. You should not need to assign a port explicitly, although you may if you wish. If you do, be aware that you may forget to clean up after yourself and start seeing "The specified address is already in use" error messages.

Encoding the Query Message

Other than correctly using sockets, the main task in this exercise is to be able to encode and decode DNS messages to and from their binary forms. For this you will need to understand the message format in detail. We swear that the clearest description of the message format is in RFC 1035 section 4.1, with details clarified elsewhere in the RFC.[3]

You may also wish to turn on Wireshark and capture a few requests made with `dig` or `nslookup`, and examine their data. Once you have reviewed the DNS request format, start working on creating a binary DNS request yourself.

NOTE

It may be useful while testing your code, to capture traffic using Wireshark. This way you can see what data Wireshark thinks your program is sending, and what kind of reply you're receiving if any!

Decoding the Response

Once you're able to send DNS queries, your next step is to be able to read the responses. Remember that a DNS response might contain many records, and they might not all be "answers" to the query that was asked. Make sure to parse *all* of the response records.

Once again, it can be very helpful in testing your code to use Wireshark to examine the responses, and compare what Wireshark reads to what your client program is reading. You may also want to strive to have your printed output to resemble that of `dig`.

Stretch Goals

- Add any assertions that might increase your confidence that you're doing things correctly, such as in relation to the flags set in the response.
- Start with 'A' and 'NS' type records and see what it takes to add any others.

- To start, support only recursive queries. As a stretch goal, extend your implementation to support iterative queries.
- Research any flags, types or classes you see in the RFC with which you're not familiar.

hello@bradfieldcs.com