Design and implementation of DNS Client and Server

Course Title: \_*Internet Application\_*

Name: *\_\_\_\_LiYuqing\_\_\_\_\_\_\_\_*

*LiRunyuan \_*

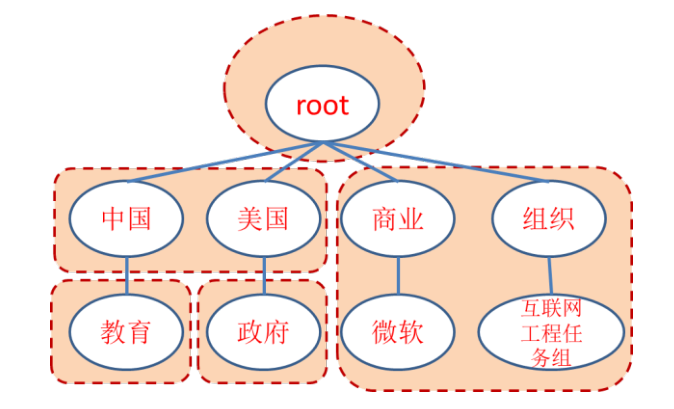
Date: 20180618

# 1. Overview

The main target of this course project is to obtain an in-depth understanding of the Domain Name System (DNS) protocol, which is used primarily to find out the IP address of a computer, given its domain or hostname, so that people could associate meaningful names to computers, instead of remembering their IP addresses. DNS protocol is also used for reverse lookup, to find the domain name of a computer by its IP address, or getting mail server’s domain name or getting the actual domain name for an aliased domain name and so forth.

By summarizing and generalizing the previous lab experiences, it’s required by this course project, that a DNS client program and a DNS server program should be developed on Linux platform to implement the DNS protocol.

* The desired DNS protocol is specialized only for the Chinese domain resolving and must support at least 4 Top-Level Domain (TLD) and at least three-level domain hierarchy. The sample server hierarchy tree is illustrated as follows:



* The DNS protocol must support 3 explicit Resource Record types, i.e. A, MX and CNAME. The IP address should be included in the Additional Section of the return record for MX type query.
* DNS servers have to support iterative resolving schema.
* The implemented DNS servers must have the capability of caching the queried records and print them out.
* TCP protocol should be used between the DNS client and the local DNS server; UDP protocol should be used among the DNS servers.
* All DNS packets are supposed to be captured and correctly identified by Wireshark.
* Database for DNS servers might be file-based.
* Extra optional features:
  + Resource Record of PTR type.
  + DNS servers support recursive resolving schema.
  + Multiple Query Question within a single request.

# 2. Requirements Analysis

## 2.1 Development Environment

The OS is an Ubuntu 16.04 machine run as a client in VirtualBox 5.2.10 which is hosted on a Windows 10 machine. The programming language is C, which will be compiled by GCC 5.4.0 and debugged by GDB 7.11.1. Besides, Wireshark 2.2.6 is also installed in the same client machine and properly configured to assist the developing procedure.

## 2.2 Functional Requirements in Details

To implement the DNS protocol, the particular two-sided communication, DNS client and DNS server as separated program are necessary to be implemented.

### 2.2.1 DNS Client

The DNS client is relatively easy to be implemented. It only needs to receive a local server address (the port will always be 53) and queries from the command line, and formulate the DNS query packet using the queries, after that, create the TCP socket connection to the specified local server, send the formulated packet and then wait for the feedback. Once the local server delivers a packet back to the client, it will be parsed and print out to the stdout by the client, after all operations are done, the client will quit.

The difficult part for client implementation is to design and use appropriate data structures to collect the queries from the command line and then formulate them into a DNS query packet. For this purpose, three types of linked list should be designed to store respectively Domain Names, Questions and Resource Records. Even though the client should only be responsible for filling up the Questions, but it also needs to print out the packet received from the local server, which will likely contain both Questions and Resource Records. After all the query information is stored in the linked list, an encoding procedure should be present to encapsulate the information into a legal DNS packet. Furthermore, a decoding procedure is also needed to extract different fields from a received packet that is returned by the local server.

There is no file I/O associated with the DNS client. The user just runs the client on the fly, no data needs to be cached for next time use.

### 2.1.2 DNS Server

The DNS server is more complicated compared to the client. It needs to play two different roles, one is the local server which is closer to the place where the client is running, another is the remote server which is often located far away on the internet.

The local DNS server is supposed to run in a recursive fashion, and do a two-sided communication, talk to the client with TCP protocol and talk to the remote server with UDP protocol, on the contrary, the remote DNS server is supposed to run iteratively, and only utilizing the UDP protocol for a one-sided communication. Which means there must be two mechanisms to implement two logic flows in the server program. One or more flags should be passed into the server program from the command line to tell the program what its role is and what schema it should stick to, recursive or iterative.

The query chain has to be implemented, to allow the server to lookup the required record in order, firstly it will peek at its own database, if no match found then it turns to the cache, if still no match found, it will give a try in the authority database. For the local server, there is a recursive process involved, in which the server will send query to other remote servers and bring back the reply, if the reply contains the result to the original query from the client, then the recursive process stops, otherwise it continues to send query to a new remote server, whose information is indicated by the last packet that’s returned.

The linked lists and the encoding/decoding procedures defined for the client could be reused for the server as the it also need to parse the packet to extract information from it plus formulate the packet from the scratch or after altering some fields in it.

There is going to have a lot of file I/O during the run of the server, e.g. read the record database file, query and update the cache file. The properly designed data formats are the key to ease the ‘painful’ I/O operations implementations.

# 3. Preliminary Design

## 3.1 Design of Data Structure

#### 3.1.1 Domain-Name type

Linked list node, contains the domain name and its length.

#### 3.1.2 Question type

Linked list node, contains the domain-name list and the question type and class.

#### 3.1.3 Resource-Record type

Linked list node, contains the domain-name list and the RR type, class, TTL and data-length and data.

#### 3.1.4 Resource-Data type

Union structure, encapsulated inside Resource-Record type to present the data.

#### 3.1.5 Message type

All-in-one structure, wrap all fields required for a DNS packet.

#### 3.1.6 Other Utility Types

Utility types that’s not listed above.

## 3.2 Decomposition of Functional Modules

#### 3.2.1 Common Module

This module will contain the type declarations and functions that are shared across client and server.

Basically, this module will mainly own the encoding and decoding functions for various types, for instance, the Domain Names, the Questions and so on.

Moreover, this module will have some low-level byte string operation functions, like extract an integer from byte string or put an integer into byte string with necessary byte order conversion.

And this module will provide some helper functions to free various types if they are dynamically heap allocated.

Finally, this module will provide some helper functions to print the content of certain data types to stdout.

#### 3.2.2 Client Module

This module will contain a single main entry point for the whole logic process.

The whole logic process has the following operations in their execution order:

* Read in the server IP address and all the queries and store them into lists.
* Formulating the DNS query packet from the queries stored in lists.
* Create the socket and connect to the given local server.
* Send the DNS query packet to the local server.
* Receive the result packet from the local server and print it out.

#### 3.2.3 Server Module

This module will contain the main entry point and several core functions for the backbone logic process, and also a bunch of other utility functions that’s helpful for the backbone logic process.

The utility operations:

* Duplicate domain names.
* Get segments from string by splitting it with the specified delimiter.
* Read database file and extract useful records.
* Query and update the cache file.
* Push/Delete a query message into a list.
* Send a query packet to other server and wait for the reply.

The core operations:

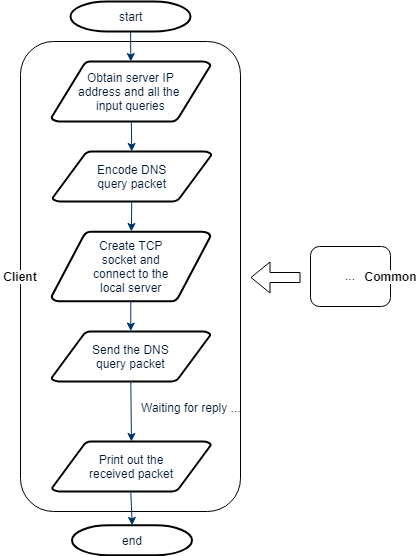
* Resolve domain name iteratively.
* Resolve domain name recursively.

The operations in the logic process in main entry point are listed below in their execution order:

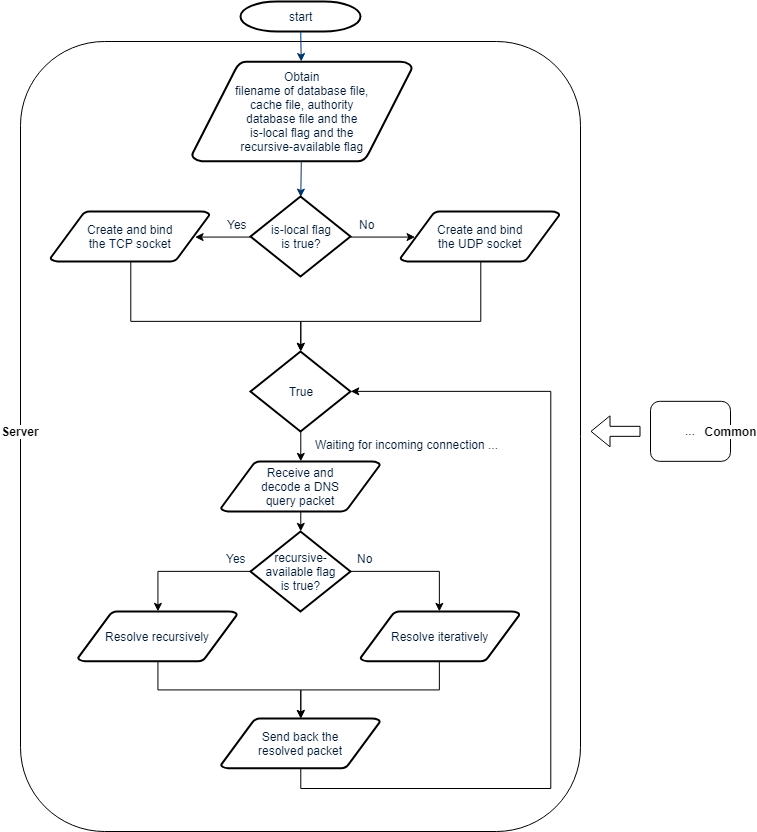
* Read in the filename of the database file, cache file, authority database file, and the is-local flag, recursive-available flag.
* Create and bind the TCP socket if the is-local flag is true, otherwise create the UDP socket.
* Enter an infinite loop to listen to the incoming packets. Every time receive a query packet,
  + Resolve it correctly in iterative fashion or recursively according to the recursive-available flag.
  + Send back the resolved packet to the requester.

## 3.2 Overall Flow Chart:

### 3.2.1 Client Module (interfacing with Common Module)



### 3.2.2 Server Module (interfacing with Common Module)



# 4. Detailed Design

## 4.1 Module Analysis

### 4.1.1 Common Module

In detail, this module will contain the following encoders/decoders, the relevant exported functions meant to be used by the Client and Server modules are listed as well.

* Encoder/decoder of Domain Names.
  + (exported) Encode Domain Name string into a byte string by removing the dots and mark the length of each segment in front of it.
  + (exported) Decode a byte string into a Domain Name type variable.
* Encoder/decoder of Questions.
* Encoder/decoder of Resource Records.
* Encoder/decoder of Message headers.
* Encoder/decoder of Message.
  + (exported) Encode the Message type variable into a packet.
  + (exported) Decode a packet into a Message type variable.

### 4.1.2 Client Module

The following list not only lists the operations in their execution order in main entry point but also lists the imported common functions from the Common module.

* Read in the server IP address and all the queries and store them into lists.
  + (imported) Use the encoder/decoder of the Domain Name.
* Formulating the DNS query packet from the queries stored in lists.
  + (imported) Use the encoder of the Message type.
* Create the socket and connect to the given local server.
* Send the DNS query packet to the local server.
* Receive the result packet from the local server and print it out.
  + (imported) Use the decoder of the Message type.

### 4.1.3 Server Module

The following list not only lists the operations in their execution order in main entry point but also lists the relevant imported functions from the Common module.

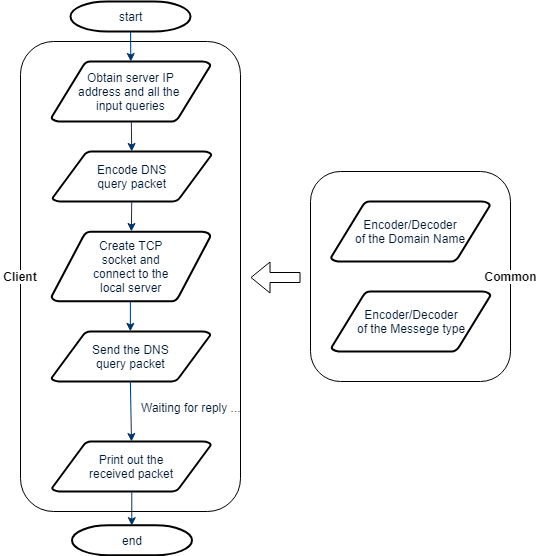
* Read in the filename of the database file, cache file, authority database file, and the is-local flag, recursive-available flag.
  + (imported) Use the encoder/decoder of the Domain Name
* Create and bind the TCP socket if the is-local flag is true, otherwise create the UDP socket.
* Enter an infinite loop to listen to the incoming packets. Every time receive a query packet,
  + Resolve it correctly in iterative fashion or recursively according to the recursive-available flag.
    - (imported) Use the decoder of the Message type.
  + Send back the resolved packet to the requester.
    - (imported) Use the encoder of the Message type.

The following list not only lists the operations in their execution order in core functions, but also lists the relevant imported functions from the Common module and the local utility functions.

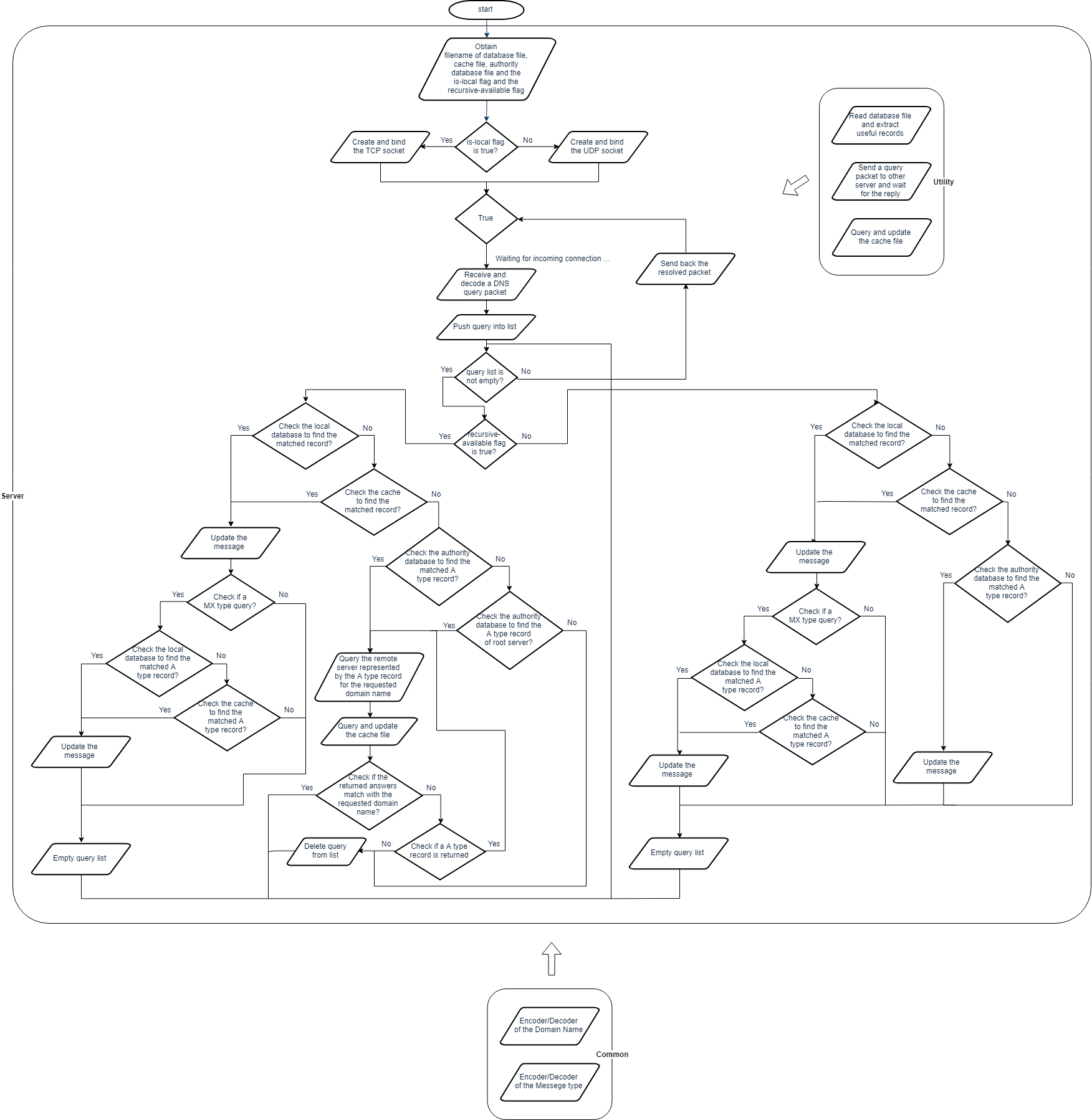
* Resolve domain name iteratively.
  + Check the local database to find the matched record.
    - (local) Read database file and extract useful records.
  + Check the cache file to find the matched record if nothing found in previous step.
    - (local) Read database file and extract useful records.
  + Update the Message accordingly if found the matched record.
  + Check the local database again for MX type query in order to find the corresponding A type record.
    - (imported) Decoder of Domain Name.
    - (local) Read database file and extract useful records.
  + Check the cache for the same purpose if nothing found in previous step.
    - (imported) Decoder of Domain Name.
    - (local) Read database file and extract useful records.
  + Turn to the authority database if still no matched record found.
    - (local) Read database file and extract useful records.
  + Update the Message accordingly if found the matched record.
* Resolve domain name recursively.
  + Check the local database to find the matched record.
    - (local) Read database file and extract useful records.
  + Check the cache file to find the matched record if nothing found in previous step.
    - (local) Read database file and extract useful records.
  + Update the Message accordingly if found the matched record.
  + Check the local database again for MX type query in order to find the corresponding A type record.
    - (imported) Decoder of Domain Name.
    - (local) Read database file and extract useful records.
  + Check the cache for the same purpose if nothing found in previous step.
    - (imported) Decoder of Domain Name.
    - (local) Read database file and extract useful records.
  + Check the authority database if still no matched record found.
    - (local) Read database file and extract useful records.
  + Get the root server A type record if still no matched record found and self is a local server.
    - (imported) Encoder of Domain Name.
    - (imported) Decoder of Domain Name.
    - (local) Read database file and extract useful records.
  + Go into a recursive process if an A type record is found.
  + Query the server presented by that A type record for the requested Domain Name.
    - (local) Send a query packet to other server and wait for the reply.
  + Look for the requested Domain Name in the cache file and update it if necessary.
    - (local) Query and update the cache file.
  + Exit the loop if a matched record is found.
  + Start a new pass if an A type record is returned instead to present another Domain server.

## 4.2 Module Flow Chart

### 4.2.1 Client Module (interfacing with Common Module)

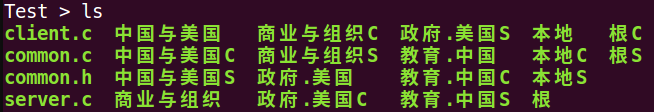


### 4.2.2 Server Module (interfacing with Common Module)



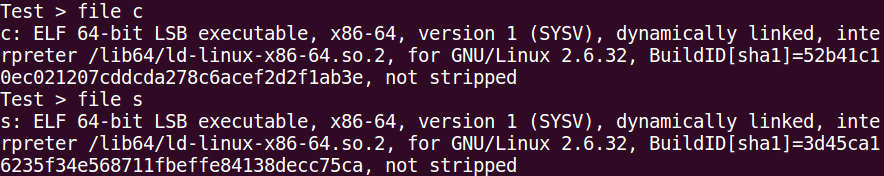
# 5. Results

The contents of the source folder:

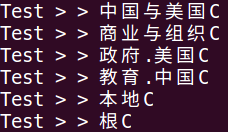


Compile the programs:





Empty all the cache files:



Start the local DNS server at 127.0.0.2 in one terminal:



Start the root DNS server at 127.0.0.3 in one terminal:



Start the other four DNS servers at 127.0.0.4, 127.0.0.5, 127.0.0.6, 127.0.0.7 respectively in separated terminals:

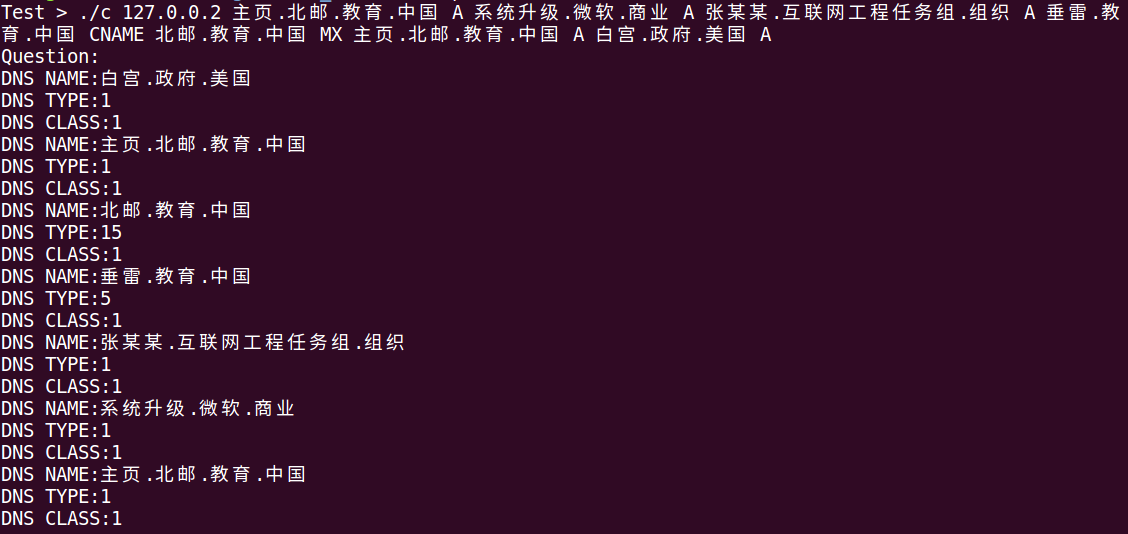


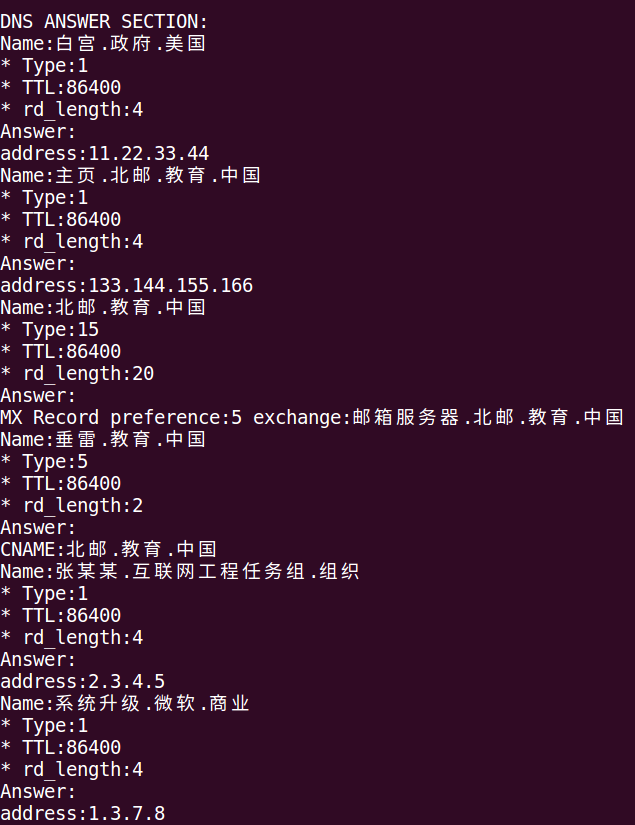


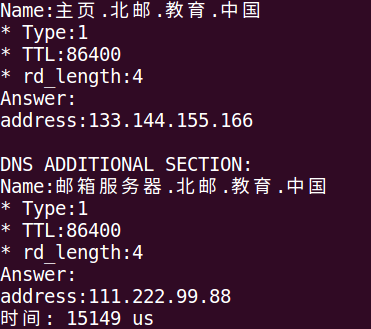




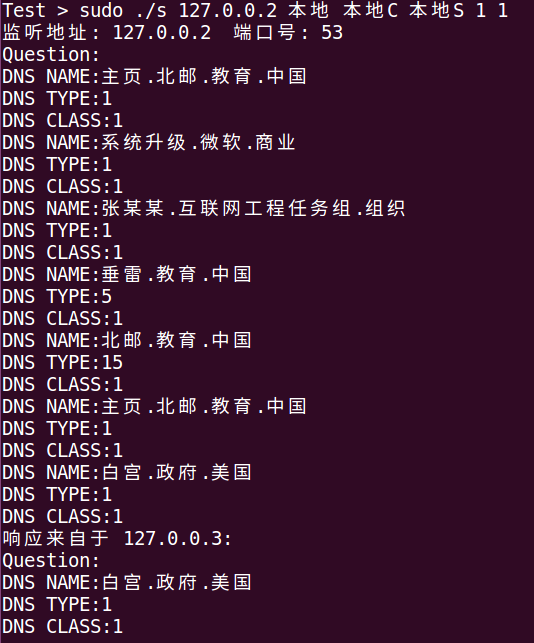
Run the DNS client in another terminal:



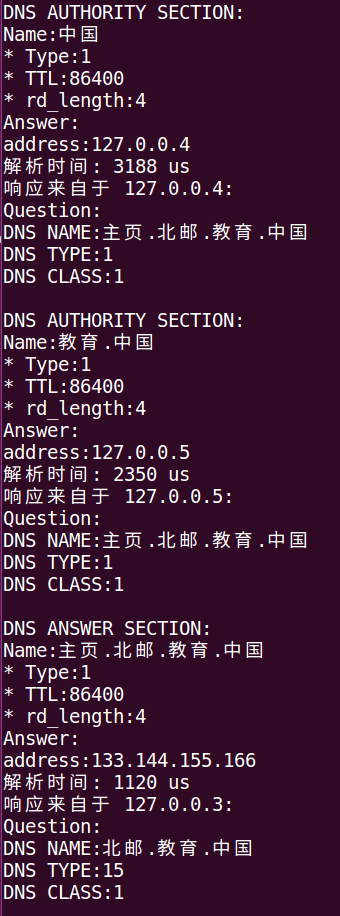


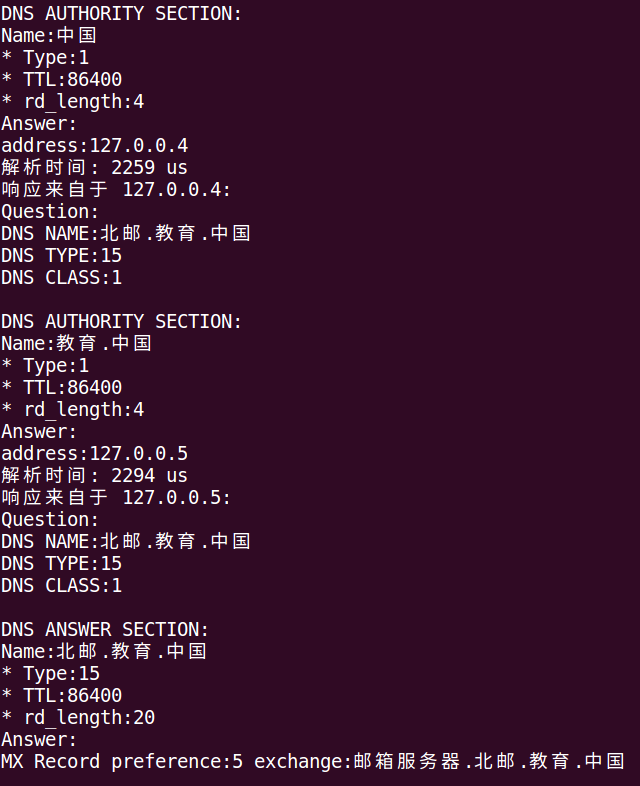


Check the output of the local DNS server:

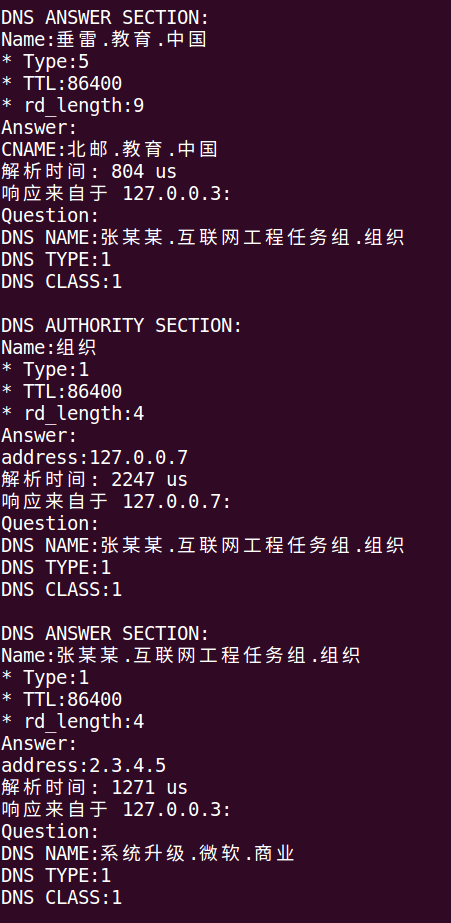


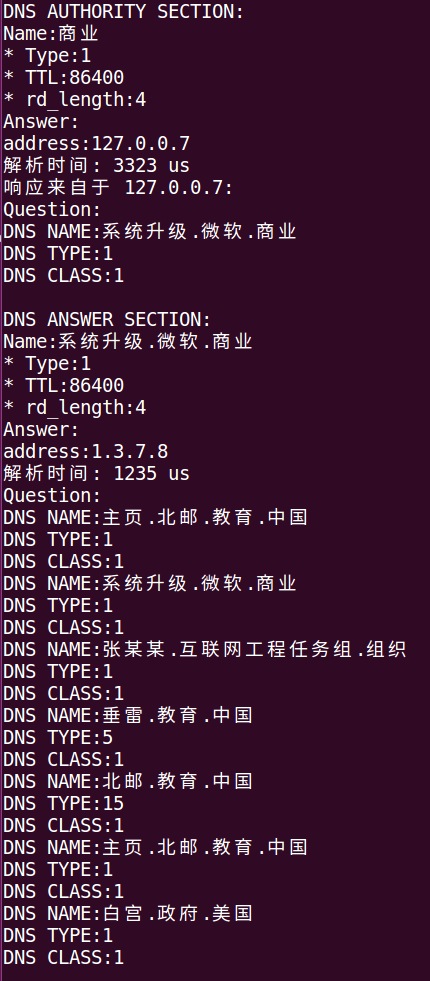


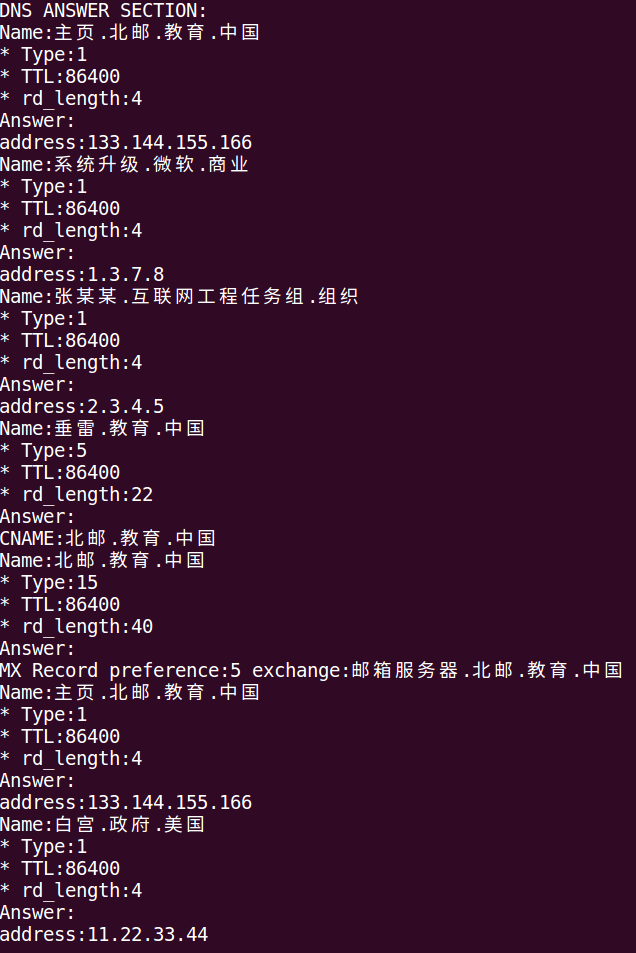


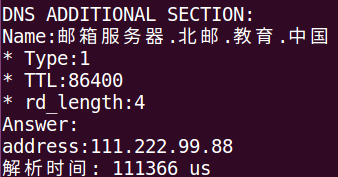




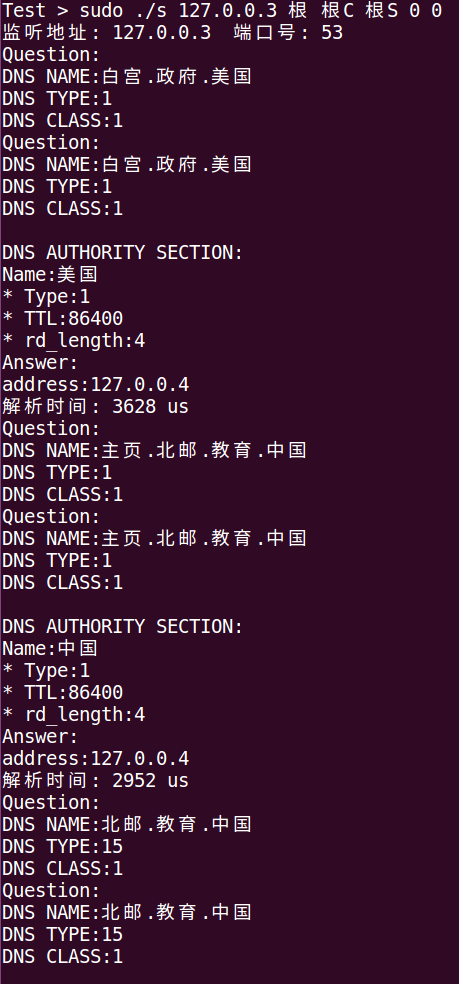


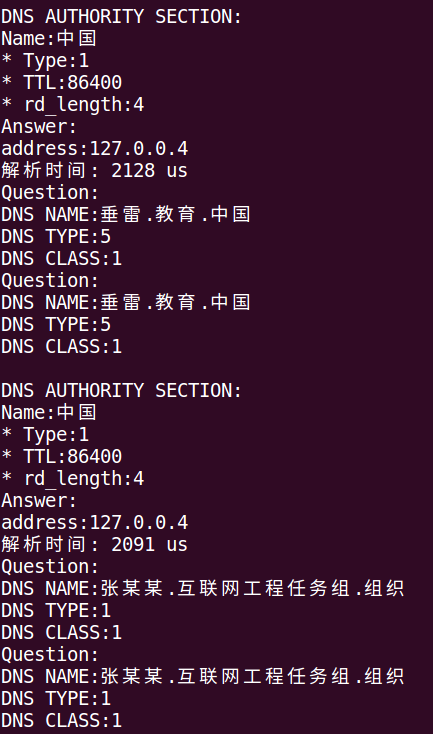


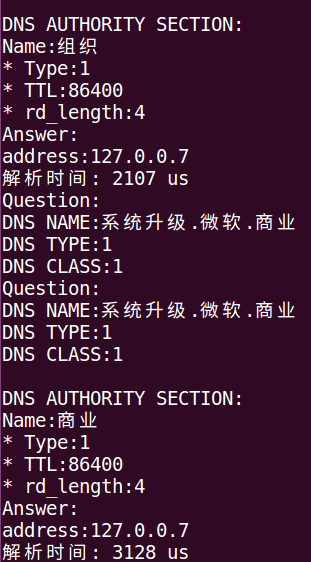




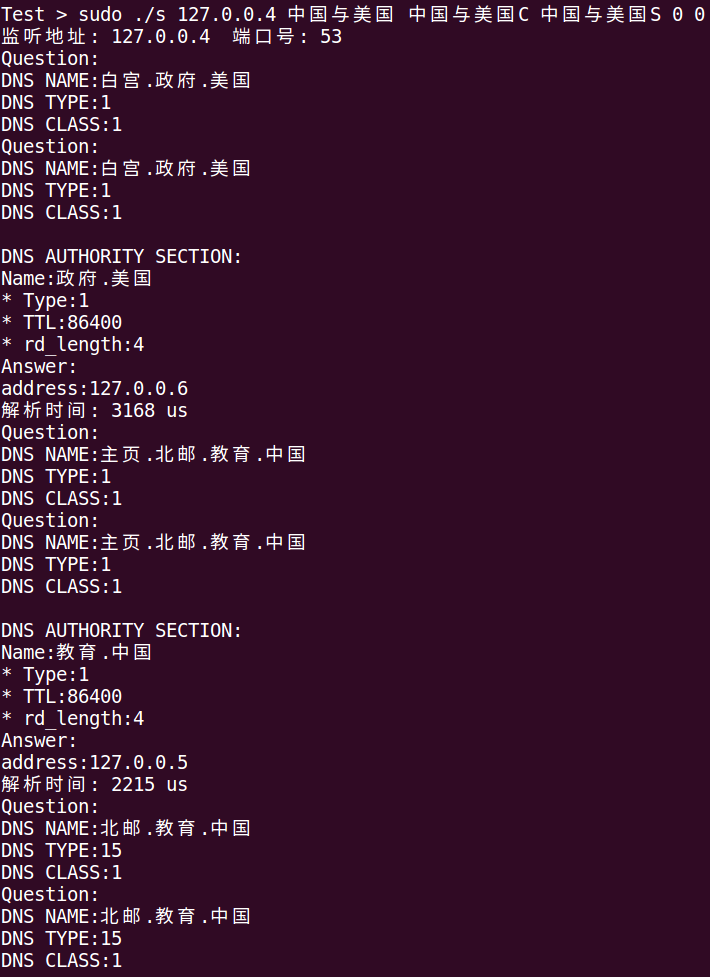
Check the output of the root DNS server:

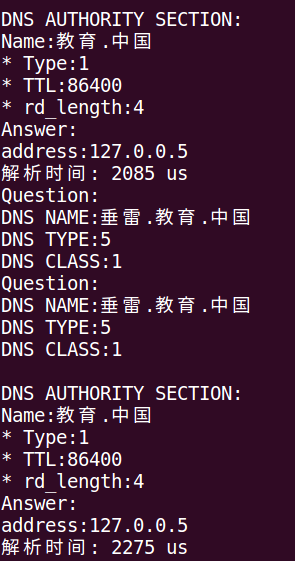




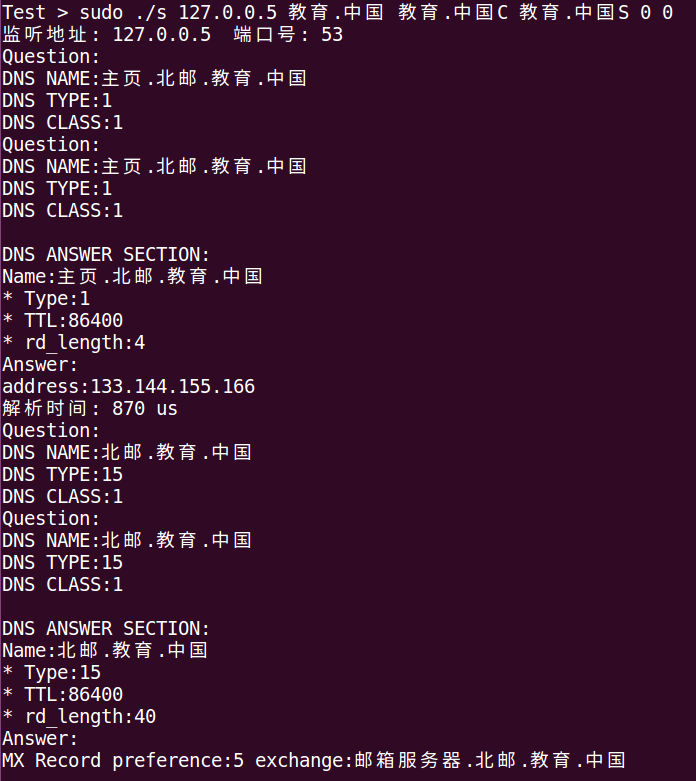


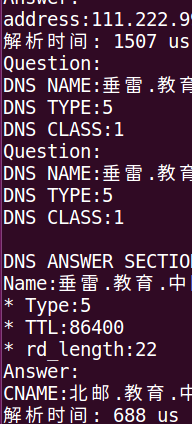
Check the output of the 中国与美国 DNS server:



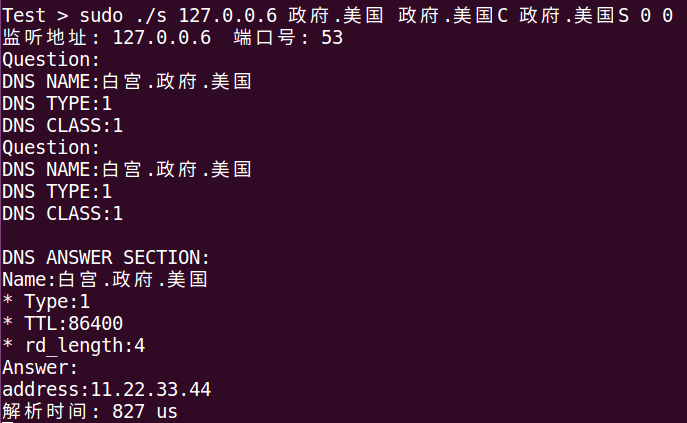


Check the output of the 教育.中国 DNS server:

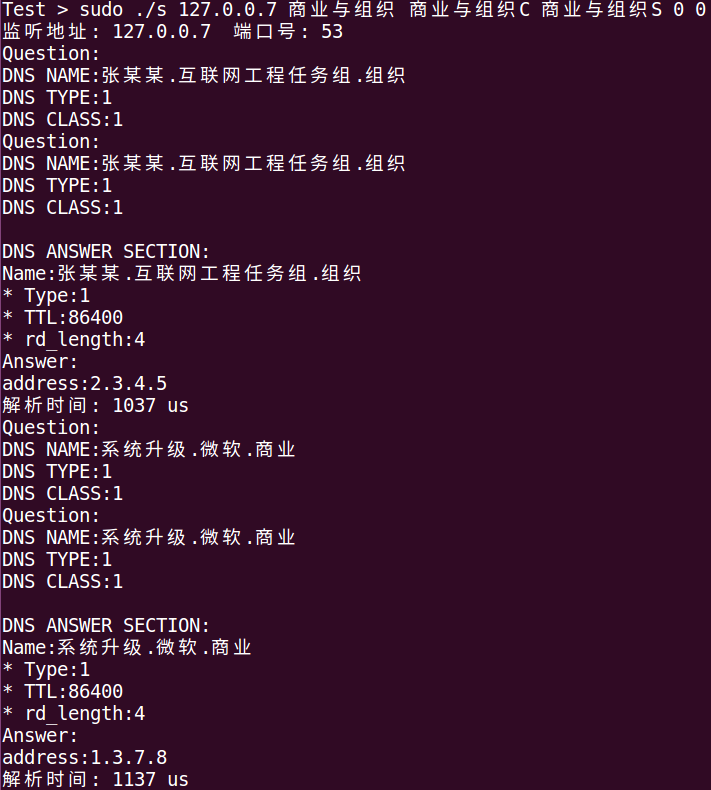




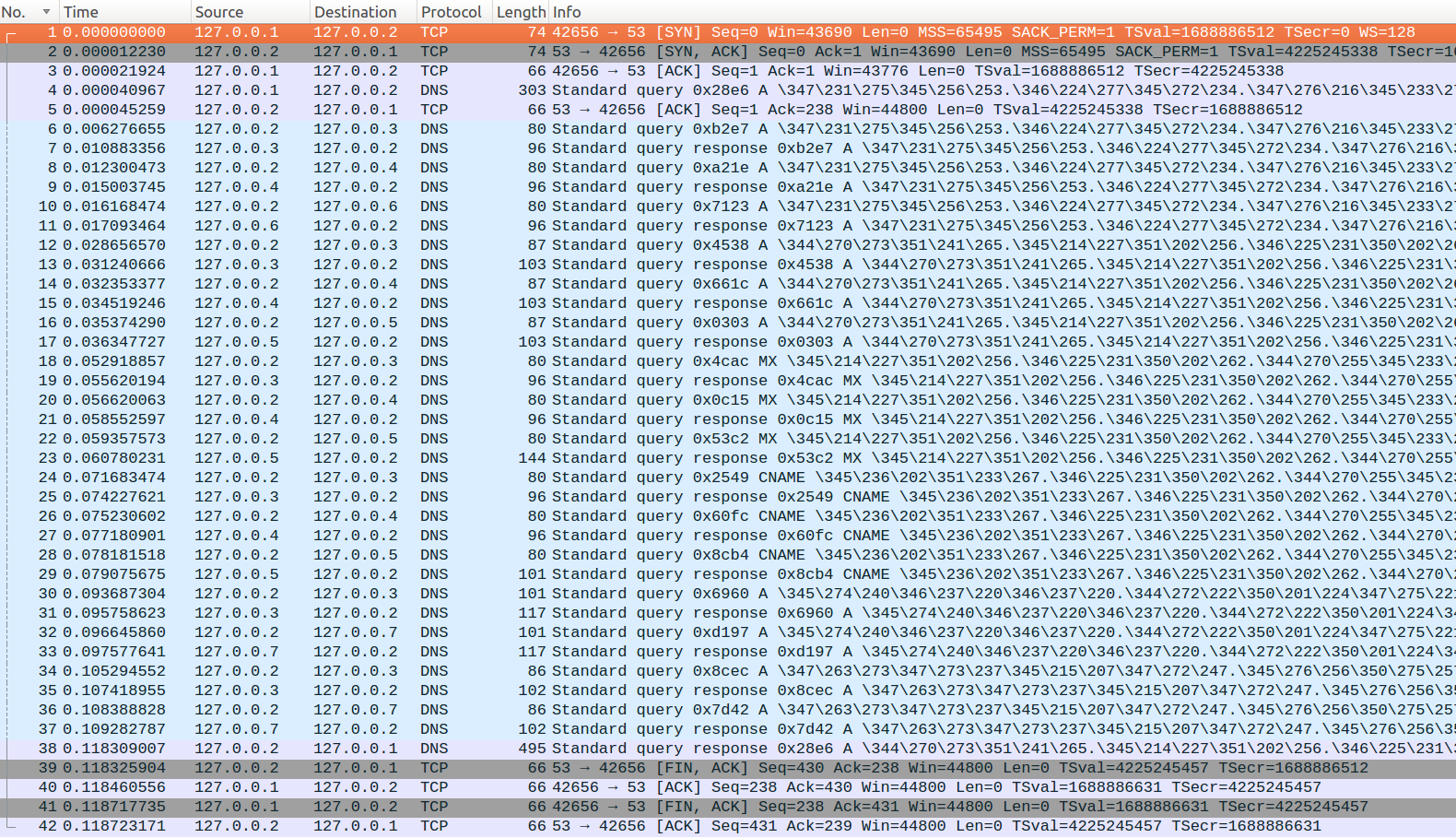
Check the output of the 政府.美国 DNS server:



Check the output of the 商业与组织 DNS server:



Check the packet capture history in Wireshark:



# 6. Summary and Conclusion

## 6.1 Work Assignment

The workload for this project is pretty heavy for even two students. Which means it has to be properly delegated as evenly as possible. We planned and carried through the following after negotiation:

* [S1] is responsible for design of the overall concept and build the basic program framework.
* [S2] then kicks in to implement the common module as well as the client module.
* After [S2] finish his work in the last phase, [S1] and [S2] start developing together the iterative and the recursive resolving mechanism for the server module.
* When the programs are finished but are flawed, [S1] and [S2] again are working alongside to debug the code and fix anything that appears incorrect.
* Finally, since the programs are fixed, tested and proved bug-free, [S1] start writing the chapter #1, #2, #5 of the report, meanwhile, [S2] start writing the chapter #3, #4 of the report, including drawing all the flow charts.
* The chapter #6 (this chapter) is left for [S1] and [S2] to work with each other to accomplish after they have done their partial work.

## 6.2 Self-Evaluation

### 6.2.1 [李雨晴]

My main effort is in the integrated design.

In order to design the overall concept, I have to rub up the theory I have acquired about the DNS protocol, conduct online research trying to clarify any points that are still unclear, and thanks to the provided material, which is a great help.

It’s really hard for me to build the basic program framework, as it’s a combination of the understanding of the theory and the skills of C language. I have definitely learned so much along the way.

During the implementing of resolving mechanism for the server, and the debugging of two programs, I have got the chance to dive into the details of the code and learn about the subtlety of the logic flow.

There are some points that could be improved in the future:

* The modules are heavy, which means they could be split into smaller ones for better code structure and also make it more readable, of course, the debug process will become easier too.
* The Resource Record of PTR type is not implemented yet, it could be implemented in the future to make the server program more complete.
* Because there’s no time, so the comments are missing for lots of code blocks, which could be carefully added in the future.

### 6.2.2 [李润源]

I focus on the detailed portions of this project.

When I thought I have known the core of the DNS protocol, it doesn’t mean that I could actually implement the code to realize it. Knowing the theory and putting it into practice is quite different. It forces me to go online and also from the given material to find the idea, related code snippet and the explanation of it.

It requires multiple times of refactoring to make a module look decent. At first, my code looks cumbersome, after the iterations, it becomes neater. It’s an awesome experience for me to go through this process of practicing and learning. I do enjoy it.

After I reviewed the implementation several times, I’d like to say there is still room for improvement:

* The socket programming part in this project is okay, but not great. Due to the testing is performed on a local machine, so we don’t need to consider the network traffic issue. However, in every real network program, it critical to consider the scenarios of failing to send a message (only part of it) and failing to receive a message (only a part of it). The next improvement could be making the socket programming part more reliable by adding the loops to check if a message has been sent/received completely.
* For every system call, we have to take care of its return value, no exceptions. So, the future improvement could be to catch the return value that is not captured yet and act accordingly.