DNS Relay Lab

Qingyu Wu

University of Science and Technology of China

2025-10-05

A. Introduction

Background

IPv4

- 第四版互联网协议，位于网络层
- 使用 32 位无符号整数标识互联网上的主机
- 常见的字符串表示形式
  - `202.38.64.59`：中科大网络通地址

IPv6

- 第六版互联网协议，位于网络层
- 使用 128 位无符号整数标识互联网上的主机
- 常见的字符串表示形式
  - `2409:8c00:6c21:1051:0:ff:b0af:279a`：一个北京用户的地址

Background

Socket

- 套接字，通常指的是运行在应用层和传输层之间的接口服务
- 基于 TCP/IP 的互联网中常用的是 TCP Socket 和 UDP Socket
  - TCP Socket 提供可靠字节流（reliable byte stream）传输服务
  - UDP Socket 提供不可靠数据包（unreliable datagram）传输服务

Domain Name System (DNS)

- 域名系统，一种分层分布式名称服务，运行在应用层
- 实现域名和 IP 地址之间的映射
- 基于 UDP 协议运行在主机 53 端口上

Background

hosts

- 主机文件，将域名（主机名）直接映射为 IP 地址的本地配置文件
  - Windows 下，位于 `C:\Windows\System32\drivers\etc\hosts`
  - Linux 下，位于 `/etc/hosts`
- 使用 # 表示行注释，一行一个映射条目，格式如下所示：

```
IP_address canonical_hostname [aliases...]
```

  - 其中 aliases 是同一个主机名的别名，说明一个映射条目可以同时表示多个主机名的映射

Background

hosts

- 主机文件，将域名（主机名）直接映射为 IP 地址的本地配置文件
  - Windows 下，位于 `C:\Windows\System32\drivers\etc\hosts`
  - Linux 下，位于 `/etc/hosts`
- Linux 下主机文件通常包含以下内容：

```
# Static table lookup for hostnames.
# See hosts(5) for details.
127.0.0.1       localhost
::1             localhost
```

即将主机名 `localhost` 绑定到回环地址 `127.0.0.1`（IPv4）和 `::1`（IPv6）

Background

mappings

为了不造成名称上的混淆，本实验中使用自定义的文件 mappings 表示本实验中使用的本地域名 IP 映射文件，从而区别于系统自带的 hosts 文件，规定如下：

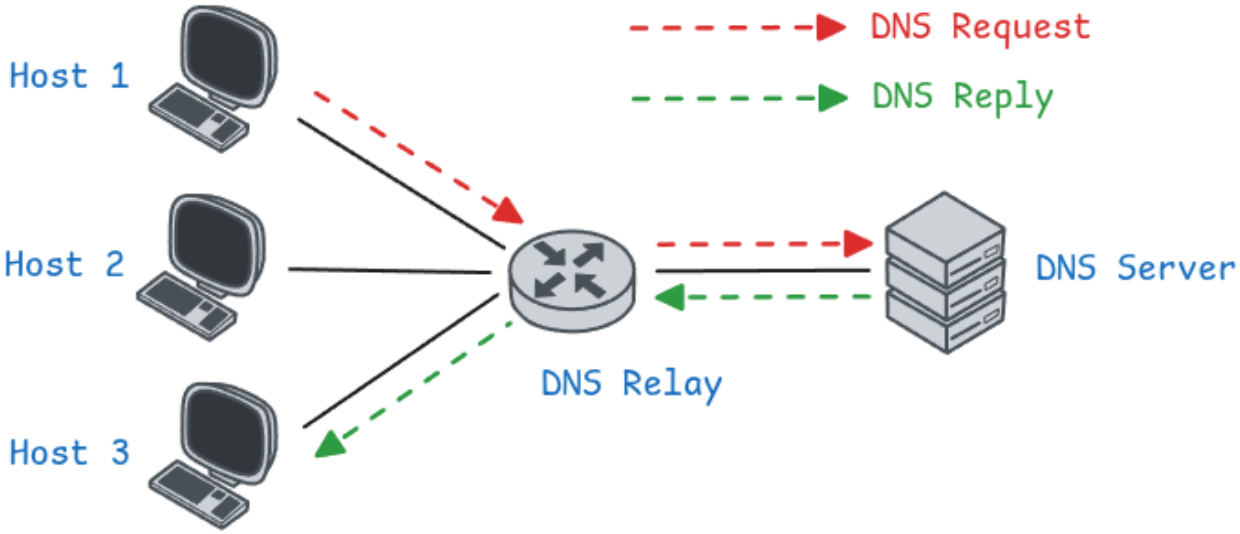
- mappings 文件的格式和 hosts 文件完全一样，只是文件名的不同
- mappings 文件存放在实验框架的根目录下，包含以下内容：

```
...
192.168.1.1             www.test1.com test1-v4 test
192.168.3.1             www.test1.com
2409::1                 www.test1.com test1-v6
...
```

Background

DNS Relay

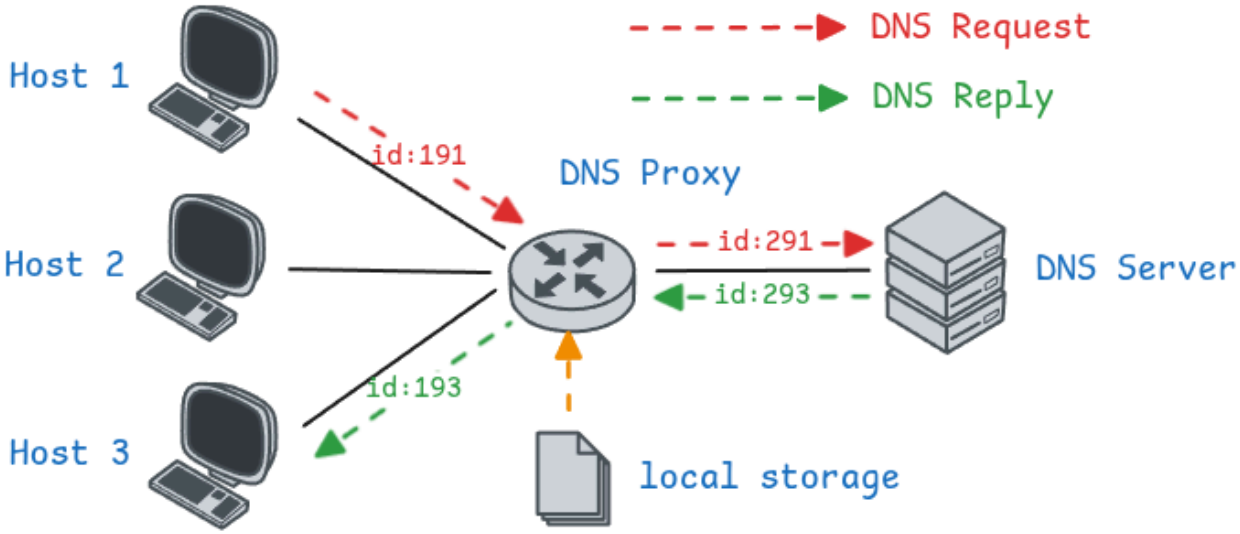
- DNS 中继器，一种位于主机和 DNS 服务器之间的中继服务
- 统一管理多个主机的上游 DNS 服务器，仅需修改 DNS 中继器即可
- 常用于路由器（搭配 DHCP 服务器返回网关 IP 作为 DNS 中继器）



Background

DNS Proxy

- DNS 代理，一种代理主机请求响应 DNS 服务器的代理服务
- 相比于 DNS 中继器，增加缓存、本地解析，修改 DNS 报文功能
- 常用于路由器，主机（绕过 DNS 污染攻击）



Background

Little Endian

- 小端序，即一个由多个字节构成的数的高位字节存放在内存地址的高位，低位字节存放在低位
- 目前大部分计算机使用的主机序（host byte order）都是小端序

Big Endian

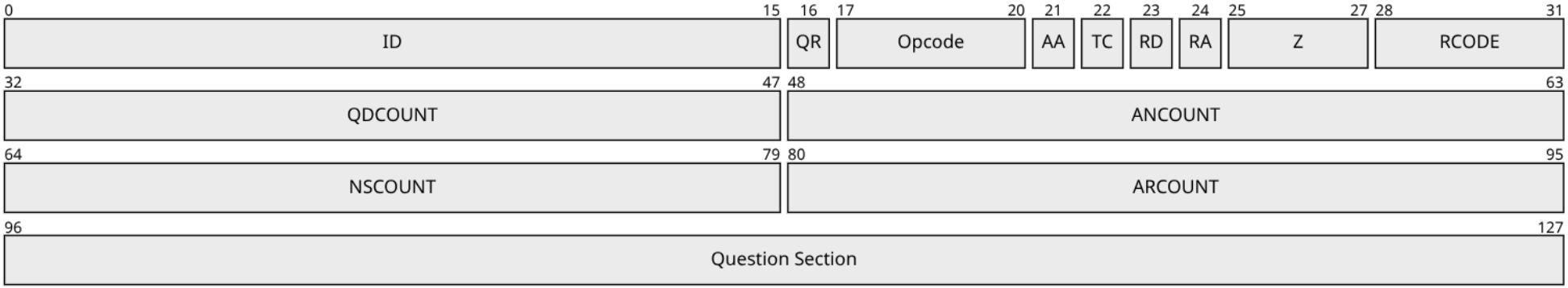
- 大端序，即一个由多个字节构成的数的高位字节存放在内存地址的低位，低位字节存放在高位
- 小部分计算机以及大部分网络传输协议（包括 DNS）使用的是都是大端序，Appendix 中介绍了一些 Linux 下方便的从主机序转换成网络大端序的库函数

Background

DNS datagram

DNS datagram 的格式由 RFC 1035[1] 规定。不定长字段会特别说明。

对于 DNS 请求（request）报文，其格式如下所示：



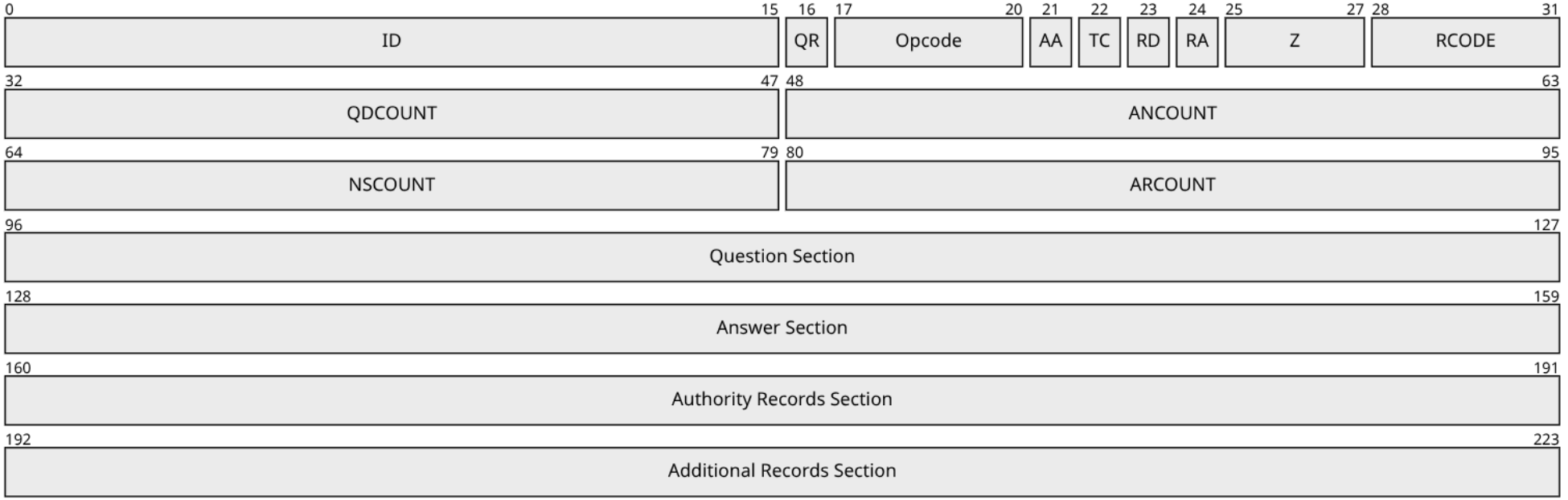
其中 Question Section 实际上是不定长的。

[1] https://www.rfc-editor.org/rfc/rfc1035

Background

DNS datagram

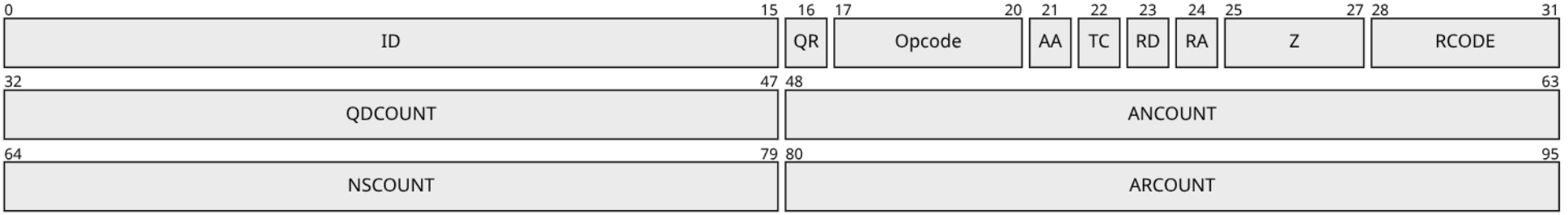
对于 DNS 响应（response）报文，其格式如下图所示：



其中除了 Header Section 其他 Section 实际上是不定长的。

Background

DNS Header Section

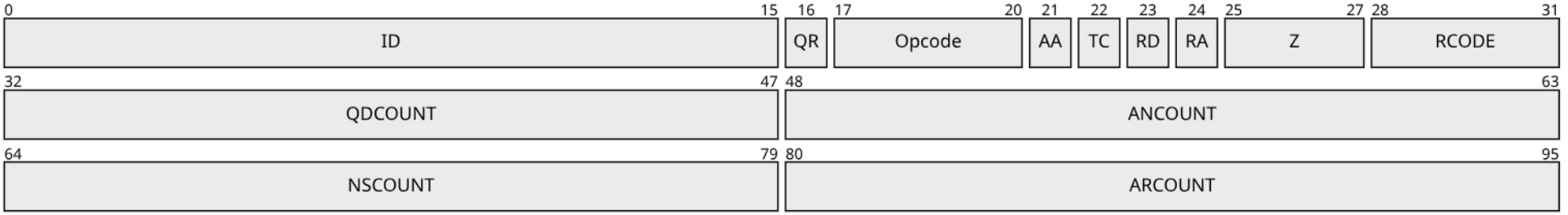


不论是 DNS 请求还是响应报文，除去后面的 Section 部分都称作 Header Section，其中各个字段的含义如下所示：

- ID：一个请求响应过程的唯一标识符
  - 由于 UDP 协议是面向无连接的，因此需要 ID 字段将发送的请求报文和收到的响应报文一一对应起来。

Background

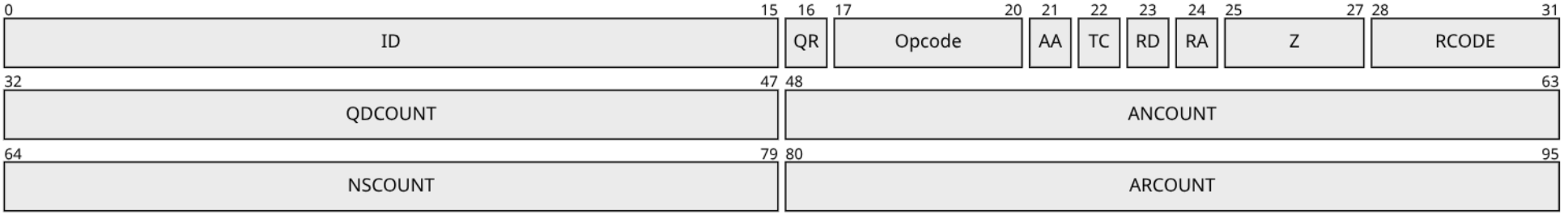
DNS Header Section



- QR：0 表示请求报文，1 表示响应报文
- OPCODE：0 表示标准查询，1 表示反向查询（IP 查域名），2 表示服务器状态查询，3-15 保留到以后使用
- AA：1 表示是权威服务器响应，0 表示不是
- TC：1 表示报文太大超过物理信道限制而被截断，0 表示没有

Background

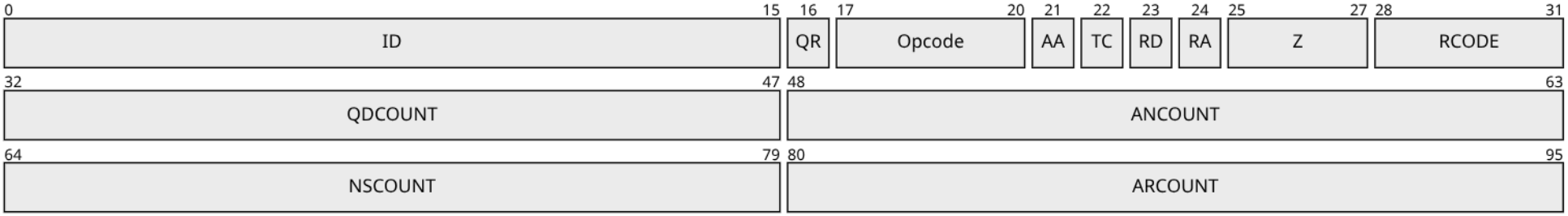
DNS Header Section



- RD：1 表示期望使用递归查询方式，0 表示不期望（只在请求报文中有效）
- RA：1 表示可以使用递归查询方式，0 表示不可以（只在响应报文中有效）
- Z：保留字段，不论请求响应必须为全 0

Background

DNS Header Section



- RCODE：0 表示没有错误，1 表示格式错误，2 表示服务器无法处理，3 表示找不到该域名（只有 AA = 1 时有效），等等
- QDCOUNT：Question Section 中包含的请求条目数
- ANCOUNT：Answer Section 中包含的资源记录数
- NSCOUNT：Authority Section 中包含的权威服务器数
- ARCOUNT：Additional Section 中包含的资源记录数

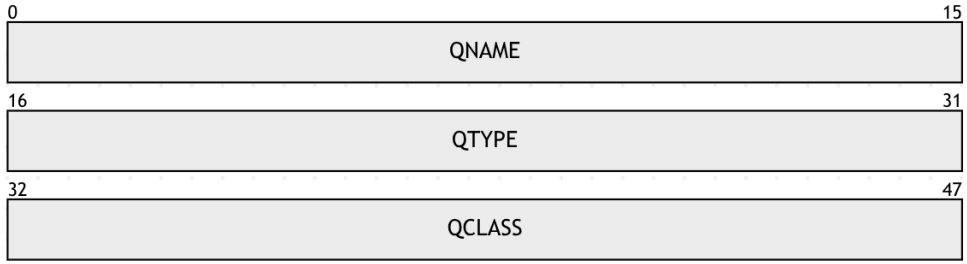
Background

DNS Question Section

Question Section 包含了所有 DNS 请求条目（entry）。

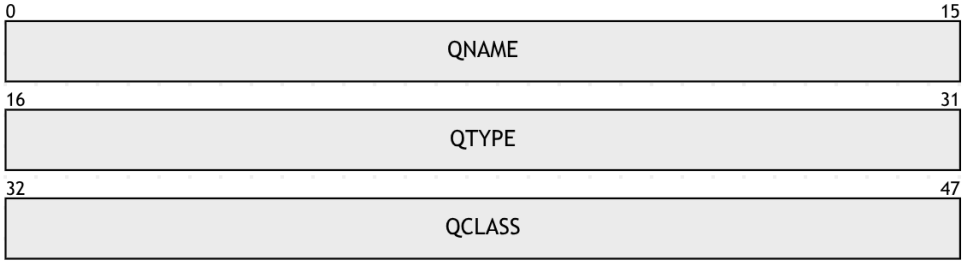
不论是 DNS 请求还是响应报文，都包含 Question Section，同一个 ID 下响应报文的 Question Section 保持和请求报文相同。

Question Section 由多个 entry 构成。由于域名长度是不固定的，每个 entry 的长度也是不固定的，格式如下：

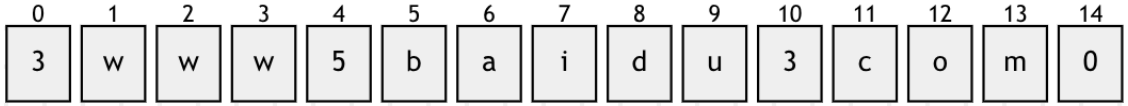


Background

DNS Question Section

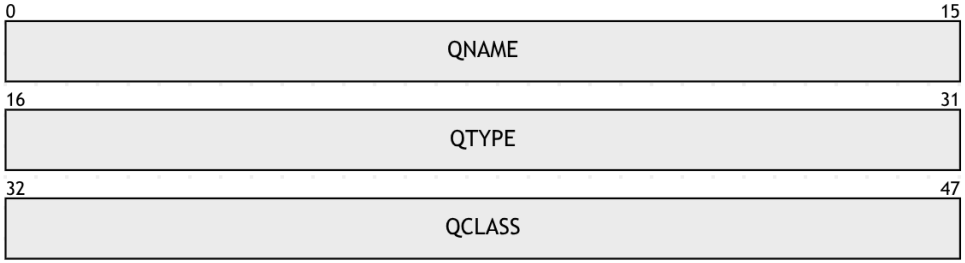


- QNAME：被查询的域名，不定长，被表示成一串 label 序列。每个 label 由一个字节表示的 length 以及一个 length 长的字符序列构成。label 序列以一个 `'\0'` 字节表示 0 长字符序列结束。
  - 比如，www.baidu.com 被表示成如下格式：



Background

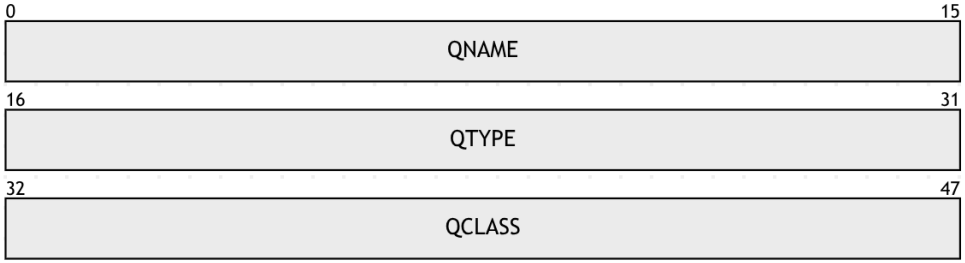
DNS Question Section



- QTYPE：被查询的资源记录（resource records）的类型（type），常用的 QTYPE 有：
  - 1 (A)：表示 IPv4 地址记录
  - 28 (AAAA)：表示 IPv6 地址记录
  - 5 (CNAME)：域名别名记录
  - 16 (TXT)：任意文本记录

Background

DNS Question Section



- QCLASS：被查询的资源记录的类别（class），常用的 QCLASS 就是 1 (IN)，表示互联网类别下的记录，其余的值可以是：
  - 2 (CS)：表示 CSNET 类别（已过时）
  - 3 (CH)：表示 CHAOS[1] 类别（基本被取代）
  - 4 (HS)：表示 Hesiod[2] 类别（基本被取代）

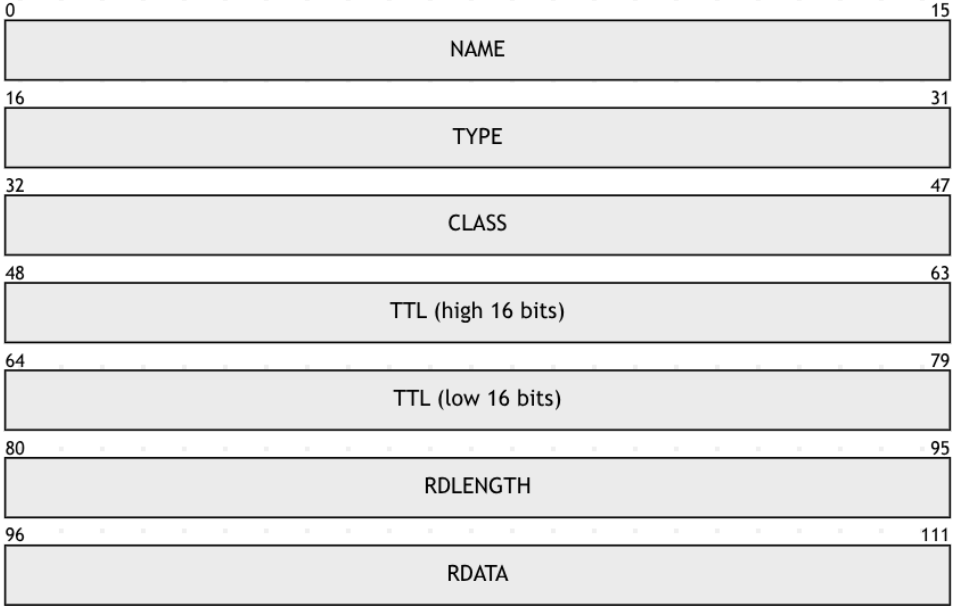
[1] https://chaosnet.net/
[2] https://simonwo.net/technical/hesiod/

Background

DNS Answer Section

Answer Section 包含了部分 DNS 资源记录，只会在响应报文中出现。

Answer Section 由多个资源记录构成。由于域名长度是不固定的，每个资源记录的长度也是不固定的，其格式如右图所示：



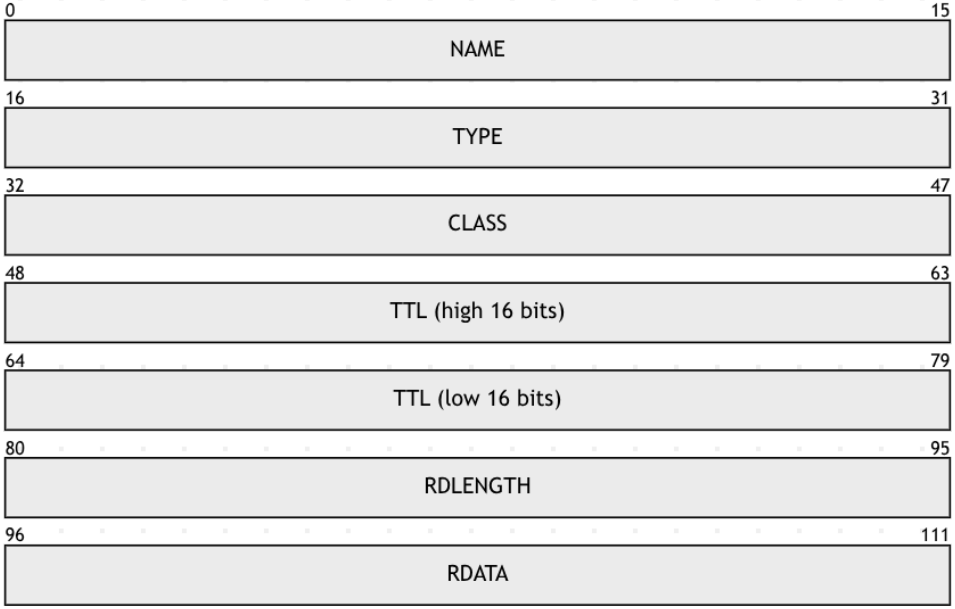
其中 NAME、TYPE、CLASS 字段的可选值和含义和 Question Section 中的基本相同

Background

DNS Answer Section

Answer Section 包含了部分 DNS 资源记录，只会在响应报文中出现。

- TTL：该资源记录应该被缓存的时间，单位为秒，0 表示不应该被缓存
- RDLENGTH：RDATA 字段的长度，单位为字节
- RDATA：该资源记录的内容，不定长，比如对于 A 记录，存放的就是 32 位 IPv4 地址



Background

DNS Authority Section

Authority Section 包含了权威服务器的资源记录，只会在响应报文中出现，和 Answer Section 的结构完全一样，通常存放 NS 类型的资源记录，用于表明被查询域名所属的权威服务器。本次实验不用。

DNS Additional Section

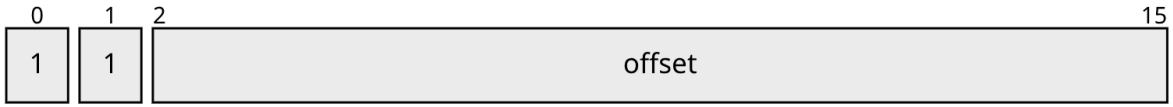
Additional Section 包含了额外提供的资源记录，只会在响应报文中出现，和 Answer Section 的结构完全一样，通常存放 A/AAAA 类型的资源记录，用于表明 Authority Section 里的权威服务器的 IP 地址，减少不必要的再次查询。本次实验不用。

Background

DNS Message Compression

为了缩短 DNS 报文大小，RFC 1035 还规定了一种减少重复表示域名的压缩方案（compression scheme）。

如果之前在**整个** DNS datagram 的 offset 字节处已经存放了一个由 label 序列构成的域名，那么在之后存放域名的字段中，可以直接存放一个 16 位的指针，其格式如下所示：



其中，开头两位仅用于区分当前字段存放的是指针还是 label 序列，对于 label 序列，这两位一定是 0，因为单个 label 不能超过 63 字节。

B. Overall

DNS Relay with local resolve

本次实验的实验框架是一个使用 C 语言通过 Linux Socket 库实现的带有本地解析功能的 DNS 中继器。

相比于 DNS Relay，我们还需额外实现通过读取 mappings 文件直接本地解析域名的功能，从而能手动控制域名解析的过程。

相比于 DNS Proxy，我们无需实现缓存功能，从而增加与这门课程无关的工作量（缓存策略算法，文件更新检测）

Prerequisites

System

由于本次实验的实验框架是使用 C 编写的，缺乏可移植性，因此实验框架只能跑在 Linux 上，可以使用 Windows Subsystem Linux (WSL)、VLab、虚拟机等形式完成此 Lab。

Build tools

本次实验使用 gcc 编译器和 GNU Make 构建工具，在 Ubuntu/Debian 下可以运行以下命令安装并查看是否正确安装：

```
sudo apt update && sudo apt install gcc make
gcc --version && make --version
```

Prerequisites

Test scripts

本次实验使用的 nslookup 工具可能在某些 Linux 发行版中默认没有安装，需要手动安装。

在 Ubuntu/Debian 下可以运行以下命令安装并查看是否正确安装：

```
sudo apt update
sudo apt install dnsutils
nslookup -version
```

Prerequisites

Test scripts

本次实验使用的 benchmark 脚本使用 Python 编写，调用 dnspython 库，因此还需要安装相关依赖库。

在 Ubuntu/Debian 下可以运行以下命令安装并查看是否正确安装：

```
sudo apt update
sudo apt install python-is-python3 python3-dnspython
python -c "import dns.resolver"
```

如果你只想在 venv 或 conda 中安装，运行以下命令：

```
pip install dnspython
python -c "import dns.resolver"
```

Build

Makefile

本次实验的实验框架使用 Makefile 组织，方便统一编译运行过程，使用说明如下所示：

- 构建框架：`make`
- 运行框架：`make run`
- 清理中间文件：`make clean`

C Flags

本次实验的实验框架使用以下编译选项：

```
-O2 -Wall -Wextra -Werror
```

这些编译选项会尽可能找到编写的代码中潜在的问题（比如未使用的变量，易误导的代码格式）并以 error 的形式终止编译，因此请尽可能保证代码规范整洁。

Test

nslookup

nslookup 是一个用于查询 DNS 服务器的命令行工具，使用方法：

```
nslookup [options] <domain_name> [dns_server]
```

比如，当在一个终端中使用 `make run` 运行起来框架后，在另一个终端中可以使用以下命令测试中继功能：

```
nslookup www.baidu.com 127.0.0.1
```

测试本地解析功能：

```
nslookup www.test1.com 127.0.0.1
```

Test

nslookup

以下是运行 nslookup 后的示例输出：

```
~/Workspace/C/dns-relay master !1 ?3
❯ nslookup www.baidu.com 127.0.0.1
Server:         127.0.0.1
Address:        127.0.0.1#53

Non-authoritative answer:
Name:   www.baidu.com
Address: 110.242.70.57
Name:   www.baidu.com
Address: 110.242.69.21
Name:   www.baidu.com
Address: 2408:871a:2100:186c:0:ff:b07e:3fbc
Name:   www.baidu.com
Address: 2408:871a:2100:1b23:0:ff:b07a:7ebc
```

```
~/Workspace/C/dns-relay master !1 ?3
❯ nslookup www.test1.com 127.0.0.1
Server:         127.0.0.1
Address:        127.0.0.1#53

Non-authoritative answer:
Name:   www.test1.com
Address: 192.168.1.1
Name:   www.test1.com
Address: 192.168.3.1
Name:   www.test1.com
Address: 2409::1
Name:   www.test1.com
Address: 2409:fe02::1
```

Benchmark

除了使用 nslookup 手动查询，我们还提供了 benchmark 脚本，分别用于测试本地解析和中继的性能。（记得在其他终端先跑起来框架再运行脚本）

测试中继功能：

```
./benchmark_remote.sh
```

测试本地解析功能：

```
./benchmark_local.sh
```

Benchmark

以下是运行 benchmark 脚本后的示例输出：

```
~/Workspace/C/dns-relay master !1 ?3
❯ ./benchmark_local.sh

Starting DNS benchmark...
Target DNS Server: 127.0.0.1:53
Number of queries: 10000
Concurrency level: 20
Number of unique domains: 16


Benchmark Results:
================================================
Total Queries:      10000
Successful:         10000
Failed:             0
Success Rate:       100.00%

Timing Statistics (ms):
Average:            8.09
Median:             5.90
Min:                0.14
Max:                105.15
Std Dev:            7.88
```

```
~/Workspace/C/dns-relay master !1 ?3
❯ ./benchmark_remote.sh

Starting DNS benchmark...
Target DNS Server: 127.0.0.1:53
Number of queries: 2000
Concurrency level: 20
Number of unique domains: 8


Benchmark Results:
================================================
Total Queries:      2000
Successful:         2000
Failed:             0
Success Rate:       100.00%

Timing Statistics (ms):
Average:            281.07
Median:             28.36
Min:                27.53
Max:                4333.82
Std Dev:            690.34
```

Browser

Linux 的 DNS 配置文件位于 /etc/resolv.conf，通过修改配置文件，添加以下行，可以将默认 DNS 服务器定位到本地的 DNS 中继器：

```
nameserver 127.0.0.1
```

现在，你可以通过修改 mappings 文件控制这台电脑的 DNS 解析过程了。比如，如果你将 `www.baidu.com` 解析成 `127.0.0.1`，那么打开 firefox 后访问 `www.baidu.com` 会发现打不开网页；再比如，如果你将页面上广告资源的 URL 中的域名解析成 `0.0.0.0` 或者 `127.0.0.1`，那么页面上的广告就会被屏蔽。

Evaluation

Correctness (25%)

正确性检验由 nslookup 调用和 benchmark 脚本组成，要求实验完成后都能正确运行。运行结果放在实验报告的截图里。

Report (25%)

实验完成后，需要提交一份实验报告，要求如下：

- 解释代码结构和作用，讲述自己实现的补全部分的逻辑
- 截图展示实验结果，包含 nslookup 和 benchmark 结果

Evaluation

Demonstration (50%)

实验完成后，实验检查会提供新的配置文件，现场使用新的 mappings 文件和新的 benchmark 脚本测试，尽可能包含多的 corner cases。

同时，我们还会提供一份 mappings 文件用于过滤一个网页的广告，现场演示通过 DNS 中继器实现广告过滤。

Submission

- 提交地点：bb.ustc.edu.cn 的“作业区”
- 提交文件：一个 zip 格式的压缩包
  - 命名格式：学号-姓名-DNS_Relay.zip
  - 包含文件：
    - 一个 pdf 格式的实验文档
      - 命名格式：学号-姓名-DNS_Relay.pdf
    - 两个包含 TODO 部分的 c 格式的源代码文件
      - `dns_dgram_utils.c`
      - `dns_relay.c`

Timeline

- 9.23 实验发布
  - 文档和代码发布在 bb.ustc.edu.cn 的“作业区”
  - 代码也支持直接用 git clone 下来：

    ```
    git clone https://github.com/Vertsineu/dns-relay-lab.git
    ```

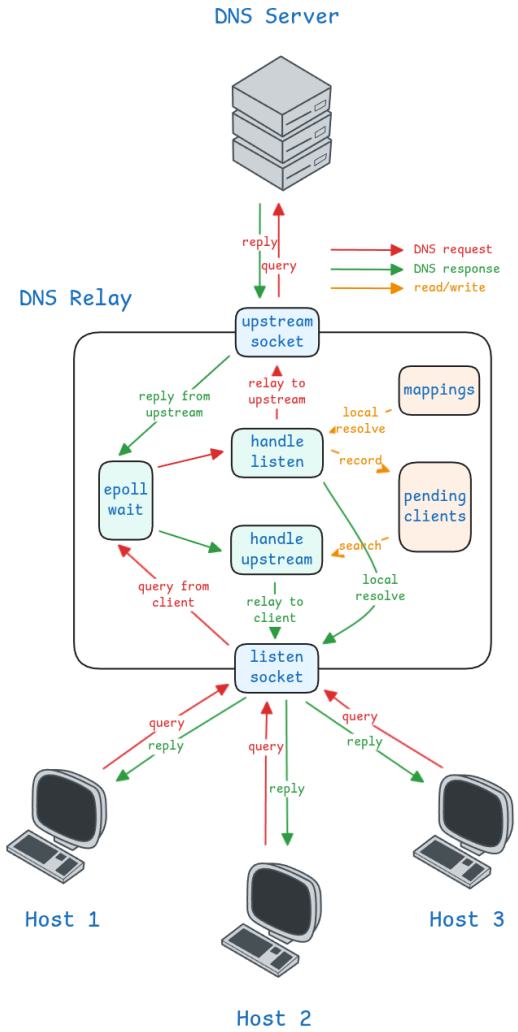
    ~~之前开发实验框架的 commits 已经被删除了~~
- 9.25 讲解实验内容
- 11.30 实验提交截止日期
- 12 月 检查实验，时间待定

C. Implementation

Architecture

接下来我们将讲述本实验的实验框架的具体架构：

- listen socket
  - 发送和接收客户端（主机）的 datagram
- upstream socket
  - 发送和接收上游 DNS 的 datagram
- pending clients
  - 记录由于无法 local resolve 而阻塞的客户端
- mappings
  - 用于本地解析的 mappings 文件

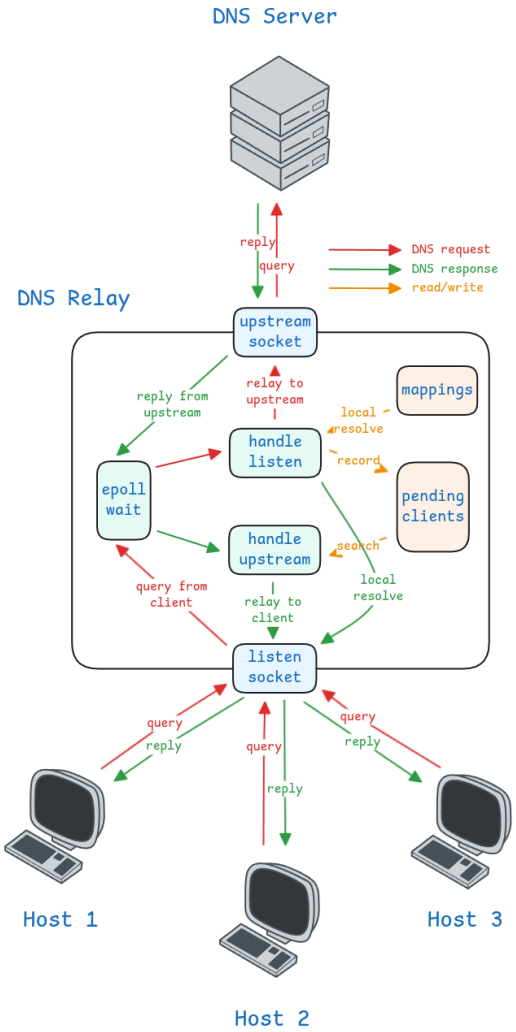


Architecture

- epoll wait
  - 等待并接收来自多个 socket 的 datagram
- handle listen
  - 处理来自 listen socket 的 datagram
- handle upstream
  - 处理来自 upstream socket 的 datagram

每一个合法的域名解析过程从 DNS request 开始，到 DNS response 结束。

一共有本地解析和中继两种合法域名解析过程。



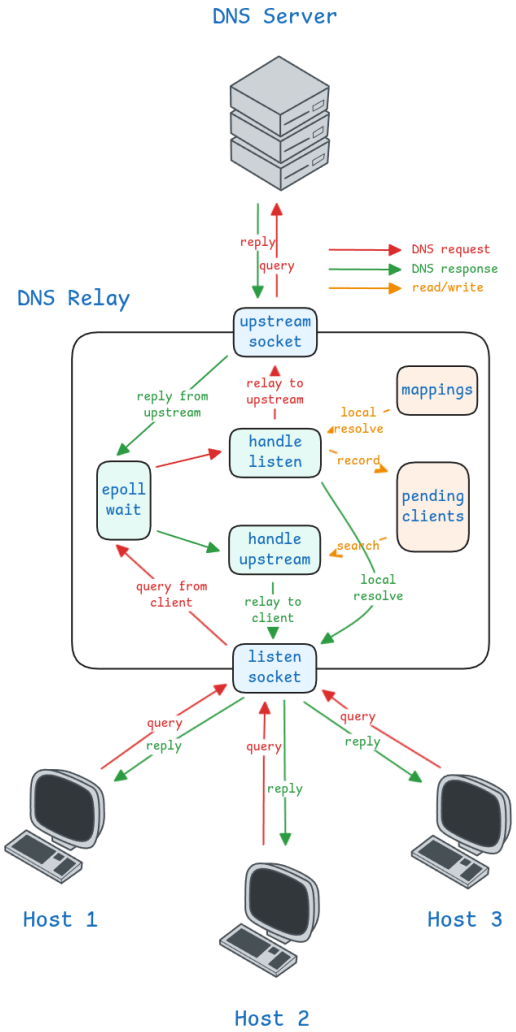
Architecture

local resolve

从 Host 开始，通过 listen socket 被 epoll wait 接收，传递给 handle listen，读取 mappings 文件，通过 listen socket 响应回 Host。

relay

从 Host 开始，通过 listen socket 被 epoll wait 接收，传递给 handle listen，通过 upstream socket 传递给 DNS Server，通过 upstream socket 被 epoll wait 接收，传递给 handle upstream，通过 listen socket 响应回 Host。



File Structure

除了测试文件、构建文件和说明文件，
实验框架主要由以下源文件构成：

- `dns_relay.c` 和 `dns_relay.h`
  - 实现 socket 的接收和发送
  - 实现 datagram 的处理逻辑
- `dns_dgram_utils.c` 和 `dns_dgram_utils.h`
  - 实现 datagram 的解析和修改
  - 实现本地解析
- `dns_relay_utils.h`
  - 一些方便的辅助/封装函数

```
.
├── benchmark_local.sh
├── benchmark_remote.sh
├── dns_benchmark.py
├── dns_dgram_utils.c
├── dns_dgram_utils.h
├── dns_relay.c
├── dns_relay.h
├── dns_relay_utils.h
├── mappings
├── Makefile
└── README.md
```

File Structure

dns_relay.h 中有一些全局配置的宏：

```
  // configs
  #define LISTEN_ADDR "127.0.0.1"
  #define LISTEN_PORT 53
  #define UPSTREAM_ADDR
"114.114.114.114"
  #define UPSTREAM_PORT 53
```

configs 里定义了 listen socket 绑定地址和端口和上游 DNS 服务器地址和端口

```
.
├── benchmark_local.sh
├── benchmark_remote.sh
├── dns_benchmark.py
├── dns_dgram_utils.c
├── dns_dgram_utils.h
├── dns_relay.c
├── dns_relay.h
├── dns_relay_utils.h
├── mappings
├── Makefile
└── README.md
```

File Structure

```c
// policies
#define MAX_EVENTS 128
#define MAX_ANSWER_COUNT 8
```

policies 里定义了最大并发接收 datagram 数，最大响应资源记录数

```c
// host file
#define HOST_FILE_PATH "./mappings"
```

host file 里定义了本地解析所使用的 mappings 文件路径

```
.
├── benchmark_local.sh
├── benchmark_remote.sh
├── dns_benchmark.py
├── dns_dgram_utils.c
├── dns_dgram_utils.h
├── dns_relay.c
├── dns_relay.h
├── dns_relay_utils.h
├── mappings
├── Makefile
└── README.md
```

File Structure

`dns_relay_utils.h` 中定义了两个完成实验过程中需要调用的函数：

```
inline void send_datagram(int fd,
unsigned char *buf, int len,
client_info_t client_info)
```

将 `buf` 中 `len` 字节 datagram 通过 `fd` 的 socket 发送给 `client_info` 指定的 client

```
inline void log_result(const char
*type, const char *name)
```

将域名 `name` 的解析过程 `type` 打印出来

```
.
├── benchmark_local.sh
├── benchmark_remote.sh
├── dns_benchmark.py
├── dns_dgram_utils.c
├── dns_dgram_utils.h
├── dns_relay.c
├── dns_relay.h
├── dns_relay_utils.h
├── mappings
├── Makefile
└── README.md
```

File Structure

`dns_dgram_utils.h` 中定义了 DNS datagram 的各种结构，并使用 gcc 拓展语法 `__attribute__((packed))` 保证 struct 没有 padding，从而方便实验过程中直接将原始二进制报文通过一个简单的类型转换转换成一个指针，比如：

```
dns_header_t *dns_header =
(dns_header_t *) buf;
```

注意 `dns_question_t` 的实现并不完全对应，可以阅读注释查看原因。

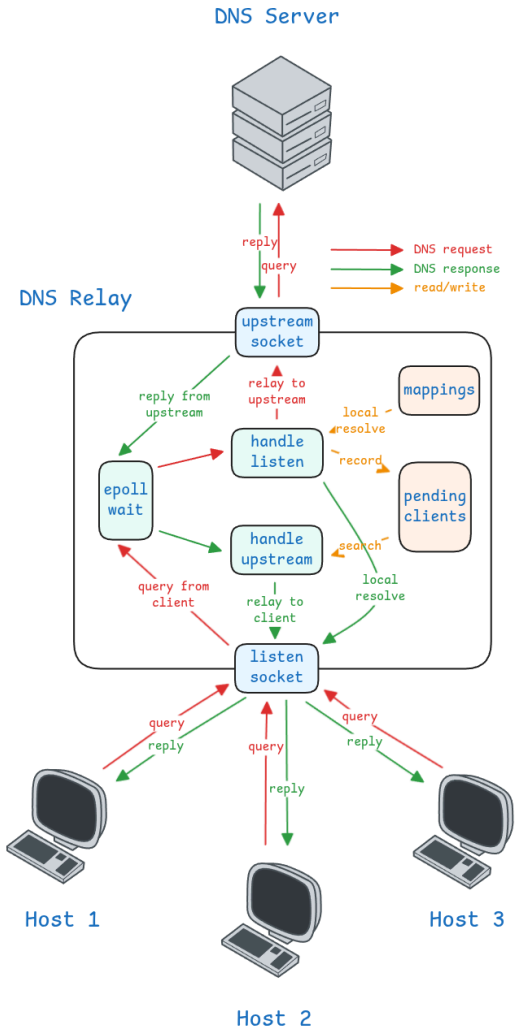
```
.
├── benchmark_local.sh
├── benchmark_remote.sh
├── dns_benchmark.py
├── dns_dgram_utils.c
├── dns_dgram_utils.h
├── dns_relay.c
├── dns_relay.h
├── dns_relay_utils.h
├── mappings
├── Makefile
└── README.md
```

TODO

给出的实验框架已经实现了除了 handle listen 以外的全部部分，而你需要实现的内容有：

- `dns_dgram_utils.c` 里的所有函数
  - 实现并调用这些函数可以大大简化 handle listen 部分的实现
- `dns_relay.c` 里的 handle listen 部分
  - 调用 `dns_dgram_utils.c` 和 `dns_relay_utils.c` 里的函数来实现该部分

你可以参考右侧的示意图来了解和完成 handle listen 的工作原理



D. Appendix

Introduction

附录中收录了本实验框架在实现完成后**所有**调用的 Linux 系统提供的**与网络相关**的库函数，其中:

- Linux Internet 部分的函数是你在实现 TODO 过程中**会直接**使用的函数
- Linux Socket 部分的函数是你在实现 TODO 过程中**不会直接**使用的函数，仅仅是为了方便理解实验框架而写上来的。

至于其他函数，比如标准库函数，框架里提供的函数都是可以使用但不一定会使用的函数，实现 TODO 时自行斟酌即可。

inet_pton()

```
int inet_pton (int __af, const char *__restrict __cp, void
*__restrict __buf)
```

将存放在 __cp 中的字符串形式的地址转换成二进制形式的地址存放在 __buf 中。如果转换成功，返回 1；如果转换失败，返回 0。

比如对于 IPv4，将 "127.0.0.1" 转换成 0x7f000001，并且保证是互联网协议要求的大端序表示。

在本次实验中，__af 为 AF_INET 或 AF_INET6，表示 IPv4 或 IPv6。

inet_ntop()

```
const char *inet_ntop (int __af, const void *__restrict __cp,
char *__restrict __buf, socklen_t __len)
```

将存放在 __buf 中的二进制形式的地址转换成字符串形式存放在 __cp 中。如果转换成功，返回 1；如果转换失败，返回 0。

比如对于 IPv6，将 0xfe80000000000000000000000000001 转换成 "fe80::1"，并且保证是互联网协议要求的大端序表示。

在本次实验中，__af 为 AF_INET 或 AF_INET6，表示 IPv4 或 IPv6。

ntohl()

```
uint32_t ntohl (uint32_t __netlong)
```

ntohs()

```
uint16_t ntohs (uint16_t __netshort)
```

htonl()

```
uint32_t htonl (uint32_t __hostlong)
```

htons()

```
uint16_t htons (uint16_t __hostshort)
```

以上是对于 16 位和 32 位数在主机序和网络大端序中转换的函数

socket()

```
int socket(int __domain, int __type, int __protocol)
```

在 Linux 下，一切皆文件，因此 socket 也可以看成是一个特殊的文件。在本实验中，我们可以使用以下方式创建一个 UDP socket，用来接收和发送 DNS datagram。

```
int fd = socket(PF_INET, SOCK_DGRAM, 0)
```

其中，返回值 fd 表示 file descriptor，相当于一种标识一个被打开的文件的变量，从而用于其他函数调用，负值表示创建失败。

sendto()

```
ssize_t sendto (int __fd, const void *__buf, size_t __n, int
__flags, __CONST_SOCKADDR_ARG __addr, socklen_t __addr_len)
```

顾名思义，将 __buf 中的 __n 字节内容通过 __fd 中对应的 socket 发送到 __addr 和 __addr_len 表示的 IP 地址和端口对应的客户端。

在本实验实验框架中，该函数被简单包装成一个 inline 函数，以方便调用：

```
inline void send_datagram(int fd, unsigned char *buf, int
len, client_info_t client_info)
```

bind()

```
int bind (int __fd, __CONST_SOCKADDR_ARG __addr, socklen_t
__len)
```

一个新建的 socket 可以直接用于发送 datagram，系统会自动分配合适的端口号，但是如果要接收 datagram 并指定端口，比如本次实验使用的 53 端口，我们就必须 bind 这个 socket 到这个端口上：

```
  bind(listen_fd, (struct sockaddr*)&listen_addr,
sizeof(listen_addr))
```

其中，listen_fd 来自之前创建 socket 的返回值，listen_addr 是一个 `struct sockaddr_in` 类型的变量，用于标识绑定的网段和端口。

recvfrom()

```
ssize_t recvfrom (int __fd, void *__restrict __buf, size_t
__n, int __flags, __SOCKADDR_ARG __addr, socklen_t
*__restrict __addr_len)
```

顾名思义，当 __fd 所标识的 socket 接收到了任何 datagram，那么就会将 datagram 放到 __buf 里，超过 __n 字节的部分直接截断，最后把发送 datagram 的客户端的 IP 地址和端口信息放在 __addr 下，最多不能超过 __addr_len 个字节。

```
  int len = recvfrom(event_fd, buf, MAX_DATAGRAM_BUFFER_SIZE,
0, (struct sockaddr *)&client_info.addr, &client_info.len)
```

recvfrom()

```
ssize_t recvfrom (int __fd, void *__restrict __buf, size_t
__n, int __flags, __SOCKADDR_ARG __addr, socklen_t
*__restrict __addr_len)
```

需要注意的是，如果调用该函数时 `__fd` 标识的 socket 没有接收到任何 datagram，那么这个函数会根据该 `__fd` 的 flag 采用以下策略：

- `O_NONBLOCK`：失败，直接返回 -1
- 默认：阻塞，直到收到任何 datagram

```
  int len = recvfrom(event_fd, buf, MAX_DATAGRAM_BUFFER_SIZE,
0, (struct sockaddr *)&client_info.addr, &client_info.len)
```

epoll_ctl()

```
int epoll_ctl (int __epfd, int __op, int __fd, struct
epoll_event *__event)
```

epoll_wait()

```
int epoll_wait (int __epfd, struct epoll_event *__events, int
__maxevents, int __timeout)
```

这两个函数是用来高效处理来自多个接收状态下的 socket 的 datagram 的，具体原理与本课无关，因此不过多讲解，仅仅只需要知道使用 epoll 机制能够保证每次调用 `recvfrom()` 时一定成功，而不会阻塞或者失败。

close()

```
int close (int __fd)
```

关闭一个已经打开的 fd，以及关联的 socket。