Resplendent RPCs

Entire Assignment due **2019-04-24 23:59**
Graded files:

- dns_query_svc_impl.c
- dns_query_clnt_impl.c

Content

Learning Objectives

The learning objectives for Resplendent RPCs are:

- Learn about RPCs with rpcgen
- Learn basic UDP networking
- Learn about DNS

Warning

Before you begin this lab, remember to run `rpcinfo`, then `rpcgen dns_query.x`! This call generates a C version of `dns_query.x`, the server stub, and the client stub. If you do not run this command, your implementations will not compile with `make` (https://linux.die.net/man/3/make)

Background

This lab will serve as an introduction to remote procedure calls and UDP. Remote procedure calls, as the name suggests, are a way to execute a procedure (in C, a function) that exists in a different address space and in any language. In this lab, our client in C executes a procedure in C, but it could execute a procedure written in python as well.

In order to accomplish this, the server will agree to take in a 'request' or 'query' from a client and send a 'response' back, and the client will agree to send a 'request' or 'query' and receive a 'response'. Actually implementing an RPC framework is really hard. Marshalling, unmarshalling, and keeping data formats consistent could easily be an MP! Most of the time software engineers use an existing RPC format like `protobuf` `thrift`. For the purposes of this lab we chose to use `rpcgen` (https://linux.die.net/man/1/rpcgen). In rpcgen, the request and response are defined in a .x file in XDR, which is a language that is useful for encoding data that is to be transferred between different machines. It is easy to translate this file into any language, so in this lab it is translated into a C-language .h file with rpcgen. You don't need to worry about implementing this, we've done it for you.

Server and client stubs can be generated as well from this .x file in any language (both in C in this lab), and these stubs deal with the networking between the server and client. The implementation logic, AKA what the server and client decide to do with these requests and responses they send back and forth, is left up to you. Effectively, using RPCs abstracts away

the complex networking calls that are usually necessary to complete this task (i.e. it's one layer above TCP/UDP). The only thing you have to do is fill out the structs defined in the .x file to create a response

Overview

In this lab, we will create a client that queries a server for an IPv4 address for a given domain over RPC. The server that receives this query over RPC sends a UDP packet to an `nameserver` which will send back an IP Address – aka a DNS Proxy server. Note: This *isn't* actually how the true DNS protocol works. The true protocol has different resolvers for different parts of the domain. For example a domain like `www.google.com` will first contact the `.com` server, then get routed to the `google.com` server and then get routed to the `www.google.com` server.

Failure

In this lab, if a server ever fails to retrieve a result, it will return "`-1.-1.-1.-1`". Make sure to check for this and set the success field in the server response accordingly! Like in chatroom's `read_all_from_socket` / `write_all_to_socket`, remember to check errno when sendto() and recvfrom() return -1; and errno = EINTR, meaning you have to restart your call.

Why UDP?

In chatroom, a TCP connection made sense because you wanted to stay in contact with the chatting server for a long time, never lose messages, and receive messages in a particular order. When it comes to querying nameservers for DNS resolution in this lab, you only send one packet and receive one packet per server and then you're done. The overhead in using TCP for this purpose is not necessary, so we opt to use the faster and less complex UDP.

What to write

More information in the header files – this is the high level overview.

`dns_query_clnt_impl.c`

- `resolve_hostname()` Given a host and a dns server, figures out what IP this hostname points to or reports if it couldn't find the hostname.

`dns_query_svc_impl.c`

- `create_response()` Given a few parameters, allocates all needed fields for the `response` object on the heap
- `contact_nameserver()` Given all appropriate parameters, sends a UDP Packet to the nameserver containing the hostname and the server sends back an IP address or sentinel value (check the error section above).

Testing

To make a nameservers for testing, run `./authoritative_nameservers` which by default runs on port 9000. This will let you fully complete a DNS lookup of the IPv4 addresses of various websites (check `hosts.txt`) given that you have a complete implementation. This shouldn't be necessary to create a complete implementation, but if you want to test your implementation out with other domains, you can supply your own files as arguments. Check the usage of `./authoritative_nameservers` with the -h flag. Make sure to follow the format exactly and make sure that you reference the correct & existing servers and ports in the files you create!

Example

This command will set up a nameserver:

```
$ ./authoritative_nameserver
Nameserver running on port 9000
```

Run your server with environment variables:

```
$ NAMESERVER_HOST=127.0.0.1 NAMESERVER_PORT=9000 ./server
```

Then send a query for a domain through the client. The client requires two arguments. The first is the IP address of the server that will accept its RPC and the second is the domain to resolve.

```
$ ./client 127.0.0.1 www.google.com
www.google.com has ipv4 address <...>
```

See if your client interprets the correct input when

- A hostname is in none of the caches or the nameserver
- A hostname is in none of the caches but on the nameserver
- A hostname is in the server not client cache
- A hostname is in the client cache
- Makes sure that your client works with your server

Things you don't need to worry about:

- Don't worry about retransmitting UDP packets.
- For this lab we won't be testing memory as intensively

Extra: Tech Descriptions

DNSSec

The motivation for this type of protocol is influenced by (https://en.wikipedia.org/wiki/Domain_Name_System_Security_Exte
(usually) sends your host name request to a server unencrypted, and has no way of verifying that the sent-to server is an actual server. Secondly, when your computer gets a response back, it has no way of verifying that the response was sent from that server. This is all because UDP packets are plain text and can be spoofed – so the first leg of this connection matters a *lot*. The other parts of the connection don't need to matter as much because often they are on your Internet Service Provider's internal, trusted network.

The benefit of our proxy server is that we can establish a secure TLS/TCP (Encrypted TCP connection) with a server and verify that the server is a DNS server with a Certificate. Then all hostnames sent to and from the server will be hidden from prying eyes and it will be nearly impossible to forge requests.

- DNS communications usually happen on port 53, so records don't need to specify a port to send packets to (unlike in this lab)
- IPv4 addresses in a cache need to have a valid time to live (they can't be too old or you have to requery!)
- DNS has more details; you can do queries that specifically ask for a nameserver or a mailserver, for example.
- There can be multiple IP addresses for one domain, but not in this lab.
- This lab only covers the case where the domain is defined as [something].[something].[top-level]., but as you know there can be more subdomains as well e.g. www.cs.illinois.edu
- DNS has some weak points as it is. A particular nameserver can be DDOSed so that all caches of a domain expire and no one can access it. You can mess with any computer's DNS resolution so that it returns invalid IPs for a domain. A good way to mess with your friends is making it so that www.google.com always directs to an IP address for www.bing.com.
- Mentioned before, but keep in mind how simple it is to send data between 2 programs written in 2 different languages with RPCs
- If you're curious about more modern frameworks for RPCs, check out gRPC (https://grpc.io/).

Extra: DNS Resolution Overview

As you know, domains are not useful as-is; they need to be translated into an IP address. The IP addresses are not stored all at one place; they are distributed hierarchically through multiple servers responsible for them, called authoritative nameservers. A recursive DNS server (which is usually provided by your ISPs) does the work of contacting all the necessary nameservers in this hierarchy. Here's an overview of how it does this: First it goes to the root nameserver, the server responsible for '.' AKA the root of all domains (fun fact; all domains have an implicit . at the end, so www.microsoft.com is also www.microsoft.com.). This server will hold the nameservers responsible for top-level domains like .com, .org, and .net, so the appropriate nameserver is returned to the recursive server. The recursive server then goes to the top-level nameserver it just got, which is responsible for holding nameservers responsible for domains registered under the top-level domain, so .com is responsible for nameservers for reddit.com, tumblr.com, twitter.com, etc, so the appropriate nameserver is returned. The nameserver just received is authoritative for your initial domain, so when you contact it it will return your final answer! The recursive server returns this final answer and the query is finished. This recursive DNS server is the server you will implement in the lab.